

# Code Template for ACM-ICPC

wodesuck  
SYSU\_Braid  
Sun Yat-sen University

January 1, 2016

# Contents

<b>1</b>	<b>Graph/Tree Theory</b>	<b>3</b>
1.1	Shortest Path	3
1.1.1	Dijkstra	3
1.1.2	SPFA	3
1.1.3	Minimum-weight Cycle(Folyd)	3
1.2	Bridge/Cutvertex-Finding(Tarjan)	4
1.3	Strongly Connected Components(Tarjan)	4
1.4	Lowest Common Ancestor(Tarjan)	5
1.5	Network Flow	5
1.5.1	Maximum Flow(Improved-SAP)	5
1.5.2	Minimum Cost Maximum Flow(Primal-Dual)	6
1.5.3	Minimum Cost Maximum Flow(Cycle Canceling)	7
1.6	Matching	8
1.6.1	Maximum Bipartite Matching(Hungarian)	8
1.6.2	Maximum Weight Perfect Biparite Matching(KM)	8
1.6.3	Maximum Matching on General Graph(Blossom Algorithm)	9
1.6.4	Maximum Weight Perfect Matching on General Graph(Randomize Greedy Matching)	11
1.7	2-SAT	12
1.8	Centroid of a Tree	12
1.9	Heavy-Light Decomposition	13
1.10	Virtual Tree	13
<b>2</b>	<b>Data Structures</b>	<b>15</b>
2.1	Segment Tree	15
2.1.1	Segment Tree(Non-recursive Implement)	15
2.1.2	Functional Segment Tree	16
2.2	Self-balancing BST	16
2.2.1	Size Balanced Tree	16
2.2.2	Treap	18
2.2.3	Splay	19
2.2.4	Functional Treap	19
2.2.5	Functional Treap(Range Operation)	20
2.3	Leftist Tree	21
2.4	Dynamic Tree	21
2.4.1	Link-cut Tree	21
2.4.2	Euler Tour Tree	23
2.4.3	Top Tree	24
2.5	KD Tree	30
2.6	Sparse Table	31
<b>3</b>	<b>Stringology</b>	<b>32</b>
3.1	KMP Algorithm	32
3.2	Extend-KMP Algorithm	32
3.3	Aho-Corasick Automation	33
3.4	Suffix Array	34
3.5	Suffix Automation	35
3.6	Longest Palindorme Substring(Manacher)	36
3.7	Palindromic Tree	36
3.8	Minimum Representation	37

<b>4</b>	<b>Computational Geometry</b>	<b>38</b>
4.1	Basic Operations . . . . .	38
4.1.1	Line . . . . .	38
4.1.2	Triangle . . . . .	39
4.1.3	Circle . . . . .	40
4.2	Point in Polygon Problem . . . . .	41
4.3	Convex Hull(Graham) . . . . .	42
4.4	Dynamic Convex Hull . . . . .	42
4.5	Half-plane Intersection . . . . .	43
4.6	Closest Pair(Divide and Conquer) . . . . .	43
4.7	Farthest Pair(Rotating Caliper) . . . . .	44
4.8	Minimum Distance Between Convex Hull(Rotating Caliper) . . . . .	44
4.9	Union Area of a Circle and a Polygon . . . . .	44
4.10	Union Area of Circles . . . . .	45
4.11	Union Area of Polygons . . . . .	45
4.12	Minimum Enclosing Circle(Randomized Incremental Method) . . . . .	46
4.13	Planar Straight-line Graph(PSLG) . . . . .	47
4.14	3D Computational Geometry . . . . .	47
4.14.1	Line . . . . .	48
4.14.2	Sphere . . . . .	49
4.14.3	3D Transformation Matrix . . . . .	50
4.15	Convex Hull in 3D . . . . .	51
4.16	Half-space Intersection . . . . .	51
<b>5</b>	<b>Number Theory</b>	<b>53</b>
5.1	Fast Fourier Transform . . . . .	53
5.2	Primality Test(Miller-Rabin) . . . . .	53
5.3	Integer Factorization(Pollard's $\rho$ Algorithm) . . . . .	54
5.4	Extended Euclid's Algorithm . . . . .	54
5.5	Euler's $\varphi$ Function . . . . .	54
<b>6</b>	<b>Others</b>	<b>56</b>
6.1	Exact Cover(DLX) . . . . .	56
6.2	Fuzzy Cover(DLX) . . . . .	57
6.3	3D Partial Order(Divide and Conquer) . . . . .	58
6.4	Power of Matrix . . . . .	59
6.5	Cantor Pairing Function . . . . .	59
6.6	Adaptive Simpson's Method . . . . .	60
6.7	Linear Programming(Simplex) . . . . .	60
<b>A</b>	<b>Snippets</b>	<b>62</b>
<b>B</b>	<b>Java Example</b>	<b>63</b>
<b>C</b>	<b>Vim Configuration</b>	<b>64</b>

# Chapter 1

## Graph/Tree Theory

### 1.1 Shortest Path

#### 1.1.1 Dijkstra

```
1 void dijkstra(int s)
2 {
3     typedef pair<int, int> T;
4     priority_queue<T, vector<T>, greater<T> > h;
5     memset(d, 0x3f, sizeof(d));
6     memset(v, 0, sizeof(v));
7     h.push(T(d[s] = 0, s));
8     while (!h.empty()) {
9         int w = h.top().first, u = h.top().second;
10        h.pop();
11        if (w > d[u]) continue;
12        for (edge *i = e[u]; i; i = i->next) {
13            int dis = d[u] + i->w;
14            if (dis < d[i->t]) h.push(T(d[i->t] = dis, i->t));
15        }
16    }
17 }
```

#### 1.1.2 SPFA

```
1 void spfa(int s)
2 {
3     queue<int> q;
4     memset(d, 0x3f, sizeof(d));
5     memset(v, 0, sizeof(v));
6     q.push(s); d[s] = 0; v[s] = true;
7     while (!q.empty()) {
8         int u = q.front(); q.pop(); v[u] = false;
9         for (edge *i = e[u]; i; i = i->next) {
10            if (d[u] + i->w < d[i->t]) {
11                d[i->t] = d[u] + i->w;
12                if (!v[i->t]) {
13                    q.push(i->t);
14                    v[i->t] = true;
15                }
16            }
17        }
18    }
19 }
```

#### 1.1.3 Minimum-weight Cycle(Folyd)

```

1 // for undirected graph
2 const int INF = 0x2a2a2a2a;
3
4 int folyd()
5 {
6     int ans = INF;
7     for (int k = 0; k < n; ++k) {
8         for (int i = 0; i < k; ++i) {
9             for (int j = 0; j < i; ++j) {
10                 ans = min(ans, f[i][j] + g[j][k] + g[k][i]);
11             }
12         }
13         for (int i = 0; i < n; ++i) {
14             for (int j = 0; j < n; ++j) {
15                 f[i][j] = min(f[i][j], f[i][k] + f[k][j]);
16             }
17         }
18     }
19     return ans;
20 }
21
22 /*
23 Initialize:
24     memset(g, 0x2a, sizeof(g));
25     memset(f, 0x2a, sizeof(f));
26 */

```

## 1.2 Bridge/Cutvertex-Finding(Tarjan)

```

1 void tarjan(int u, int fa)
2 {
3     dfn[u] = low[u] = ++stamp;
4     int ch = 0;
5     for (edge *i = e[u]; i; i = i->next) {
6         int v = i->t;
7         if (!dfn[v]) {
8             tarjan(v, u);
9             low[u] = min(low[u], low[v]);
10            if (u ? low[v] >= dfn[u] : ++ch > 1) cut[u] = true;
11            if (low[v] > dfn[u]) bridge[u][v] = true;
12        } else if (v != fa) {
13            low[u] = min(low[u], dfn[v]);
14        }
15    }
16 }

```

## 1.3 Strongly Connected Components(Tarjan)

```

1 void tarjan(int u)
2 {
3     dfn[u] = low[u] = ++stamp;
4     sta[top++] = u; ins[u] = true;
5     for (edge *i = e[u]; i; i = i->next) {
6         int v = i->t;
7         if (!dfn[v]) {
8             tarjan(v);
9             low[u] = min(low[u], low[v]);
10        } else if (ins[v]) {
11            low[u] = min(low[u], dfn[v]);
12        }
13    }
14 }

```

```

13     }
14     if (dfn[u] == low[u]) {
15         int v;
16         do {
17             v = sta[--top];
18             ins[v] = false;
19             scc[v] = cnt;
20         } while (v != u);
21         ++cnt;
22     }
23 }

```

## 1.4 Lowest Common Ancestor(Tarjan)

```

1 void tarjan(int u)
2 {
3     anc[u] = u; v[u] = 1;
4     for (edge *i = e[u]; i; i = i->next) {
5         if (!v[i->t]) {
6             tarjan(i->t);
7             join(u, i->t);
8             anc[find(u)] = u;
9         }
10    }
11    v[u] = 2;
12    for (quest *i = q[u]; i; i = i->next) {
13        if (v[i->t] == 2) lca[i->id] = anc[find(i->t)];
14    }
15 }

```

## 1.5 Network Flow

### 1.5.1 Maximum Flow(Improved-SAP)

```

1 struct edge {
2     int t, u;
3     edge *next, *pair;
4 }epool[MAXE * 2], *e[MAXV];
5 int esz, psz, s, t;
6 int h[MAXV], vh[MAXV + 1];
7
8 void addedge(int u, int v, int c)
9 {
10     edge *e1 = epool + psz++, *e2 = epool + psz++;
11     *e1 = (edge){v, c, e[u], e2}, e[u] = e1;
12     *e2 = (edge){u, 0, e[v], e1}, e[v] = e2;
13 }
14
15 int aug(int u, int m)
16 {
17     if (u == t) return m;
18     int d = m;
19     for (edge *i = e[u]; i; i = i->next) {
20         if (i->u && h[u] == h[i->t] + 1) {
21             int f = aug(i->t, min(i->u, d));
22             i->u -= f, i->pair->u += f, d -= f;
23             if (h[s] == esz || !d) return m - d;
24         }
25     }
26     int w = d < m ? min(esz, h[u] + 2) : esz;

```

```

27     for (edge *i = e[u]; i; i = i->next) {
28         if (i->u) w = min(w, h[i->t] + 1);
29     }
30     ++vh[w];
31     --vh[h[u]] ? h[u] = w : h[s] = esz;
32     return m - d;
33 }
34
35 void maxflow()
36 {
37     flow = 0;
38     memset(h, 0, sizeof(h));
39     memset(vh, 0, sizeof(vh));
40     vh[0] = esz;
41     while (h[s] != esz) flow += aug(s, INT_MAX);
42 }

```

## 1.5.2 Minimum Cost Maximum Flow(Primal-Dual)

```

1 struct edge {
2     int t, u, c;
3     edge *next, *pair;
4 }epool[MAXE * 2], *e[MAXV];
5 int psz, s, t;
6 int cost, dist, d[MAXV];
7 bool vis[MAXV];
8
9 void addedge(int u, int v, int c, int w)
10 {
11     edge *e1 = epool + psz++, *e2 = epool + psz++;
12     *e1 = (edge){v, c, w, e[u], e2}, e[u] = e1;
13     *e2 = (edge){u, 0, -w, e[v], e1}, e[v] = e2;
14 }
15
16 int aug(int u, int m)
17 {
18     if (u == t) return cost += dist * m, m;
19     int d = m; vis[u] = true;
20     for (edge *i = e[u]; i; i = i->next) {
21         if (i->u && !i->c && !vis[i->t]) {
22             int f = aug(i->t, min(d, i->u));
23             i->u -= f, i->pair->u += f, d -= f;
24             if (!d) return m;
25         }
26     }
27     return m - d;
28 }
29
30 bool modlabel()
31 {
32     deque<int> q;
33     memset(vis, 0, sizeof(vis));
34     memset(d, 0x3f, sizeof(d));
35     q.push_back(s); d[s] = 0; vis[s] = true;
36     while (!q.empty()) {
37         int u = q.front(); q.pop_front(); vis[u] = false;
38         for (edge *i = e[u]; i; i = i->next) {
39             int v = i->t;
40             if (i->u && d[u] + i->c < d[v]) {
41                 d[v] = d[u] + i->c;
42                 if (vis[v]) continue;

```

```

43         vis[v] = true;
44         if (q.size() && d[v] < d[q[0]]) q.push_front(v);
45         else q.push_back(v);
46     }
47 }
48 }
49 for (edge *i = epool; i < epool + psz; ++i) {
50     i->c -= d[i->t] - d[i->pair->t];
51 }
52 dist += d[t];
53 return d[t] < inf;
54 }
55
56 void costflow()
57 {
58     cost = dist = 0;
59     while (modlabel()) {
60         do memset(vis, 0, sizeof(vis));
61         while (aug(s, INT_MAX));
62     }
63 }

```

### 1.5.3 Minimum Cost Maximum Flow(Cycle Canceling)

```

1 struct edge {
2     int t, u, c;
3     edge *next, *pair;
4 }epool[MAXE * 2], *e[MAXV];
5 int psz, s, t;
6 int d[MAXV];
7 bool vis[MAXV];
8 edge *fa[MAXV];
9
10 void addedge(int u, int v, int c, int w)
11 {
12     edge *e1 = epool + psz++, *e2 = epool + psz++;
13     *e1 = (edge){v, c, w, e[u], e2}, e[u] = e1;
14     *e2 = (edge){u, 0, -w, e[v], e1}, e[v] = e2;
15 }
16
17 void cancelcycle(int u)
18 {
19     int i = u;
20     do {
21         --fa[i]->u, ++fa[i]->pair->u, cost += fa[i]->c;
22         i = fa[i]->pair->t;
23     } while (i != u);
24 }
25
26 bool aug(int u)
27 {
28     vis[u] = true;
29     for (edge *i = e[u]; i; i = i->next) {
30         int v = i->t;
31         if (i->u && d[u] + i->c < d[v]) {
32             d[v] = d[u] + i->c;
33             fa[v] = i;
34             if (vis[v]) cancelcycle(v);
35             if (vis[v] || aug(v)) return true;
36         }
37     }

```



```

38         vis[u] = false;
39         return false;
40     }
41
42     void costflow()
43     {
44         cost = 0;
45         for (;;) {
46             memset(d, 0, sizeof(d));
47             memset(vis, 0, sizeof(vis));
48             bool flag = false;
49             for (int i = 0; i < esz; ++i) {
50                 if (aug(i)) { flag = true; break; }
51             }
52             if (!flag) return;
53         }
54     }
55
56     /*
57     Initialize:
58         addedge(t, s, inf, -inf);
59         CAUTION: maybe OVERFLOW
60     */

```

## 1.6 Matching

### 1.6.1 Maximum Bipartite Matching(Hungarian)

```

1  int n, m;
2  bool g[MAXN][MAXM];
3  int match[MAXM];
4  bool v[MAXN];
5
6  bool dfs(int i)
7  {
8      for (int j = 0; j < m; ++j) {
9          if (g[i][j] && !v[j]) {
10             v[j] = true;
11             if (match[j] < 0 || dfs(match[j])) {
12                 match[j] = i;
13                 return true;
14             }
15         }
16     }
17     return false;
18 }
19
20 int hungarian()
21 {
22     int c = 0;
23     memset(match, -1, sizeof(match));
24     for (int i = 0; i < n; ++i) {
25         memset(v, 0, sizeof(v));
26         if (dfs(i)) ++c;
27     }
28     return c;
29 }

```

### 1.6.2 Maximum Weight Perfect Bipartite Matching(KM)

```

1  int n;

```

```

2 | int w[MAXN][MAXN];
3 | int lx[MAXN], ly[MAXN], match[MAXN], slack[MAXN];
4 | bool vx[MAXN], vy[MAXN];
5 |
6 | bool dfs(int i)
7 | {
8 |     vx[i] = true;
9 |     for (int j = 0; j < n; ++j) {
10 |         if (lx[i] + ly[j] > w[i][j]) {
11 |             slack[j] = min(slack[j], lx[i] + ly[j] - w[i][j]);
12 |         } else if (!vy[j]) {
13 |             vy[j] = true;
14 |             if (match[j] < 0 || dfs(match[j])) {
15 |                 match[j] = i;
16 |                 return true;
17 |             }
18 |         }
19 |     }
20 |     return false;
21 | }
22 |
23 | void km()
24 | {
25 |     memset(match, -1, sizeof(match));
26 |     memset(ly, 0, sizeof(ly));
27 |     for (int i = 0; i < n; ++i) lx[i] = *max_element(w[i], w[i] + n);
28 |     for (int i = 0; i < n; ++i) {
29 |         for (;;) {
30 |             memset(vx, 0, sizeof(vx));
31 |             memset(vy, 0, sizeof(vy));
32 |             memset(slack, 0x3f, sizeof(slack));
33 |             if (dfs(i)) break;
34 |             int d = inf;
35 |             for (int i = 0; i < n; ++i) {
36 |                 if (!vy[i]) d = min(d, slack[i]);
37 |             }
38 |             for (int i = 0; i < n; ++i) {
39 |                 if (vx[i]) lx[i] -= d;
40 |                 if (vy[i]) ly[i] += d;
41 |             }
42 |         }
43 |     }
44 | }

```

### 1.6.3 Maximum Matching on General Graph(Blossom Algorithm)

```

1 | int n;
2 | int next[MAXN], match[MAXN], v[MAXN], f[MAXN];
3 | int que[MAXN], head, tail;
4 |
5 | int find(int p)
6 | {
7 |     return f[p] < 0 ? p : f[p] = find(f[p]);
8 | }
9 |
10 | void join(int x, int y)
11 | {
12 |     x = find(x); y = find(y);
13 |     if (x != y) f[x] = y;
14 | }
15 |

```

```

16 int lca(int x, int y)
17 {
18     static int v[MAXN], stamp = 0;
19     ++stamp;
20     for (;;) {
21         if (x >= 0) {
22             x = find(x);
23             if (v[x] == stamp) return x;
24             v[x] = stamp;
25             if (match[x] >= 0) x = next[match[x]];
26             else x = -1;
27         }
28         swap(x, y);
29     }
30 }
31
32 void group(int a, int p)
33 {
34     while (a != p) {
35         int b = match[a], c = next[b];
36         if (find(c) != p) next[c] = b;
37         if (v[b] == 2) v[que[tail++]] = b;
38         if (v[c] == 2) v[que[tail++]] = c;
39         join(a, b); join(b, c);
40         a = c;
41     }
42 }
43
44 void aug(int s)
45 {
46     memset(v, 0, sizeof(v));
47     memset(f, -1, sizeof(f));
48     memset(next, -1, sizeof(next));
49     que[0] = s; head = 0; tail = 1; v[s] = 1;
50     while (head < tail && match[s] < 0) {
51         int x = que[head++];
52         for (edge *i = e[x]; i; i = i->next) {
53             int y = i->t;
54             if (match[x] == y || v[y] == 2 || find(x) == find(y)) {
55                 continue;
56             } else if (v[y] == 1) {
57                 int p = lca(x, y);
58                 if (find(x) != p) next[x] = y;
59                 if (find(y) != p) next[y] = x;
60                 group(x, p);
61                 group(y, p);
62             } else if (match[y] < 0) {
63                 next[y] = x;
64                 while (~y) {
65                     int z = next[y];
66                     int p = match[z];
67                     match[y] = z; match[z] = y;
68                     y = p;
69                 }
70                 break;
71             } else {
72                 next[y] = x;
73                 v[que[tail++]] = match[y] = 1;
74                 v[y] = 2;
75             }
76         }
77     }
78 }

```

```

77     }
78 }
79
80 void blossom()
81 {
82     memset(match, -1, sizeof(match));
83     for (int i = 0; i < n; ++i) {
84         if (match[i] < 0) aug(i);
85     }
86 }

```

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

```

1  int n;
2  int w[MAXN][MAXN];
3  int match[MAXN], p[MAXN], d[MAXN];
4  int path[MAXN], len;
5  bool v[MAXN];
6  const int inf = 0x3f3f3f3f;
7
8  bool dfs(int i)
9  {
10     path[len++] = i;
11     if (v[i]) return true;
12     v[i] = true;
13     for (int j = 0; j < n; ++j) {
14         if (i != j && match[i] != j && !v[j]) {
15             int k = match[j];
16             if (d[k] < d[i] + w[i][j] - w[j][k]) {
17                 d[k] = d[i] + w[i][j] - w[j][k];
18                 if (dfs(k)) return true;
19             }
20         }
21     }
22     --len;
23     v[i] = false;
24     return false;
25 }
26
27 int matching()
28 {
29     for (int i = 0; i < n; ++i) p[i] = i, match[i] = i^1;
30     int cnt = 0;
31     for (;;) {
32         len = 0;
33         bool flag = false;
34         memset(d, 0, sizeof(d));
35         memset(v, 0, sizeof(v));
36         for (int i = 0; i < n; ++i) {
37             if (dfs(p[i])) {
38                 flag = true;
39                 int t = match[path[len - 1]], j = len - 2;
40                 while (path[j] != path[len - 1]) {
41                     match[t] = path[j];
42                     swap(t, match[path[j]]);
43                     --j;
44                 }
45                 match[t] = path[j];
46                 match[path[j]] = t;
47                 break;

```

```

48         }
49     }
50     if (!flag) {
51         if (++cnt >= 3) break;
52         random_shuffle(p, p + n);
53     }
54 }
55 }

```

## 1.7 2-SAT

```

1 // n vars
2 // sat variable Bi and !Bi are encoded as i<<1 and i<<1^1
3 // add edge (i<<1 -> j<<1^1) for Bi -> !Bj
4 // Bi is true if scc[i<<1] < scc[i<<1^1]
5
6 bool twosat()
7 {
8     cnt = stamp = 0;
9     memset(dfn, 0, sizeof(dfn));
10    for (int i = 0; i < n<<1; ++i) if (!dfn[i]) tarjan(i);
11    for (int i = 0; i < n; ++i) {
12        if (scc[i<<1] == scc[i<<1^1]) return false;
13        ans[i] = scc[i<<1] < scc[i<<1^1];
14    }
15    return true;
16 }

```

## 1.8 Centroid of a Tree

```

1 int getsize(int u, int fa)
2 {
3     size[u] = 1;
4     for (edge *i = e[u]; i; i = i->next) {
5         if (i->t != fa) size[u] += getsize(i->t, u);
6     }
7     return size[u];
8 }
9
10 int divide(int u)
11 {
12     for (edge *i = e[u]; i; i = i->next) {
13         if (size[i->t] > size[u] / 2) {
14             size[u] -= size[i->t], size[i->t] += size[u];
15             return divide(i->t);
16         }
17     }
18     return u;
19 }
20
21 void solve(int u) // Divide and Conquer for Tree
22 {
23     u = divide(u);
24     size[u] = 0; // delete
25     for (edge *i = e[u]; i; i = i->next) {
26         if (size[i->t]) {
27             dfs1(i->t, u); // calculate answer
28             dfs2(i->t, u); // update
29         }
30     }

```

```

31 |         // calculate answer with root
32 |         for (edge *i = e[u]; i; i = i->next) {
33 |             if (size[i->t]) solve(i->t);
34 |         }
35 | }

```

## 1.9 Heavy-Light Decomposition

```

1 | int fa[MAXN], dep[MAXN], size[MAXN], hson[MAXN], top[MAXN];
2 | int dfn[MAXN], stamp;
3 |
4 | void dfs1(int u)
5 | {
6 |     size[u] = 1, hson[u] = 0;
7 |     for (edge *i = e[p]; i; i = i->next) {
8 |         int v = i->t;
9 |         if (v == fa[u]) continue;
10 |        fa[v] = u;
11 |        dep[v] = dep[u] + 1;
12 |        dfs1(v);
13 |        size[u] += size[v];
14 |        if (!hson[u] || size[v] > size[hson[u]]) hson[u] = v;
15 |    }
16 | }
17 |
18 | void dfs2(int u, int anc)
19 | {
20 |     dfn[u] = stamp++;
21 |     top[u] = anc;
22 |     if (hson[u]) dfs2(hson[u], anc);
23 |     for (edge *i = e[p]; i; i = i->next) {
24 |         int v = i->t;
25 |         if (v != fa[u] && v != hson[u]) dfs2(v, v);
26 |     }
27 | }
28 |
29 | int lca(int u, int v)
30 | {
31 |     while (top[u] != top[v]) {
32 |         if (dep[top[u]] < dep[top[v]]) swap(u, v);
33 |         // query(dfn[top[u]], dfn[u])
34 |         u = fa[top[u]];
35 |     }
36 |     if (dep[u] > dep[v]) swap(u, v);
37 |     // query(dfn[u], dfn[v]) -- include LCA
38 |     // if (u != v) query(dfn[u] + 1, dfn[v]) -- exclude LCA
39 |     return u;
40 | }

```

## 1.10 Virtual Tree

```

1 | bool cmp(int i, int j) { return dfn[i] < dfn[j]; }
2 |
3 | int vtree(int h[], int m, int T[], int fa[])
4 | {
5 |     static int sta[MAXN];
6 |     int tot = 0, top = 0;
7 |     sort(h, h + m, cmp);
8 |     sta[top++] = 0; // d[0] == -1
9 |     for (int i = 0; i < m; ++i) {

```

```

10         if (top <= 1) {
11             sta[top++] = h[i];
12             fa[h[i]] = 0;
13         } else {
14             int g = lca(h[i], sta[top - 1]);
15             while (d[sta[top - 1]] > d[g]) {
16                 --top;
17                 if (d[sta[top - 1]] <= d[g]) fa[sta[top]] = g;
18             }
19             if (sta[top - 1] != g) {
20                 T[tot++] = g;
21                 fa[g] = sta[top - 1];
22                 sta[top++] = g;
23             }
24             fa[h[i]] = g;
25             sta[top++] = h[i];
26         }
27         T[tot++] = h[i];
28     }
29     sort(T, T + tot, cmp);
30     return tot;
31 }
32
33 // return the number of nodes in virtual tree
34 // T[] -- nodes, fa[] -- father in virtual tree

```

## Chapter 2

# Data Structures

### 2.1 Segment Tree

#### 2.1.1 Segment Tree(Non-recursive Implement)

```
1 | int n, h;
2 | S T[MAXN * 2]; // val
3 | int d[MAXN * 2]; // lazy flag
4 |
5 | void push(int p)
6 | {
7 |     for (int s = h, k = 1<<(h-1); s; --s, k >>= 1) {
8 |         int i = p >> s;
9 |         if (d[i]) {
10 |             apply(i<<1, k, d[i]);
11 |             apply(i<<1|1, k, d[i]);
12 |             d[i] = 0;
13 |         }
14 |     }
15 | }
16 |
17 | void build()
18 | {
19 |     for (int i = n - 1; i; --i) update(i);
20 | }
21 |
22 | S query(int l, int r)
23 | {
24 |     S L, R;
25 |     push(l += n), push(r += n);
26 |     for (; l <= r; l >>= 1, r >>= 1) {
27 |         if (l&1) L = merge(L, T[l++]);
28 |         if (~r&1) R = merge(T[r--], R);
29 |     }
30 |     return merge(L, R);
31 | }
32 |
33 | void modify(int l, int r, int x)
34 | {
35 |     bool cl = false, cr = false;
36 |     push(l += n), push(r += n);
37 |     for (int k = 1; l <= r; l >>= 1, r >>= 1, k <<= 1) {
38 |         if (cl) update(l - 1);
39 |         if (cr) update(r + 1);
40 |         if (l&1) apply(l++, k, x), cl = true;
41 |         if (~r&1) apply(r--, k, x), cr = true;
```



```

42     }
43     for (--l, ++r; r; l >>= 1, r >>= 1) {
44         if (cl) update(l);
45         if (cr && (!cl || l != r)) update(r);
46     }
47 }
48
49 // h = sizeof(int) * 8 - __builtin_clz(n);

```

## 2.1.2 Functional Segment Tree

```

1 struct sgt {
2     int sum;
3     sgt *left, *right;
4 }tpool[PSZ];
5 int tpsz;
6
7 sgt *new_node(int sum)
8 {
9     sgt *p = tpool + tpsz++;
10    p->sum = sum;
11    p->left = p->right = 0;
12    return p;
13 }
14
15 sgt *merge(sgt *l, sgt *r)
16 {
17     sgt *p = tpool + tpsz++;
18     p->sum = l->sum + r->sum;
19     p->left = l; p->right = r;
20     return p;
21 }
22
23 sgt *build(int l, int r)
24 {
25     if (l == r) return new_node(0);
26     int mid = (l + r) >> 1;
27     return merge(build(l, mid), build(mid + 1, r));
28 }
29
30 sgt *add(sgt *p, int l, int r, int x)
31 {
32     if (l == r) return new_node(p->sum + 1);
33     int mid = (l + r) >> 1;
34     return x <= mid ? merge(add(p->left, l, mid, x), p->right)
35                     : merge(p->left, add(p->right, mid + 1, r, x));
36 }
37
38 int kth(sgt *a, sgt *b, int l, int r, int k)
39 {
40     if (l == r) return l;
41     int mid = (l + r) >> 1;
42     int lsum = a->left->sum - b->left->sum;
43     return k <= lsum ? kth(a->left, b->left, l, mid, k)
44                     : kth(a->right, b->right, mid + 1, l, k - lsum);
45 }

```

## 2.2 Self-balancing BST

### 2.2.1 Size Balanced Tree

```

1 struct sbt {
2     int k, sz;
3     sbt *ch[2];
4 }pool[MAXN], *null;
5 int psz;
6
7 sbt *new_sbt(int v)
8 {
9     sbt *t = pool + psz++;
10    t->k = v; t->sz = 1;
11    t->ch[0] = t->ch[1] = null;
12    return t;
13 }
14
15 void rot(sbt *&t, int i)
16 {
17     sbt *k = t->ch[i^1];
18     t->ch[i^1] = k->ch[i]; k->ch[i] = t;
19     k->sz = t->sz; t->sz = t->ch[0]->sz + t->ch[1]->sz + 1;
20     t = k;
21 }
22
23 void maintain(sbt *&t, int i)
24 {
25     if (t->ch[i]->ch[i]->sz > t->ch[i^1]->sz) {
26         rot(t, i^1);
27     } else if (t->ch[i]->ch[i^1]->sz > t->ch[i^1]->sz) {
28         rot(t->ch[i], i), rot(t, i^1);
29     } else return;
30     maintain(t->ch[0], 0);
31     maintain(t->ch[1], 1);
32     maintain(t, 0);
33     maintain(t, 1);
34 }
35
36 void insert(sbt *&t, int v)
37 {
38     if (t == null) { t = new_sbt(v); return; }
39     ++t->sz;
40     insert(t->ch[v > t->k], v);
41     maintain(t, v > t->k);
42 }
43
44 int erase(sbt *&t, int v)
45 {
46     --t->sz;
47     if (v == t->k || t->ch[v > t->k] == null) {
48         v = t->k;
49         if (t->ch[0] == null) t = t->ch[1];
50         else if (t->ch[1] == null) t = t->ch[0];
51         else t->k = erase(t->ch[0], v + 1);
52         return v;
53     }
54     return erase(t->ch[v > t->k], v);
55 }
56
57 sbt *find(sbt *t, int v)
58 {
59     if (t == null) return 0;
60     if (v == t->k) return t;
61     return find(t->ch[v > t->k], v);

```

```

62 }
63
64 int rank(sbt *t, int v)
65 {
66     if (t == null) return 0;
67     else if (v < t->k) return rank(t->ch[0], v);
68     else return t->ch[0]->sz + 1 + rank(t->ch[1], v);
69 }
70
71 sbt *select(int t, int k)
72 {
73     if (k == t->ch[0]->sz + 1) return t;
74     else if (k <= t->ch[0]->sz) return select(t->ch[0], k);
75     else return select(t->ch[1], k - t->ch[0]->sz - 1);
76 }
77
78 void init()
79 {
80     psz = 0;
81     null = new_sbt(0);
82     null->sz = 0, null->ch[0] = null->ch[1] = null;
83 }

```

## 2.2.2 Treap

```

1 struct treap {
2     treap *l, *r;
3     int w, size;
4     void update() { size = l->size + r->size + 1; }
5     void sink() {}
6 } node[MAXN], *null;
7
8 void split(treap *t, int k, treap *&l, treap *&r) // split first k elements
9 {
10     if (k == 0) { l = null, r = t; return; }
11     t->sink();
12     if (k <= t->l->size) {
13         split(t->l, k, l, r);
14         t->l = r, r = t;
15     } else {
16         split(t->r, k - t->l->size - 1, l, r);
17         t->r = l, l = t;
18     }
19     t->update();
20 }
21
22 treap *merge(treap *l, treap *r)
23 {
24     if (l == null) return r;
25     if (r == null) return l;
26     l->sink(), r->sink();
27     treap *t;
28     if (l->w < r->w) {
29         t = l, l->r = merge(l->r, r);
30     } else {
31         t = r, r->l = merge(l, r->l);
32     }
33     t->update();
34     return t;
35 }

```

### 2.2.3 Splay

```
1 struct node_t *null, *root;
2 struct node_t {
3     node_t *ch[2], *fa;
4     int size;
5
6     int dir() { return fa->ch[0] == this ? 0 : 1; }
7     void setc(node_t *c, int d) { ch[d] = c; if (c != null) c->fa = this; }
8     void update() { size = ch[0]->size + ch[1]->size + 1; }
9     void sink() {}
10
11     void rot()
12     {
13         node_t *p = fa;
14         int d = dir();
15         if (p->fa == null) fa = null, root = this;
16         else p->fa->setc(this, p->dir());
17         p->setc(ch[d^1], d), setc(p, d^1);
18         p->update(), update();
19     }
20
21     void splay(node_t *header = null)
22     {
23         for (; fa != header; rot()) {
24             if (fa->fa != header) {
25                 if (dir() == fa->dir()) fa->rot();
26                 else rot();
27             }
28         }
29     }
30
31     node_t *select(int k)
32     {
33         node_t *t = this;
34         while (t->sink(), k != t->ch[0]->size + 1) {
35             if (k <= t->ch[0]->size) t = t->ch[0];
36             else k -= t->ch[0]->size + 1, t = t->ch[1];
37         }
38         t->splay(fa);
39         return t;
40     }
41
42     node_t *select(int l, int r)
43     {
44         return select(r + 1)->ch[0]->select(l - 1)->ch[1];
45     }
46 }node[MAXN];
```

### 2.2.4 Functional Treap

```
1 struct node {
2     int k, w; // key, weight
3     node *l, *r;
4 }pool[PSZ];
5 int psz;
6
7 node *new_node(int key, int weight, node *left, node *right)
8 {
9     node *t = pool + psz++;
10    t->k = key; t->w = weight; t->l = left; t->r = right;
```

```

11         return t;
12     }
13
14     node *split_l(node *t, int key)
15     {
16         return !t ? 0 : (key < t->k ? split_l(t->l, key) :
17             new_node(t->k, t->w, t->l, split_l(t->r, key)));
18     }
19
20     node *split_r(node *t, int key)
21     {
22         return !t ? 0 : (key >= t->k ? split_r(t->r, key) :
23             new_node(t->k, t->w, split_r(t->l, key), t->r));
24     }
25
26     node *merge(node *a, node *b)
27     {
28         return (!a || !b) ? (a ? a : b) : (a->w < b->w ?
29             new_node(a->k, a->w, a->l, merge(a->r, b)) :
30             new_node(b->k, b->w, merge(a, b->l), b->r));
31     }
32
33     node *insert(node *t, int key)
34     {
35         return merge(merge(split_l(t, key), new_node(key, rand(), 0, 0)),
36             split_r(t, key));
37     }

```

### 2.2.5 Functional Treap(Range Operation)

```

1 struct node {
2     int v, w, sz; // value, weight, size
3     node *l, *r;
4 }pool[PSZ];
5 int psz;
6
7 inline int sz(node *t) { return t ? t->sz : 0; }
8
9 node *new_node(int val, int weight, node *left, node *right)
10 {
11     node *t = pool + psz++;
12     t->v = val; t->w = weight; t->l = left; t->r = right;
13     t->sz = sz(left) + sz(right) + 1;
14     return t;
15 }
16
17 node *split_l(node *t, int k) // get the first k elements
18 {
19     return !t ? 0 :
20         (k <= sz(t->l) ? split_l(t->l, k) :
21             new_node(t->v, t->w, t->l, split_l(t->r, k - sz(t->l) - 1)));
22 }
23
24 node *split_r(node *t, int k)
25 {
26     return !t ? 0 :
27         (k > sz(t->l) ? split_r(t->r, k - sz(t->l) - 1) :
28             new_node(t->v, t->w, split_r(t->l, k), t->r));
29 }
30
31 node *merge(node *a, node *b)

```

```

32 {
33     return (!a || !b) ? (a ? a : b) :
34         (a->w < b->w ?
35             new_node(a->v, a->w, a->l, merge(a->r, b)) :
36             new_node(b->v, b->w, merge(a, b->l), b->r));
37 }
38
39 node *insert(node *t, int pos, int *val, int n) // insert before pos
40 {
41     node *l = split_l(t, pos), *r = split_r(t, pos);
42     for (int i = 0; i < n; ++i) {
43         l = merge(l, new_node(val[i], rand(), 0, 0));
44     }
45     return merge(l, r);
46 }
47
48 node *fetch(node *t, int l, int r) // fetch [l, r]
49 {
50     return split_l(split_r(t, l), r - l + 1);
51 }
52
53 // index from 0

```

## 2.3 Leftist Tree

```

1 int n;
2 int key[MAXN], left[MAXN], right[MAXN], dist[MAXN];
3
4 int merge(int a, int b)
5 {
6     if (!a) return b;
7     if (!b) return a;
8     if (key[b] > key[a]) swap(a, b);
9     right[a] = merge(right[a], b);
10    if (dist[left[a]] < dist[right[a]]) swap(left[a], right[a]);
11    dist[a] = dist[right[a]] + 1;
12    return a;
13 }
14
15 /*
16 Initialize:
17     memset(left, 0, sizeof(left));
18     memset(right, 0, sizeof(right));
19     dist[0] = -1;
20 */

```

## 2.4 Dynamic Tree

### 2.4.1 Link-cut Tree

```

1 struct node_t {
2     node_t *ch[2], *fa;
3     int val, mx;
4     bool rev;
5
6     bool isroot() { return !fa || (fa->ch[0] != this && fa->ch[1] != this); }
7     int dir() { return fa->ch[0] == this ? 0 : 1; }
8     void setc(node_t *c, int d) { ch[d] = c; if (c) c->fa = this; }
9     void reverse() { rev ^= 1; swap(ch[0], ch[1]); }
10 }

```

```

11 void init(int v)
12 {
13     ch[0] = ch[1] = fa = 0;
14     rev = false;
15     val = mx = v;
16 }
17
18 void update()
19 {
20     mx = val;
21     if (ch[0]) mx = max(mx, ch[0]->mx);
22     if (ch[1]) mx = max(mx, ch[1]->mx);
23 }
24
25 void sink()
26 {
27     if (rev) {
28         if (ch[0]) ch[0]->reverse();
29         if (ch[1]) ch[1]->reverse();
30         rev = 0;
31     }
32 }
33
34 void rot()
35 {
36     node_t *p = fa;
37     int d = dir();
38     if (p->isroot()) fa = p->fa;
39     else p->fa->setc(this, p->dir());
40     p->setc(ch[d^1], d), setc(p, d^1);
41     p->update(), update();
42 }
43
44 void sinkdown()
45 {
46     if (!isroot()) fa->sinkdown();
47     sink();
48 }
49
50 void splay()
51 {
52     sinkdown();
53     for (; !isroot(); rot()) {
54         if (!fa->isroot()) {
55             if (dir() == fa->dir()) fa->rot();
56             else rot();
57         }
58     }
59 }
60
61 node_t *expose()
62 {
63     node_t *u = 0, *t = this;
64     for (; t; u = t, t = t->fa) {
65         t->splay();
66         t->ch[1] = u;
67         t->update();
68     }
69     return u;
70 }
71

```

```

72     node_t *root()
73     {
74         node_t *t = expose();
75         while (t->sink(), t->ch[0]) t = t->ch[0];
76         return t;
77     }
78
79     void setroot()
80     {
81         expose()->reverse();
82     }
83
84     void link(node_t *p)
85     {
86         setroot(); // have no need for rooted tree
87         expose()->fa = p;
88     }
89
90     void cut(node_t *p)
91     {
92         p->setroot(); // have no need for rooted tree
93         expose();
94         splay();
95         ch[0] = ch[0]->fa = 0;
96         update();
97     }
98
99     int query(node_t *p)
100    {
101        p->setroot();
102        return expose()->mx;
103    }
104
105    int query(node_t *t) // without setroot
106    {
107        expose();
108        t = t->expose(); // lca
109        int ret = t->val; // analysis lca
110        if (t->ch[1]) ret = max(ret, t->ch[1]->mx); // lca -> v
111        if (t != this) {
112            splay();
113            ret = max(ret, mx); // lca -> u
114        }
115        return ret;
116    }
117 }node[MAXN];

```

### 2.4.2 Euler Tour Tree

```

1 struct node_t {
2     // splay tree ...
3
4     node_t *walkdown(int d)
5     {
6         node_t *t = this;
7         while (t->ch[d] != null) t = t->ch[d];
8         return t;
9     }
10
11     node_t *adj(int d) // 0 -- prev, 1 -- succ
12     {

```



```

13         if (ch[d] != null) return ch[d]->walkdown(d^1);
14         node_t *t = this;
15         while (t->dir() == d) t = t->fa;
16         return t->fa;
17     }
18 }node[MAXN * 2]; // each node split into 2 nodes. i --> (i<<1) and (i<<1)^1
19
20 void cut(int t)
21 {
22     node_t *x = node[t<<1].adj(0), *y = node[t<<1^1].adj(1);
23     x->splay(), y->splay(x);
24     y->ch[0]->fa = null, y->setc(null, 0);
25     y->update(), x->update();
26 }
27
28 void link(int t, int p) // link subtree t to p
29 {
30     node_t *x = &node[p<<1], *y = &node[t<<1];
31     x->splay(), x->adj(1)->splay(x), y->splay();
32     x->ch[1]->setc(y, 0);
33     x->ch[1]->update(), x->update();
34 }

```

### 2.4.3 Top Tree

```

1 struct info_t {
2     int min, max, sum, size;
3
4     info_t(int min = inf, int max = -inf, int sum = 0, int size = 0)
5         : min(min), max(max), sum(sum), size(size) {}
6
7     info_t& operator+=(const info_t& a)
8     {
9         if(a.size == 0) return *this;
10        if(size == 0) return *this = a;
11        min = std::min(min, a.min);
12        max = std::max(max, a.max);
13        sum = sum + a.sum;
14        size = size + a.size;
15        return *this;
16    }
17 };
18
19 struct flag_t {
20     int mul, add;
21
22     flag_t(int mul = 1, int add = 0)
23         : mul(mul), add(add) {}
24
25     flag_t operator+=(const flag_t& a)
26     {
27         mul = mul * a.mul;
28         add = add * a.mul + a.add;
29         return *this;
30    }
31
32     bool empty() const
33     {
34         return mul == 1 && add == 0;
35    }
36 };

```

```

37
38 info_t operator+(const info_t& a, const flag_t& b)
39 {
40     if(!a.size) return a;
41
42     info_t z;
43     z.max = a.max * b.mul + b.add;
44     z.min = a.min * b.mul + b.add;
45     z.sum = a.sum * b.mul + b.add * a.size;
46     z.size = a.size;
47     return z;
48 }
49
50 struct node_t;
51 node_t* newnode();
52 void delnode(node_t *n);
53 struct node_t
54 {
55     int val;
56     node_t *fa, *ch[4];
57     info_t chain, tree, all;
58     flag_t chain_flag, tree_flag;
59     bool is_rev, is_inner;
60
61     bool is_root(int z) const
62     {
63         if(z == 0)
64         {
65             if(!fa) return true;
66             return fa->ch[0] != this && fa->ch[1] != this;
67         } else {
68             return !fa || !fa->is_inner || !is_inner;
69         }
70     }
71
72     void setc(node_t *c, int z)
73     {
74         ch[z] = c;
75         if(c) c->fa = this;
76     }
77
78     int where() const
79     {
80         for(int i = 0; i != 4; ++i)
81             if(fa->ch[i] == this) return i;
82         return -1; // throw 0;
83     }
84
85     int where(int z) const
86     {
87         return fa->ch[z] != this;
88     }
89
90     node_t* son(int i)
91     {
92         if(ch[i]) ch[i]->pushdown();
93         return ch[i];
94     }
95
96     void pushup()
97     {

```

```

98         chain = tree = all = info_t();
99         if(!is_inner) chain = all = info_t(val, val, val, 1);
100         for(int i = 0; i != 4; ++i) if(ch[i]) all += ch[i]->all;
101         for(int i = 2; i != 4; ++i) if(ch[i]) tree += ch[i]->all;
102         for(int i = 0; i != 2; ++i) {
103             if(ch[i]) {
104                 chain += ch[i]->chain;
105                 tree += ch[i]->tree;
106             }
107         }
108     }
109
110     void rotate(int z)
111     {
112         node_t *p = fa;
113         int d = where(z);
114         if(!p->fa) fa = 0;
115         else p->fa->setc(this, p->where());
116         p->setc(ch[z + !d], z + d);
117         setc(p, z + !d);
118         p->pushup();
119     }
120
121     void push_chain(const flag_t& r)
122     {
123         chain = chain + r, chain_flag += r;
124         all = chain, all += tree;
125         val = val * r.mul + r.add;
126     }
127
128     void push_tree(const flag_t& r, int virt)
129     {
130         all = all + r;
131         tree = tree + r;
132         tree_flag += r;
133         if(virt) push_chain(r);
134     }
135
136     void make_rev()
137     {
138         is_rev ^= 1;
139         std::swap(ch[0], ch[1]);
140     }
141
142     void pushdown()
143     {
144         if(is_rev) {
145             is_rev = false;
146             std::swap(ch[0], ch[1]);
147             if(ch[0]) ch[0]->is_rev ^= 1;
148             if(ch[1]) ch[1]->is_rev ^= 1;
149         }
150
151         if(!tree_flag.empty()) {
152             for(int i = 0; i != 4; ++i)
153                 if(ch[i]) ch[i]->push_tree(tree_flag, i >> 1);
154             tree_flag = flag_t();
155         }
156
157         if(!chain_flag.empty()) {
158             for(int i = 0; i != 2; ++i)

```

```

159             if(ch[i]) ch[i]->push_chain(chain_flag);
160             chain_flag = flag_t();
161         }
162     }
163
164     void splay(int z)
165     {
166         for(; !is_root(z); rotate(z)) {
167             if(!fa->is_root(z)) {
168                 if(where(z) == fa->where(z))
169                     fa->rotate(z);
170                 else rotate(z);
171             }
172         }
173
174         pushup();
175     }
176
177     void add(node_t *w)
178     {
179         w->pushdown();
180         for(int i = 2; i != 4; ++i)
181         {
182             if(!w->ch[i])
183             {
184                 w->setc(this, i);
185                 return;
186             }
187         }
188
189         node_t *x = newnode(), *n = w;
190         for(; n->ch[2]->is_inner; n = n->son(2));
191         x->setc(n->ch[2], 2);
192         x->setc(this, 3);
193         n->setc(x, 2);
194         x->splay(2);
195     }
196
197     void del()
198     {
199         if(fa->is_inner) {
200             fa->fa->setc(fa->ch[5 - where()], fa->where());
201             fa->fa->splay(2);
202             delnode(fa);
203         } else fa->setc(0, where());
204         fa = 0;
205     }
206
207     void access()
208     {
209         static node_t *stack[MAXN];
210         int st = 0;
211         for(node_t *u = this; u; u = u->fa) stack[st++] = u;
212         while(st) stack[--st]->pushdown();
213         node_t *prev = 0, *now = this;
214         while(now) {
215             now->splay(0);
216             if(now->ch[1]) now->ch[1]->add(now);
217             if(prev) prev->del();
218             now->setc(prev, 1);
219             prev = now;

```

```

220         now->pushup();
221         for(now = now->fa; now && now->is_inner; now = now->fa);
222     }
223
224     splay(0);
225 }
226
227 void make_root()
228 {
229     access();
230     is_rev ^= 1;
231 }
232
233 node_t *find_root()
234 {
235     node_t *z = this;
236     for(; z->fa; z = z->fa);
237     return z;
238 }
239
240 node_t *find_parent()
241 {
242     access();
243     node_t *z = son(0);
244     while(z && z->ch[1]) z = z->son(1);
245     return z;
246 }
247 };
248
249 node_t* cut(node_t *u)
250 {
251     node_t *v = u->find_parent();
252     if(v) v->access(), u->del(), v->pushup();
253     return v;
254 }
255
256 void link(node_t *u, node_t *v)
257 {
258     node_t *p = cut(u);
259     if(u->find_root() != v->find_root()) p = v;
260     if(p) p->access(), u->add(p), p->pushup();
261 }
262
263 int n, m;
264 int u[MAXN], v[MAXN];
265 int root;
266 int node_used;
267 node_t node[MAXN];
268 stack<node_t*> garbages;
269
270 void delnode(node_t *n)
271 {
272     garbages.push(n);
273 }
274
275 node_t* newnode()
276 {
277     node_t* z;
278     if(!garbages.empty()) {
279         z = garbages.top();
280         garbages.pop();

```

```

281     } else z = node + node_used++;
282     z->chain = z->all = info_t();
283     z->chain_flag = z->tree_flag = flag_t();
284     z->val = z->is_rev = 0;
285     z->is_inner = 1; z->fa = 0;
286     for(int i = 0; i != 4; ++i) z->ch[i] = 0;
287     return z;
288 }
289
290 void init()
291 {
292     scanf("%d%d", &n, &m);
293     node_used = n + 1;
294     for(int i = 1; i < n; ++i) scanf("%d%d", &u[i], &v[i]);
295     for(int i = 1; i <= n; ++i) {
296         scanf("%d", &node[i].val);
297         node[i].pushup();
298     }
299
300     for(int i = 1; i != n; ++i) {
301         node[v[i]].make_root();
302         link(&node[v[i]], &node[u[i]]);
303     }
304
305     scanf("%d", &root);
306     node[root].make_root();
307 }
308
309 void query()
310 {
311     // 0 -- subtree modify
312     // 1 -- change root
313     // 2 -- chain modify
314     // 3 -- subtree query minimum
315     // 4 -- subtree query maximum
316     // 5 -- subtree add
317     // 6 -- chain add
318     // 7 -- chain query minimum
319     // 8 -- chain query maximum
320     // 9 -- change x's father to y
321     // 10 -- chain query sum
322     // 11 -- subtree query sum
323     int k, x;
324     scanf("%d%d", &k, &x);
325     node_t *u = node + x;
326     if(k == 0 || k == 3 || k == 4 || k == 5 || k == 11) {
327         u->access();
328         if(k == 3 || k == 4 || k == 11) {
329             int ans = u->val;
330             for(int i = 2; i != 4; ++i) {
331                 if(u->ch[i]) {
332                     info_t info = u->ch[i]->all;
333                     if(k == 3) ans = min(ans, info.min);
334                     else if(k == 4) ans = max(ans, info.max);
335                     else if(k == 11) ans += info.sum;
336                 }
337             }
338
339             printf("%d\n", ans);
340         } else {
341             int y;

```

```

342         scanf("%d", &y);
343         flag_t flag(k == 5, y);
344         u->val = u->val * flag.mul + flag.add;
345         for(int i = 2; i != 4; ++i)
346             if(u->ch[i]) u->ch[i]->push_tree(flag, 1);
347         u->pushup();
348     }
349     } else if(k == 9) {
350         int v;
351         scanf("%d", &v);
352         link(u, node + v);
353     } else if(k == 1) {
354         u->make_root();
355         root = x;
356     } else {
357         int y;
358         scanf("%d", &y);
359         u->make_root();
360         node[y].access();
361         u->splay(0);
362         if(k == 7 || k == 8 || k == 10)
363         {
364             info_t ans = u->chain;
365             if(k == 7) printf("%d\n", ans.min);
366             else if(k == 8) printf("%d\n", ans.max);
367             else printf("%d\n", ans.sum);
368         } else {
369             int v;
370             scanf("%d", &v);
371             u->push_chain(flag_t(k == 6, v));
372         }
373
374         node[root].make_root();
375     }
376 }

```

## 2.5 KD Tree

```

1  const int K = 2;
2  struct kd {
3      double x[K];
4      int id;
5  }t[MAXN];
6
7  double dis(const kd &a, const kd &b)
8  {
9      double s = 0;
10     for (int i = 0; i < K; ++i) s += sqr(a.x[i] - b.x[i]);
11     return sqrt(s);
12 }
13
14 struct cmpk {
15     int k;
16     cmpk(int k): k(k) {}
17     bool operator()(const kd &a, const kd &b)
18     { return a.x[k] < b.x[k]; }
19 };
20
21 void build(int l, int r, int d)
22 {

```

```

23     if (r - l <= 1) return;
24     int mid = (l + r) >> 1;
25     nth_element(t + l, t + mid, t + r, cmpk(d));
26     if (++d == K) d = 0;
27     build(l, mid, d); build(mid + 1, r, d);
28 }
29
30 typedef priority_queue<pair<double, int> > heap;
31 void knn(int l, int r, int d, const kd &p, size_t k, heap &h)
32 {
33     if (r - l < 1) return;
34     int mid = (l + r) >> 1;
35     h.push(make_pair(dis(p, t[mid]), t[mid].id));
36     if (h.size() > k) h.pop();
37     double dx = p.x[d] - t[mid].x[d];
38     if (++d == K) d = 0;
39     if (dx < 0) {
40         knn(l, mid, d, p, k, h);
41         if (h.top().first > dx) knn(mid + 1, r, d, p, k, h);
42     } else {
43         knn(mid + 1, r, d, p, k, h);
44         if (h.top().first > dx) knn(l, mid, d, p, k, h);
45     }
46 }
47
48 /*
49 Usage:
50     build(0, n, 0);
51     knn(0, n, 0, pos, ans_heap);
52 */

```

## 2.6 Sparse Table

```

1  int rmq[MAXN][LOGN];
2
3  void initRMQ(int a[], int n)
4  {
5      for (int i = 0; i < n; ++i) rmq[i][0] = a[i];
6      for (int j = 1; (1<<j) <= n; ++j) {
7          for (int i = 0; i + (1<<j) <= n; ++i) {
8              rmq[i][j] = min(rmq[i][j-1], rmq[i+(1<<(j-1))][j-1]);
9          }
10     }
11 }
12
13 int RMQ(int l, int r)
14 {
15     int k = sizeof(int) * 8 - __builtin_clz(r - l + 1) - 1;
16     return min(rmq[l][k], rmq[r-(1<<k)+1][k]);
17 }

```



## Chapter 3

# Stringology

### 3.1 KMP Algorithm

```
1 void getf(char *s, int *f)
2 {
3     int n = strlen(s);
4     f[0] = 0; f[1] = 0;
5     for (int i = 1; i < n; ++i) {
6         int j = f[i];
7         while (j && s[i] != s[j]) j = f[j];
8         f[i + 1] = s[i] == s[j] ? j + 1 : 0;
9     }
10 }
11
12 int match(char *s, char *p, int *f)
13 {
14     int n = strlen(s), m = strlen(p);
15     int j = 0;
16     for (int i = 0; i < n; ++i) {
17         while (j && s[i] != p[j]) j = f[j];
18         if (s[i] == p[j]) ++j;
19         if (j == m) return i - m + 1;
20     }
21 }
```

### 3.2 Extend-KMP Algorithm

```
1 void getf(char *s, int *f)
2 {
3     int n = strlen(s), j = 0, k = 1;
4     while (j + 1 < n && s[j] == s[j + 1]) ++j;
5     f[0] = n; f[1] = j;
6     for (int i = 2; i < n; ++i) {
7         int len = k + f[k] - 1, t = f[i - k];
8         if (i + t <= len) {
9             f[i] = t;
10        } else {
11            j = max(0, len - i + 1);
12            while (i + j < n && s[i + j] == s[j]) ++j;
13            f[i] = j; k = i;
14        }
15    }
16 }
17
18 void match(char *s, char *p, int *f, int *ex)
19 {
```

```

20     int n = strlen(s), j = 0, k = 0;
21     while (j < n && s[j] == p[j]) ++j;
22     ex[0] = j;
23     for (int i = 1; i < n; ++i) {
24         int len = k + ex[k] - 1, t = f[i - k];
25         if (i + t <= len) {
26             ex[i] = t;
27         } else {
28             j = max(0, len - i + 1);
29             while (i + j < n && s[i + j] == p[j]) ++j;
30             ex[i] = j; k = i;
31         }
32     }
33 }

```

### 3.3 Aho-Corasick Automation

```

1  const int PSZ = MAXN * LEN;
2  struct trie {
3      trie *ch[SIGMA], *f;
4      // trie *last;
5      int val;
6  }pool[PSZ], *dict;
7  int psz;
8  int head, tail;
9  trie *que[PSZ];
10
11 void insert(trie *t, const char *s)
12 {
13     for (; *s; ++s) {
14         int c = *s - 'a';
15         if (!t->ch[c]) memset(t->ch[c] = pool + psz++, 0, sizeof(trie));
16         t = t->ch[c];
17     }
18     ++t->val;
19 }
20
21 void build_fail(trie *t)
22 {
23     head = tail = 0;
24     for (int i = 0; i < SIGMA; ++i) {
25         if (t->ch[i]) (que[tail++] = t->ch[i])->f = t;
26         else t->ch[i] = t;
27     }
28     while (head < tail) {
29         t = que[head++];
30         // t->val += t->f->val; # method 1
31         // t->last = t->f->val ? t->f : t->f->last; # method 2
32         for (int i = 0; i < SIGMA; ++i) {
33             if (t->ch[i]) (que[tail++] = t->ch[i])->f = t->f->ch[i];
34             else t->ch[i] = t->f->ch[i];
35         }
36     }
37 }
38
39 int find(trie *t, const char *s)
40 {
41     int sum = 0;
42     for (; *s; ++s) {
43         int c = *s - 'a';

```

```

44         t = t->ch[c];
45         // sum += i->val; # method 1
46         // for (trie *i = t; i && i->val != -1; i = i->last) {
47         //     sum += i->val, i->val = -1; // mark as visited
48         // }
49     }
50     return sum;
51 }
52
53 /*
54 Initialize:
55     psz = 1; memset(dict = pool, 0, sizeof(trie));
56 Method 1: counting matching times
57 Method 2: counting the number of pattern matched
58 ** duplicate patterns counted only once in method 2
59 */

```

### 3.4 Suffix Array

```

1  int n;
2  char s[MAXN];
3  int sa[MAXN], rank[MAXN], height[MAXN];
4  int c[MAXN], wx[MAXN], wy[MAXN];
5
6  void build_sa(int m)
7  {
8      int *x = wx, *y = wy;
9      for (int i = 0; i < m; ++i) c[i] = 0;
10     for (int i = 0; i < n; ++i) ++c[x[i] = s[i]];
11     for (int i = 1; i < m; ++i) c[i] += c[i - 1];
12     for (int i = n - 1; i >= 0; --i) sa[--c[x[i]]] = i;
13     for (int k = 1; k <= n; k <= 1) {
14         int p = 0;
15         for (int i = n - k; i < n; ++i) y[p++] = i;
16         for (int i = 0; i < n; ++i) if (sa[i] >= k) y[p++] = sa[i] - k;
17         for (int i = 0; i < m; ++i) c[i] = 0;
18         for (int i = 0; i < n; ++i) ++c[x[y[i]]];
19         for (int i = 1; i < m; ++i) c[i] += c[i - 1];
20         for (int i = n - 1; i >= 0; --i) sa[--c[x[y[i]]]] = y[i];
21         swap(x, y);
22         p = 1; x[sa[0]] = 0;
23         for (int i = 1; i < n; ++i) {
24             x[sa[i]] = y[sa[i - 1]] == y[sa[i]] &&
25                 y[sa[i - 1] + k] == y[sa[i] + k] ?
26                 p - 1 : p++;
27         }
28         if (p == n) break;
29         m = p;
30     }
31 }
32
33 void build_height()
34 {
35     for (int i = 0; i < n; ++i) rank[sa[i]] = i;
36     for (int i = 0, k = 0; i < n; ++i) {
37         if (k) --k;
38         if (!rank[i]) continue;
39         int j = sa[rank[i] - 1];
40         while (s[i + k] == s[j + k]) ++k;
41         height[rank[i]] = k;

```

```

42     }
43 }
44
45 // height[i] == lcp(suffix(sa[i-1]), suffix(sa[i]))
46 // REMEBER: add '$' after the string

```

### 3.5 Suffix Automation

```

1 struct sam {
2     int l;
3     sam *f, *ch[SIGMA];
4 }pool[LEN * 2], *root, *tail;
5 int psz;
6
7 sam *init_node(sam *p)
8 {
9     memset(p->ch, 0, sizeof(p->ch));
10    p->f = 0; p->l = 0;
11    return p;
12 }
13
14 void sam_add(int v)
15 {
16     sam *p = init_node(pool + psz++), *i;
17     p->l = tail->l + 1;
18     for (i = tail; i && !i->ch[v]; i = i->f) i->ch[v] = p;
19     if (!i) {
20         p->f = root;
21     } else if (i->ch[v]->l == i->l + 1) {
22         p->f = i->ch[v];
23     } else {
24         sam *q = pool + psz++, *r = i->ch[v];
25         *q = *r;
26         q->l = i->l + 1;
27         p->f = r->f = q;
28         for (; i && i->ch[v] == r; i = i->f) i->ch[v] = q;
29     }
30     tail = p;
31 }
32
33 int match(sam *root, char *s)
34 {
35     int k = 0, ret = 0;
36     sam *p = root;
37     for (; *s; ++s) {
38         int c = *s - 'a';
39         if (p->ch[c]) {
40             ++k, p = p->ch[c];
41         } else {
42             while (p && !p->ch[c]) p = p->f;
43             if (p) k = p->l + 1, p = p->ch[c];
44             else p = root; k = 0;
45         }
46         ret = max(ret, k);
47         // p->match = max(p->match, k);
48     }
49     return ret;
50 }
51
52 // Initalize: init_node(root = tail = pool); psz = 1;

```

### 3.6 Longest Palindrome Substring(Manacher)

```
1 char s[MAXN], t[MAXN + MAXN + 3];
2 int rad[MAXN + MAXN + 3];
3
4 void manacher(char *s)
5 {
6     int n = strlen(s), len = 0;
7     t[len++] = '^'; t[len++] = '#';
8     for (int i = 0; i < n; ++i) {
9         t[len++] = s[i];
10        t[len++] = '#';
11    }
12    t[len] = 0;
13    int i = 1, j = 1, k;
14    while (i < len) {
15        while (t[i - j] == t[i + j]) ++j;
16        rad[i] = j;
17        for (k = 1; k < j && rad[i - k] != rad[i] - k; ++k) {
18            rad[i + k] = min(rad[i - k], rad[i] - k);
19        }
20        i += k; j = max(j - k, 1);
21    }
22 }
23
24 /*
25 s: abaaba
26 t:  ^ # a # b # a # a # b # a # \0
27 rad: 0 1 2 1 4 1 2 7 2 1 4 1 2 1
28 */
```

### 3.7 Palindromic Tree

```
1 char s[MAXN];
2 struct node {
3     int len;
4     node *ch[2], *fail;
5 }pool[MAXN], *root, *last;
6 int psz;
7
8 node *newnode(int len)
9 {
10    node *t = pool + psz++;
11    t->len = len;
12    t->ch[0] = t->ch[1] = t->fail = 0;
13    return t;
14 }
15
16 node *find(node *t, int i)
17 {
18    while (i <= t->len || s[i - t->len - 1] != s[i]) t = t->fail;
19    return t;
20 }
21
22 bool add(int i) // return whether add a new palindrome or not
23 {
24    int c = s[i] - 'a';
25    node *t = find(last, i);
26    if (t->ch[c]) { last = t->ch[c]; return false; }
27    last = t->ch[c] = newnode(t->len + 2);
```

```

28 |         last->fail = t->fail ? find(t->fail, i)->ch[c] : root;
29 |         return true;
30 |     }
31 |
32 | void init()
33 | {
34 |     n = strlen(s);
35 |     psz = 0;
36 |     node *t0 = newnode(-1), *t1 = newnode(0);
37 |     t1->fail = t0;
38 |     root = last = t1;
39 |     for (int i = 0; i < n; ++i) add(i);
40 | }

```

### 3.8 Minimum Representation

```

1 | int minrep(char *s, int n)
2 | {
3 |     int i = 0, j = 1, k = 0, t;
4 |     while (i < n && j < n && k < n) {
5 |         t = s[(i + k) % n] - s[(j + k) % n];
6 |         if (!t) { ++k; continue; }
7 |         if (t > 0) i = i + k + 1;
8 |         else j = j + k + 1;
9 |         if (i == j) ++j;
10 |        k = 0;
11 |    }
12 |    return min(i, j);
13 | }

```

## Chapter 4

# Computational Geometry

### 4.1 Basic Operations

```
1 typedef complex<double> point;
2 typedef point vec;
3 #define X real()
4 #define Y imag()
5
6 const double eps = 1e-8;
7
8 int dcmp(double x) { return x < -eps ? -1 : x > eps; }
9 bool zero(vec v) { return !dcmp(v.X) && !dcmp(v.Y); }
10
11 double sqr(double x) { return x * x; }
12 double dis(point a, point b) { return abs(a - b); }
13
14 double cross(vec a, vec b) { return a.X * b.Y - a.Y * b.X; }
15 double cross(point a, point b, point c) { return cross(b - a, c - a); }
16 double dot(vec a, vec b) { return a.X * b.X + a.Y * b.Y; }
17 double dot(point a, point b, point c) { return dot(b - a, c - a); }
18
19 vec dir(line ln) { return ln.t - ln.s; }
20 vec normal(vec v) { return vec(-v.Y, v.X); }
21 vec unit(vec v) { return v / abs(v); }
22
23 vec proj(vec v, vec n) { return n * dot(v, n) / norm(n); }
24 point proj(point p, line ln) { return ln.s + proj(p - ln.s, dir(ln)); }
25 vec reflect(vec v, vec n) { return proj(v, n) * 2. - v; }
26 point reflect(point p, line ln) { return ln.s + reflect(p - ln.s, dir(ln)); }
27
28 vec rotate(vec v, double a) { return v * polar(1., a); }
29 double angle(vec a, vec b) { return arg(b / a); }
```

#### 4.1.1 Line

```
1 double dis(point p, line ln) { return fabs(cross(p, ln.s, ln.t)) / len(ln); }
2
3 bool onseg(point p, line ln)
4 { return dcmp(cross(p, ln.s, ln.t)) == 0 && dcmp(dot(p, ln.s, ln.t)) <= 0; }
5
6 double dtoseg(point p, line ln)
7 {
8     if (dcmp(dot(ln.s, ln.t, p)) <= 0) return dis(p, ln.s);
9     if (dcmp(dot(ln.t, ln.s, p)) <= 0) return dis(p, ln.t);
10    return dis(p, ln);
11 }
```

```

12
13 bool inter(line a, line b, point &p)
14 {
15     double s1 = cross(a.s, a.t, b.s);
16     double s2 = cross(a.s, a.t, b.t);
17     if (!dcmp(s1 - s2)) return false;
18     p = (s1 * b.t - s2 * b.s) / (s1 - s2);
19     return true;
20 }
21
22 bool seginter(line a, line b, point &p) // segment intersection(strict)
23 {
24     double s1 = cross(a.s, a.t, b.s);
25     double s2 = cross(a.s, a.t, b.t);
26     if ((dcmp(s1) ^ dcmp(s2)) != -2) return false;
27     double s3 = cross(b.s, b.t, a.s);
28     double s4 = cross(b.s, b.t, a.t);
29     if ((dcmp(s3) ^ dcmp(s4)) != -2) return false;
30     p = (s1 * b.t - s2 * b.s) / (s1 - s2);
31     return true;
32 }

```

#### 4.1.2 Triangle

```

1 double area(double a, double b, double c) // Heron's Formula
2 {
3     double p = (a + b + c) * 0.5;
4     return sqrt(p * (p - a) * (p - b) * (p - c));
5 }
6
7 double angle(double a, double b, double c) // Law of Cosines
8 {
9     return acos((sqr(a) + sqr(b) - sqr(c)) / (2 * a * b));
10 }
11
12 point center(point A, point B, point C) // Circumcenter
13 {
14     double d1 = dot(A, B, C), d2 = dot(B, C, A), d3 = dot(C, A, B);
15     double c1 = d2 * d3, c2 = d1 * d3, c3 = d1 * d2, c = c1 + c2 + c3;
16     if (!dcmp(c)) return A; // coincident
17     return ((c2 + c3) * A + (c1 + c3) * B + (c1 + c2) * C) / (2 * c);
18 }
19
20 point incenter(point A, point B, point C)
21 {
22     double a = abs(B - C), b = abs(C - A), c = abs(A - B);
23     if (!dcmp(a + b + c)) return A; // coincident
24     return (a * A + b * B + c * C) / (a + b + c);
25 }
26
27 point centroid(point A, point B, point C)
28 {
29     return (A + B + C) / 3;
30 }
31
32 point orthocenter(point A, point B, point C)
33 {
34     double d1 = dot(A, B, C), d2 = dot(B, C, A), d3 = dot(C, A, B);
35     double c1 = d2 * d3, c2 = d1 * d3, c3 = d1 * d2, c = c1 + c2 + c3;
36     if (!dcmp(c)) return A; // coincident
37     return (c1 * A + c2 * B + c3 * C) / c;

```



```

38 }
39
40 point feramat(point A, point B, point C)
41 {
42     double a = abs(B - C), b = abs(C - A), c = abs(A - B);
43     if (dot(A, B, C) / b / c < -.5) return A;
44     if (dot(B, C, A) / c / a < -.5) return B;
45     if (dot(C, A, B) / a / b < -.5) return C;
46     if (cross(A, B, C) < 0) swap(B, C);
47     point CC = (B - A) * polar(1., -pi / 3) + A;
48     point BB = (C - A) * polar(1., pi / 3) + A;
49     return inter(line(B, BB), line(C, CC));
50 }

```

### 4.1.3 Circle

```

1 double adjust(double a)
2 {
3     while (a < -pi) a += 2 * pi;
4     while (a > pi) a -= 2 * pi;
5     return a;
6 }
7
8 bool inter(circle c, line ln, point &p1, point &p2)
9 {
10     point p = proj(c.c, ln);
11     double d = dis(p, c.c);
12     if (dcmp(d - c.r) > 0) return false;
13     vec v = sqrt(c.r * c.r - d * d) * unit(dir(ln));
14     p1 = p - v; p2 = p + v;
15     return true;
16 }
17
18 bool inter(circle c, line ln, double &a1, double &a2)
19 {
20     point p = proj(c.c, ln);
21     double d = dis(p, c.c);
22     if (dcmp(d - c.r) > 0) return false;
23     double alpha = arg(p - c.c), beta = acos(d / c.r);
24     a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
25     return true;
26 }
27
28 bool inter(circle a, circle b, point &p1, point &p2)
29 {
30     double d = dis(a.c, b.c);
31     if (dcmp(d - (a.r + b.r)) > 0) return false; // disjoint
32     if (!dcmp(d) || dcmp(d - fabs(a.r - b.r)) < 0) return false; // include
33     double d1 = (sqr(d) + sqr(a.r) - sqr(b.r)) / (2 * d), d2 = d - d1;
34     point p = (d1 * b.c + d2 * a.c) / d;
35     vec v = sqrt(sqr(a.r) - sqr(d1)) * unit(normal(b.c - a.c));
36     p1 = p - v; p2 = p + v;
37     return true;
38 }
39
40 bool inter(circle a, circle b, double &a1, double &a2)
41 {
42     double d = dis(a.c, b.c);
43     if (dcmp(d - (a.r + b.r)) > 0) return false; // disjoint
44     if (!dcmp(d) || dcmp(d - fabs(a.r - b.r)) < 0) return false; // include
45     double alpha = arg(b.c - a.c), beta = angle(a.r, d, b.r);

```

```

46         a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
47         return true;
48     }
49
50 bool tan(circle c, point p, point &p1, point &p2)
51 {
52     double d = dis(p, c.c);
53     if (dcmp(d - c.r) < 0) return false;
54     double d1 = c.r * c.r / d, d2 = d - d1;
55     point p0 = (d1 * p + d2 * c.c) / d;
56     vec v = sqrt(sqr(c.r) - sqr(d1)) * unit(normal(p - c.c));
57     p1 = p0 - v; p2 = p0 + v;
58     return true;
59 }
60
61 bool tan(circle c, point p, double &a1, double &a2)
62 {
63     double d = dis(p, c.c);
64     if (dcmp(d - c.r) < 0) return false;
65     double alpha = arg(p - c.c), beta = acos(c.r / d);
66     a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
67     return true;
68 }
69
70 bool outertan(circle a, circle b, double &a1, double &a2)
71 {
72     double d = dis(a.c, b.c);
73     if (!dcmp(d) || dcmp(d - fabs(a.r - b.r)) < 0) return false; // include
74     double alpha = arg(b.c - a.c), beta = acos((a.r - b.r) / d);
75     a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
76     return true;
77 }
78
79 bool innertan(circle a, circle b, double &a1, double &a2)
80 {
81     double d = dis(a.c, b.c);
82     if (!dcmp(d) || dcmp(d - (a.r + b.r)) < 0) return false; // disjoint
83     double alpha = arg(b.c - a.c), beta = acos((a.r + b.r) / d);
84     a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
85     return true;
86 }

```

## 4.2 Point in Polygon Problem

```

1 bool inpoly(point a, point *p, int n)
2 {
3     int wn = 0;
4     for (int i = 0; i < n; ++i) {
5         point p1 = p[i], p2 = p[(i + 1) % n];
6         int d = dcmp(cross(a, p1, p2));
7         if (!s && dot(a, p1, p2) <= 0) return true;
8         int d1 = dcmp(p1.Y - a.Y);
9         int d2 = dcmp(p2.Y - a.Y);
10        if (d > 0 && d1 <= 0 && d2 > 0) ++wn;
11        if (d < 0 && d2 <= 0 && d1 > 0) --wn;
12    }
13    return wn != 0;
14 }

```

### 4.3 Convex Hull(Graham)

```
1 bool cmpx(point a, point b) { return dcmp(a.X - b.X) ? a.X < b.X : a.Y < b.Y; }
2
3 int graham(point p[], int n, point h[])
4 {
5     int m = 0;
6     sort(p, p + n, cmpx);
7     for (int i = 0; i < n; ++i) {
8         while (m > 1 && dcmp(cross(h[m - 2], h[m - 1], p[i])) <= 0) --m;
9         h[m++] = p[i];
10    }
11    int k = m;
12    for (int i = n - 2; i >= 0; --i) {
13        while (m > k && dcmp(cross(h[m - 2], h[m - 1], p[i])) <= 0) --m;
14        h[m++] = p[i];
15    }
16    if (n > 1) --m;
17    return m;
18 }
```

### 4.4 Dynamic Convex Hull

```
1 struct cmpx {
2     bool operator()(const point &a, point &b) const
3     { return dcmp(a.X - b.X) < 0; }
4 }
5
6 set<point, cmpx> lower, upper;
7
8 double insert(set<point, cmpx> &h, point p)
9 {
10     double s = 0;
11     set<point, cmpx>::iterator it = h.lower_bound(p);
12     if (it != h.end() && !dcmp(p.X - it->X)) {
13         if (dcmp(p.Y - it->Y) >= 0) return 0;
14         if (it != h.begin()) s += cross(p, *it, *prev(it));
15         if (next(it) != h.end()) s += cross(p, *next(it), *it);
16         h.erase(it);
17     } else if (it != h.begin() && it != h.end()) {
18         double ds = cross(p, *it, *prev(it));
19         if (dcmp(ds) <= 0) return 0;
20         s += ds;
21     }
22     it = h.insert(p).first;
23     while (it != h.begin() && prev(it) != h.begin()) {
24         double ds = cross(p, *prev(it), *prev(prev(it)));
25         if (dcmp(ds) < 0) break;
26         h.erase(prev(it));
27         s += ds;
28     }
29     while (next(it) != h.end() && next(next(it)) != h.end()) {
30         double ds = cross(p, *next(it), *next(next(it)));
31         if (dcmp(ds) > 0) break;
32         h.erase(next(it));
33         s -= ds;
34     }
35     return s * 0.5;
36 }
37
```

```

38 double insert(point p) // return area increment
39 {
40     double s = 0;
41     if (lower.size()) {
42         s += max(0., cross(p, *lower.begin(), conj(*upper.begin())));
43         s += max(0., cross(p, conj(*upper.rbegin()), *lower.rbegin()));
44     }
45     s += insert(lower, p);
46     s += insert(upper, conj(p));
47     return s;
48 }

```

## 4.5 Half-plane Intersection

```

1 bool inhp(point p, line hp) { return dcmp(cross(hp.s, hp.t, p)) >= 0; }
2
3 bool cmpang(line a, line b)
4 { return dcmp(a.a - b.a) ? a.a < b.a : cross(a.s, a.t, b.s) < 0; }
5
6 int hpinter(line q[], int n, point h[])
7 {
8     // line q[i] represent the half-plane on its left
9     int head = 0, tail = 0, m = 0;
10    for (int i = 0; i < n; ++i) q[i].a = arg(dir(q[i]));
11    sort(q, q + n, cmpang);
12    for (int i = 1; i < n; ++i) {
13        if (!dcmp(q[i].a - q[i - 1].a)) continue;
14        while (head < tail && !inhp(h[tail - 1], q[i])) --tail;
15        while (head < tail && !inhp(h[head], q[i])) ++head;
16        q[++tail] = q[i];
17        if (head < tail) h[tail - 1] = inter(q[tail - 1], q[tail]);
18    }
19    while (head < tail && !inhp(h[tail - 1], q[head])) --tail;
20    if (head < tail) h[tail] = inter(q[tail], q[head]);
21    for (int i = head; i <= tail; ++i) h[m++] = h[i];
22    return m;
23 }
24
25 line makehp(double a, double b, double c) // ax + by + c > 0
26 {
27     point p1 = fabs(a) > fabs(b) ? point(-c / a, 0) : point(0, -c / b);
28     point p2 = p1 + vec(b, -a);
29     return line(p1, p2);
30 }

```

## 4.6 Closest Pair(Divide and Conquer)

```

1 bool cmpx(point a, point b) { return a.X < b.X; }
2 bool cmpy(point a, point b) { return a.Y < b.Y; }
3
4 double mindis(point p[], int l, int r)
5 {
6     static point t[MAXN];
7     if (r - l <= 1) return inf;
8     int mid = (l + r) >> 1, m = 0;
9     double x = p[mid].X;
10    double d = min(mindis(l, mid), mindis(mid, r));
11    inplace_merge(p + l, p + mid, p + r, cmpy());
12    for (int i = l; i < r; ++i) {
13        if (fabs(x - p[i].X) < d) t[m++] = p[i];

```

```

14     }
15     for (int i = 0; i < m; ++i) {
16         for (int j = i + 1; j < m; ++j) {
17             if (t[j].Y - t[i].Y >= d) break;
18             d = min(d, abs(t[i] - t[j]));
19         }
20     }
21     return d;
22 }
23
24 /*
25 Initialize: sort(p, p + n, cmpx)
26 Usage: mindis(0, n)
27 */

```

## 4.7 Farthest Pair(Rotating Caliper)

```

1 double maxdis(point *p, int n)
2 {
3     int m = graham(p, n, h);
4     if (m == 2) return abs(h[0] - h[1]);
5     h[m] = h[0];
6     double d = 0;
7     for (int i = 0, j = 1; i < m; ++i) {
8         while (dcmp(cross(h[i + 1] - h[i], h[j + 1] - h[j])) > 0) {
9             j = (j + 1) % m;
10        }
11        d = max(d, abs(h[i] - h[j]));
12    }
13    return d;
14 }

```

## 4.8 Minimum Distance Between Convex Hull(Rotating Caliper)

```

1 void mindis(point *p1, int n, point *p2, int m)
2 {
3     int i = 0, j = 0;
4     for (int k = 1; k < n; ++k) if (cmpx()(p1[k], p1[i])) i = k;
5     for (int k = 1; k < m; ++k) if (cmpx()(p2[k], p2[j])) j = k;
6     for (int t = 0; t < n + m; ++t) {
7         if (dcmp(cross(p1[i + 1] - p1[i], p2[j + 1] - p2[j])) < 0) {
8             ans = min(ans, dtoseg(p2[j], line(p1[i], p1[i + 1])));
9             i = (i + 1) % n;
10        } else {
11            ans = min(ans, dtoseg(p1[i], line(p2[j], p2[j + 1])));
12            j = (j + 1) % m;
13        }
14    }
15 }

```

## 4.9 Union Area of a Circle and a Polygon

```

1 double area(circle c, point a, point b)
2 {
3     a -= c.c; b -= c.c;
4     if (zero(a) || zero(b)) return 0;
5     double s1 = .5 * arg(b / a) * sqr(c.r);
6     double s2 = .5 * cross(a, b);
7     return fabs(s1) < fabs(s2) ? s1 : s2;

```

```

8 }
9
10 double unionarea(circle c, point p[], int n)
11 {
12     double s = 0;
13     for (int i = 0; i < n; ++i) {
14         point A = p[i], B = p[(i + 1) % n], p1, p2;
15         line AB = line(A, B);
16         if (inter(c, AB, p1, p2) && (onseg(p1, AB) || onseg(p2, AB))) {
17             s += area(c, A, p1) + area(c, p1, p2) + area(c, p2, B);
18         } else {
19             s += area(c, A, B);
20         }
21     }
22     return fabs(s);
23 }

```

## 4.10 Union Area of Circles

```

1 bool same(circle a, circle b) { return zero(a.c - b.c) && !dcmp(a.r - b.r); }
2 bool incir(circle a, circle b) { return dcmp(dis(a.c, b.c) + a.r - b.r) <= 0; }
3
4 void unionarea(circle c[], int n, double tot[])
5 {
6     static pair<double, int> a[MAXN * 2];
7     for (int i = 0; i <= n; ++i) tot[i] = 0;
8     for (int i = 0; i < n; ++i) {
9         int m = 0, k = 0;
10        for (int j = 0; j < n; ++j) if (i != j) {
11            double a1, a2;
12            if (same(c[i], c[j]) && i < j) continue;
13            if (incir(c[i], c[j])) { ++k; continue; }
14            if (!inter(c[i], c[j], a1, a2)) continue;
15            a[m++] = make_pair(a1, 1);
16            a[m++] = make_pair(a2, -1);
17            if (a1 > a2) ++k;
18        }
19        sort(a, a + m);
20        double a1 = a[m - 1].first - 2 * pi, a2, rad;
21        for (int j = 0; j < m; ++j) {
22            a2 = a[j].first, rad = a2 - a1;
23            tot[k] += .5 * sqr(c[i].r) * (rad - sin(rad));
24            tot[k] += .5 * cross(c[i].p(a1), c[i].p(a2));
25            k += a[j].second;
26            a1 = a2;
27        }
28        if (!m) tot[k] += pi * sqr(c[i].r);
29    }
30 }
31
32 /*
33 tot[0]                = the aera of union
34 tot[n-1]              = the aera of intersection
35 tot[k-1] - tot[k]    = the aera covered k times
36 */

```

## 4.11 Union Area of Polygons

```

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

```

```

3
4 void unionarea(vector<point> p[], int n, double tot[])
5 {
6     for (int i = 0; i <= n; ++i) tot[i] = 0;
7     for (int i = 0; i < n; ++i)
8     for (int ii = 0; ii < p[i].size(); ++ii) {
9         point A = p[i][ii], B = p[i][(ii + 1) % p[i].size()];
10        line AB = line(A, B);
11        vector<pair<double, int>> c;
12        for (int j = 0; j < n; ++j) if (i != j)
13        for (int jj = 0; jj < p[j].size(); ++jj) {
14            point C = p[j][jj], D = p[j][(jj + 1) % p[j].size()];
15            line CD = line(C, D);
16            int f1 = dcmp(cross(A, B, C));
17            int f2 = dcmp(cross(A, B, D));
18            if (!f1 && !f2) {
19                if (i < j && dcmp(dot(dir(AB), dir(CD))) > 0) {
20                    c.push_back(make_pair(pos(C, AB), 1));
21                    c.push_back(make_pair(pos(D, AB), -1));
22                }
23                continue;
24            }
25            double s1 = cross(C, D, A);
26            double s2 = cross(C, D, B);
27            double t = s1 / (s1 - s2);
28            if (f1 >= 0 && f2 < 0) c.push_back(make_pair(t, 1));
29            if (f1 < 0 && f2 >= 0) c.push_back(make_pair(t, -1));
30        }
31        c.push_back(make_pair(0., 0));
32        c.push_back(make_pair(1., 0));
33        sort(c.begin(), c.end());
34        double s = .5 * cross(A, B), z = min(max(c[0].s, 0.), 1.);
35        for (int j = 1, k = c[0].second; j < c.size(); ++j) {
36            double w = min(max(c[j].first, 0.), 1.);
37            tot[k] += s * (w - z);
38            k += c[j].second;
39            z = w;
40        }
41    }
42 }
43
44 /*
45 tot[0]                = the aera of union
46 tot[n-1]              = the aera of intersection
47 tot[k-1] - tot[k]    = the aera covered by k times
48 */

```

## 4.12 Minimum Enclosing Circle(Randomized Incremental Method)

```

1 circle mincir(point *p, int n)
2 {
3     point c;
4     double r;
5     random_shuffle(p, p + n);
6     c = p[0]; r = 0;
7     for (int i = 1; i < n; ++i) {
8         if (dcmp(abs(p[i] - c) - r) <= 0) continue;
9         c = p[i]; r = 0;
10        for (int j = 0; j < i; ++j) {
11            if (dcmp(abs(p[j] - c) - r) <= 0) continue;

```

```

12         c = (p[i] + p[j]) * 0.5; r = dis(p[j], c);
13         for (int k = 0; k < j; ++k) {
14             if (dcmp(abs(p[k] - c) - r) <= 0) continue;
15             c = center(p[i], p[j], p[k]); r = dis(p[k], c);
16         }
17     }
18 }
19 return circle(c, r);
20 }

```

### 4.13 Planar Strainght-line Graph(PSLG)

```

1  int pcnt, ecnt, fcnt;
2  point p[MAXN];
3  struct edge { int t; double ang; };
4  edge E[MAXN * MAXN * 2];
5  vector<int> G[MAXN * MAXN];
6  int co[MAXN * MAXN * 2];
7  int pre[MAXN * MAXN * 2];
8  vector<point> face[MAXN * MAXN * 2];
9
10 void addedge(int u, int v)
11 {
12     G[u].push_back(ecnt);
13     E[ecnt++] = (edge){v, arg(p[v] - p[u])};
14     G[v].push_back(ecnt);
15     E[ecnt++] = (edge){u, arg(p[u] - p[v])};
16 }
17
18 bool cmpang(int i, int j) { return E[i].ang < E[j].ang; }
19
20 void build_pslg()
21 {
22     for (int u = 0; u < pcnt; ++u) {
23         sort(G[u].begin(), G[u].end(), cmpang);
24         int n = G[u].size();
25         for (int j = 0; j < n; ++j) pre[G[u][(j + 1) % n]] = G[u][j];
26     }
27     memset(co, -1, sizeof(co));
28     for (int u = 0; u < pcnt; ++u) {
29         for (int i = 0; i < G[u].size(); ++i) {
30             int e = G[u][i];
31             if (co[e] != -1) continue;
32             while (co[e] == -1) {
33                 co[e] = fcnt;
34                 face[fcnt].push_back(p[E[e].t]);
35                 e = pre[e^1];
36             }
37             face[++fcnt].clear();
38         }
39     }
40 }

```

### 4.14 3D Computational Geometry

```

1  bool zero(vec3 v)
2  { return !dcmp(v.x) && !dcmp(v.y) && !dcmp(v.z); }
3
4  double dot(vec3 a, vec3 b)
5  { return a.x * b.x + a.y * b.y + a.z * b.z; }

```



```

6
7 double abs(vec3 v)
8 { return sqrt(dot(v, v)); }
9
10 vec3 unit(vec3 v)
11 { return v / abs(v); }
12
13 vec3 cross(vec3 a, vec3 b)
14 {
15     return vec3(a.y * b.z - a.z * b.y,
16                 a.z * b.x - a.x * b.z,
17                 a.x * b.y - a.y * b.x);
18 }
19
20 double area2(point3 a, point3 b, point3 c)
21 { return abs(cross(b - a, c - a)); }
22
23 double vol6(point3 a, point3 b, point3 c, point3 d)
24 { return dot(cross(b - a, c - a), d - a); }
25
26 double len(line3 ln)
27 { return abs(ln.s - ln.t); }
28
29 vec3 dir(line3 ln)
30 { return ln.t - ln.s; }
31
32 vec3 proj(vec3 v, vec3 d)
33 { return d * dot(v, d) / dot(d, d); }
34
35 point3 proj(point3 p, line3 ln)
36 { return ln.s + proj(p - ln.s, dir(ln)); }
37
38 point3 proj(point3 p, point3 p0, vec3 n) // projection on plane
39 { return p - proj(p - p0, n); }
40
41 vec3 reflect(vec3 v, vec3 n)
42 { return proj(v, n) * 2 - v; }
43
44 point3 reflect(point3 p, line3 ln)
45 { return ln.s + reflect(p - ln.s, dir(ln)); }
46
47 point3 reflect(point3 p, point3 p0, vec3 n) // reflection to plane
48 { return p - proj(p - p0, n) * 2; }
49
50 double angle(vec3 a, vec3 b)
51 { return acos(dot(a, b) / abs(a) / abs(b)); }
52
53 vec3 rotate(vec3 v, vec3 n, double a)
54 {
55     n = unit(n);
56     double cosa = cos(a), sina = sin(a);
57     return v * cosa + cross(n, v) * sina + n * dot(n, v) * (1 - cosa);
58 }

```

#### 4.14.1 Line

```

1 double dis(point3 p, line3 ln)
2 { return area2(p, ln.s, ln.t) / len(ln); }
3
4 double dtoseg(point3 p, line3 ln)
5 {

```

```

6      if (dcmp(dot(p - ln.s, dir(ln))) <= 0) return dis(p, ln.s);
7      if (dcmp(dot(p - ln.t, dir(ln))) >= 0) return dis(p, ln.t);
8      return dis(p, ln);
9  }
10
11 bool onseg(point3 p, line3 ln)
12 {
13     return zero(cross(p - ln.s, p - ln.t))
14         && dcmp(dot(p - ln.s, p - ln.t)) <= 0;
15 }
16
17 bool inter(line3 ln, point3 p0, vec3 n, point3 &p) // line & plane intersection
18 {
19     double d1 = dot(ln.s - p0, n);
20     double d2 = dot(ln.t - p0, n);
21     if (!dcmp(d1 - d2)) return false;
22     p = (ln.t * d1 - ln.s * d2) / (d1 - d2);
23     return true;
24 }
25
26 double dis(line3 a, line3 b)
27 {
28     vec3 n = cross(dir(a), dir(b));
29     if (zero(n)) return dis(a.s, b);
30     return fabs(dot(a.s - b.s, n)) / abs(n);
31 }
32
33 bool approach(line3 a, line3 b, point3 &p) // closest approach point of 2 lines
34 {
35     vec3 u = dir(a), v = dir(b), w = a.s - b.s;
36     double d = dot(u, u) * dot(v, v) - dot(u, v) * dot(u, v);
37     if (!dcmp(d)) return false; // parallel
38     double c = dot(u, v) * dot(v, w) - dot(v, v) * dot(u, w);
39     p = a.s + u * (c / d);
40     return true;
41 }

```

#### 4.14.2 Sphere

```

1 struct sphere {
2     point3 c;
3     double r;
4     sphere() {}
5     sphere(point3 c, double r): c(c), r(r) {}
6 };
7
8 bool inter(sphere s, line3 ln, point3 &p1, point3 &p2)
9 {
10     point3 p = proj(s.c, ln);
11     double d = abs(p - s.c);
12     if (dcmp(d - s.r) > 0) return false;
13     vec3 v = unit(dir(ln)) * sqrt(s.r * s.r - d * d);
14     p1 = p - v; p2 = p + v;
15     return true;
16 }

```

### 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  $\alpha$  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

```

1 struct face {
2     int v[3];
3     face(int a, int b, int c) { v[0] = a; v[1] = b; v[2] = c; }
4     int operator[](int i) const { return v[i % 3]; }
5 };
6
7 bool visible(point3 p[], face f, int i)
8 { return dcmp(vol6(p[f[0]], p[f[1]], p[f[2]], p[i])) > 0; }
9
10 vector<face> ch3d(point3 p[], int n)
11 {
12     static bool v[MAXN][MAXN];
13     int i, j, k;
14     for (i = 2; i < n && !dcmp(area2(p[0], p[1], p[i])); ++i) {}
15     swap(p[2], p[i]);
16     for (i = 3; i < n && !dcmp(vol6(p[0], p[1], p[2], p[i])); ++i) {}
17     swap(p[3], p[i]);
18     vector<face> cur;
19     cur.push_back(face(0, 1, 2));
20     cur.push_back(face(2, 1, 0));
21     for (i = 3; i < n; ++i) {
22         vector<face> next;
23         for (j = 0; j < cur.size(); ++j) {
24             face f = cur[j];
25             bool vis = visible(p, f, i);
26             if (!vis) next.push_back(f);
27             for (int k = 0; k < 3; ++k) v[f[k]][f[k + 1]] = vis;
28         }
29         for (j = 0; j < cur.size(); ++j) {
30             for (k = 0; k < 3; ++k) {
31                 int a = cur[j][k], b = cur[j][k + 1];
32                 if (v[a][b] && !v[b][a]) {
33                     next.push_back(face(a, b, i));
34                 }
35             }
36         }
37         cur.swap(next);
38     }
39     return cur;
40 }

```

## 4.16 Half-space Intersection

```

1 struct plane {
2     point3 p; vec3 n; // represent the half-space that n direct to
3     plane() {}
4     plane(point3 p, point3 n): p(p), n(n) {}
5 };

```

```

6
7 point3 inter(line3 ln, plane f)
8 {
9     double d1 = dot(ln.s - f.p, f.n);
10    double d2 = dot(ln.t - f.p, f.n);
11    return (ln.t * d1 - ln.s * d2) / (d1 - d2);
12 }
13
14 struct face {
15     point3 p[3];
16     face(point3 a, point3 b, point3 c) { p[0] = a, p[1] = b, p[2] = c; }
17     point3 &operator[](int i) { return p[i]; }
18     vec3 normal() { return cross(p[1] - p[0], p[2] - p[0]); }
19     void adjust(vec3 n) { if (dot(n, normal()) < 0) swap(p[0], p[1]); }
20 };
21
22 vector<face> cut(vector<face> h, plane f)
23 {
24     vector<face> ans;
25     point3 p0; int m = 0;
26     for (int i = 0; i < h.size(); ++i) {
27         vector<point3> c0, c1, c2;
28         for (int j = 0; j < 3; ++j) {
29             int d = dcmp(dot(h[i][j] - f.p, f.n));
30             if (d == 0) c0.push_back(h[i][j]);
31             else if (d > 0) c1.push_back(h[i][j]);
32             else c2.push_back(h[i][j]);
33         }
34         if (c0.size() == 3) {
35             if (dot(f.n, h[i].normal()) < 0) h.clear();
36             return h;
37         }
38         if (c0.size() == 1) {
39             if (c1.size() > c2.size()) c1.push_back(c0[0]);
40             else c2.push_back(c0[0]);
41         }
42         if (c0.size() == 2) c2.push_back(c0[0]), c2.push_back(c0[1]);
43         if (c1.size() == 3) ans.push_back(h[i]);
44         if (c1.size() == 3 || c2.size() == 3) continue;
45         point3 p1, p2; vec3 n = h[i].normal();
46         if (c1.size() == 1) {
47             p1 = inter(line3(c1[0], c2[0]), f);
48             p2 = inter(line3(c1[0], c2[1]), f);
49             ans.push_back(face(c1[0], p1, p2));
50             ans.back().adjust(n);
51         } else {
52             p1 = inter(line3(c1[0], c2[0]), f);
53             p2 = inter(line3(c1[1], c2[0]), f);
54             ans.push_back(face(c1[0], p1, p2));
55             ans.back().adjust(n);
56             ans.push_back(face(c1[0], c1[1], p2));
57             ans.back().adjust(n);
58         }
59         if (m++) {
60             ans.push_back(face(p0, p1, p2));
61             ans.back().adjust(f.n);
62         } else p0 = p1;
63     }
64     return ans;
65 }

```

# Chapter 5

## Number Theory

### 5.1 Fast Fourier Transform

```
1 typedef complex<double> cp;
2 void fft(cp a[], int n, int inv)
3 {
4     for(int i = 0, j = 0; i < n; i++) {
5         if(j > i) swap(a[i], a[j]);
6         int k = n;
7         while (j & (k >>= 1)) j ^= ~k;
8         j |= k;
9     }
10    for(int k = 1; k < n; k <= 1) {
11        double arg = inv * pi / k;
12        for(int i = 0; i < k; ++i) {
13            cp w = exp(cp(0, arg * i));
14            for(int j = i; j < n; j += k << 1) {
15                cp t = w * a[j + k];
16                a[j + k] = a[j] - t;
17                a[j] += t;
18            }
19        }
20    }
21    if(inv == -1) for(int i = 0; i < n; i++) a[i] /= n;
22 }
23
24 /*
25 Usage:
26 fft(a, n, 1); -- dft
27 fft(a, n, -1); -- idft
28 n should be 2^k
29 */
```

### 5.2 Primality Test(Miller-Rabin)

```
1 bool Witness(ll n, ll a)
2 {
3     ll m=(n-1), j=0;
4     while(!(m&1)) m>>=1, j++;
5     ll ans=Make_Power(a,m,n);
6     while(j-->0)
7     {
8         ll tmp=Make_Multi(ans,ans,n);
9         if(tmp==1 && ans!=1 && ans!=n-1) return 1;
10        ans=tmp;
11    }
```

```

12     if(ans!=1) return 1;
13     return 0;
14 }
15 bool Miller_Rabin(ll n)
16 {
17     if(n<2) return 0; if(n==2) return 1; if(!(n&1)) return 0;
18     for(int i=0; i<max_test; i++)
19     {
20         ll a=rand()%(n-2)+2;
21         if(Witness(n,a)) return 0;
22     }
23     return 1;
24 }

```

### 5.3 Integer Factorization(Pollard's $\rho$ Algorithm)

```

1  ll Pollard_Rho(ll n,ll c)
2  {
3      ll i=1,k=2,x=rand()%(n-1)+1,y=x,d;
4      while(1)
5      {
6          i++;
7          x=( Make_Multi(x,x,n)+c)%n;
8          d=Gcd(n,y-x);
9          if(d>1&&d<n) return d;
10         if(y==x) return n;
11         if(i==k) k<=1,y=x;
12     }
13 }

```

### 5.4 Extended Euclid's Algorithm

```

1  int exgcd(int a, int b, int &x, int &y)
2  {
3      if (b == 0) {
4          x = 1; y = 0;
5          return a;
6      } else {
7          int g = exgcd(b, a % b, y, x);
8          y -= (a / b) * x;
9          return g;
10     }
11 }

```

### 5.5 Euler's $\varphi$ Function

```

1  void phi_table()
2  {
3      for (int i = 2; i * i < MAX; ++i) {
4          if (!phi[i]) {
5              for (int k = (MAX - 1) / i, j = i * k;
6                  k >= i; --k, j -= i) {
7                  if (!phi[k]) phi[j] = i;
8                  // i is a prime factor of j
9              }
10         }
11     }
12     phi[1] = 1;
13     for (int i = 2; i < MAX; ++i) {

```

```

14         if (!phi[i]) {
15             phi[i] = i - 1;
16         } else {
17             int j = i / phi[i];
18             if (j % phi[i] == 0) phi[i] = phi[j] * phi[i];
19             else phi[i] = phi[j] * (phi[i] - 1);
20         }
21     }
22 }
23
24 //  $n = p1^{a1} * p2^{a2} * \dots$ 
25 //  $\phi[n] = n / p1 * (p1 - 1) / p2 * (p2 - 1) \dots$ 

```



# Chapter 6

## Others

### 6.1 Exact Cover(DLX)

```
1 int N, S[COL + 1], L[NODE], R[NODE], U[NODE], D[NODE], row[NODE], C[NODE];
2
3 void dlxinit(int c) // c Cumns, numbered from 1
4 {
5     for (int i = 0; i <= c; ++i) {
6         U[i] = D[i] = i;
7         L[i] = i - 1; R[i] = i + 1;
8         S[i] = 0;
9     }
10    L[0] = c; R[c] = 0; N = c + 1;
11 }
12
13 void addrow(const vector<int> &c)
14 {
15     int h = N;
16     for (int i = 0; i < c.size(); ++i) {
17         U[N] = U[c[i]]; D[N] = c[i];
18         D[U[N]] = U[D[N]] = N;
19         L[N] = N - 1; R[N] = N + 1;
20         ++S[C[N++]] = c[i];
21     }
22     L[h] = N - 1; R[N - 1] = h;
23 }
24
25 void remove(int c)
26 {
27     L[R[c]] = L[c];
28     R[L[c]] = R[c];
29     for (int i = D[c]; i != c; i = D[i]) {
30         for (int j = R[i]; j != i; j = R[j]) {
31             U[D[j]] = U[j];
32             D[U[j]] = D[j];
33             --S[C[j]];
34         }
35     }
36 }
37
38 void resume(int c)
39 {
40     for (int i = U[c]; i != c; i = U[i]) {
41         for (int j = L[i]; j != i; j = L[j]) {
42             U[D[j]] = j;
43             D[U[j]] = j;
```

```

44         ++S[C[j]];
45     }
46 }
47 L[R[c]] = c;
48 R[L[c]] = c;
49 }
50
51 bool dance(int d)
52 {
53     if (R[0] == 0) return true;
54     int c = R[0];
55     for (int i = R[0]; i; i = R[i]) {
56         if (S[i] < S[c]) c = i;
57     }
58     remove(c);
59     for (int i = D[c]; i != c; i = D[i]) {
60         // select row[i]
61         for (int j = R[i]; j != i; j = R[j]) remove(C[j]);
62         if (dance(d + 1)) return true;
63         for (int j = L[i]; j != i; j = L[j]) resume(C[j]);
64     }
65     resume(c);
66     return false;
67 }

```

## 6.2 Fuzzy Cover(DLX)

```

1 void remove(int i)
2 {
3     for (int j = D[i]; j != i; j = D[j]) {
4         R[L[j]] = R[j];
5         L[R[j]] = L[j];
6     }
7 }
8
9 void resume(int i)
10 {
11     for (int j = U[i]; j != i; j = U[j]) {
12         R[L[j]] = j;
13         L[R[j]] = j;
14     }
15 }
16
17 int h()
18 {
19     static int v[COL + 1], m;
20     int s = 0; ++m;
21     for (int i = R[0]; i; i = R[i]) {
22         if (v[i] == m) continue;
23         ++s; v[i] = m;
24         for (int j = D[i]; j != i; j = D[j]) {
25             for (int k = R[j]; k != j; k = R[k]) {
26                 v[C[k]] = m;
27             }
28         }
29     }
30     return s;
31 }
32
33 bool dance(int d)

```

```

34 {
35     if (!R[0]) return true;
36     if (d + h() > limit) return false;
37     int c = R[0];
38     for (int i = R[c]; i; i = R[i]) {
39         if (S[i] < S[c]) c = i;
40     }
41     for (int i = D[c]; i != c; i = D[i]) {
42         remove(i);
43         for (int j = R[i]; j != i; j = R[j]) remove(j);
44         if (dance(d + 1)) return true;
45         for (int j = L[i]; j != i; j = L[j]) resume(j);
46         resume(i);
47     }
48     return false;
49 }

```

### 6.3 3D Partial Order(Divide and Conquer)

```

1 struct triple {
2     int x, y, z;
3     bool operator<(const triple &b) const
4     {
5         return x != b.x ? x < b.x : y > b.y;
6         // return x != b.x ? x < b.x : y < b.y;
7     }
8 }v[MAXN];
9 int f[MAXN];
10
11 /*
12  solve LIS problem,
13  the longest chain that (x[i] < x[i+1] && y[i] < y[i+1] && z[i] < z[i+1])
14
15  for problem with (x[i] <= x[i+1] && y[i] <= y[i+1] && z[i] <= z[i+1]),
16  use the commented code instead
17 */
18 void solve(int l, int r)
19 {
20     if (r - l == 1) f[l] = max(f[l], 1);
21     if (r - l <= 1) return;
22     static int p[MAXN];
23     int mid = (l + r) / 2;
24     solve(l, mid);
25     for (int i = l; i < r; ++i) p[i] = i;
26     sort(p + l, p + r, [](int i, int j) {
27         return v[i].y != v[j].y ? v[i].y < v[j].y : i > j;
28         // return v[i].y != v[j].y ? v[i].y < v[j].y : i < j;
29     });
30     for (int i = l; i < r; ++i) {
31         if (p[i] < mid) bit.add(v[p[i]].z, f[p[i]]); // maintain maximum
32         else f[p[i]] = max(f[p[i]], bit.query(v[p[i]].z - 1) + 1);
33         // f[p[i]] = max(f[p[i]], bit.query(v[p[i]].z) + 1);
34     }
35     for (int i = l; i < mid; ++i) bit.clear(v[i].z);
36     solve(mid, r);
37 }
38
39 void solve()
40 {
41     sort(v, v + n);

```

```

42     static int z[MAXN];
43     for (int i = 0; i < n; ++i) z[i] = v[i].z;
44     sort(z, z + n);
45     int tot = unique(z, z + n) - z;
46     for (int i = 0; i < n; ++i) {
47         v[i].z = lower_bound(z, z + tot, v[i].z) - z + 1;
48         f[i] = 0;
49     }
50     solve(0, n);
51     return *max_element(f, f + n);
52 }

```

## 6.4 Power of Matrix

```

1  class matrix {
2  private:
3      int row, col;
4      vector<int> val;
5  public:
6      matrix(int r, int c): row(r), col(c), val(r * c) {}
7      matrix(int r, int c, int *v): row(r), col(c), val(v, v + r * c) {}
8      int rows() const { return row; }
9      int cols() const { return col; }
10     int get(int r, int c) const { return val[r * col + c]; }
11     void set(int r, int c, int v) { val[r * col + c] = v; }
12 };
13
14 matrix operator*(const matrix &lhs, const matrix &rhs)
15 {
16     matrix ret(lhs.rows(), rhs.cols());
17     for (int i = 0; i < lhs.rows(); ++i) {
18         for (int j = 0; j < rhs.cols(); ++j) {
19             int s = 0;
20             for (int k = 0; k < lhs.cols(); ++k) {
21                 s += lhs.get(i, k) * rhs.get(k, j);
22             }
23             ret.set(i, j, s);
24         }
25     }
26     return ret;
27 }
28
29 matrix pow(const matrix &mat, int k)
30 {
31     if (k == 1) return mat;
32     matrix ret = pow(mat, k >> 1);
33     return k & 1 ? ret * ret * mat : ret * ret;
34 }

```

## 6.5 Cantor Pairing Function

```

1  int cantor()
2  {
3      // s = a[0] * (n - 1)! + a[1] * (n - 2)! + ... + a[n - 1]
4      int s = 0;
5      for (int i = 0; i < n; ++i) {
6          int t = 0;
7          for (int j = i + 1; j < n; ++j) if (a[j] < a[i]) ++t;
8          s = (s + t) * (n - i - 1);
9      }

```

```

10         return s;
11     }
12
13     int uncantor(int s)
14     {
15         memset(u, 0, sizeof(u));
16         for (int i = 0; i < n; ++i) {
17             int t = s / fac[n - i - 1];
18             s -= t * fac[n - i - 1];
19             int l = 0;
20             for (int j = 0; l <= t; ++j) if (!u[j]) ++l;
21             u[a[i] = --j] = true;
22         }
23     }

```

## 6.6 Adaptive Simpson's Method

```

1 double simpson(double a, double b)
2 {
3     return (b - a) / 6 * (f(a) + 4 * f((a + b) * 0.5) + f(b));
4 }
5
6 double rsimpson(double a, double b)
7 {
8     double m = (a + b) * 0.5;
9     double s = simpson(a, b);
10    double s1 = simpson(a, m);
11    double s2 = simpson(m, b);
12    if (fabs(s1 + s2 - s) < ESP) return s;
13    return rsimpson(a, m) + rsimpson(m, b);
14 }

```

## 6.7 Linear Programming(Simplex)

```

1 int n, m;
2 double a[MAXM][MAXN];
3 double x[MAXN];
4 int N[MAXN], B[MAXM];
5 const double eps = 1e-10, inf = 1e100;
6
7 // a[i][0]*x[0] + a[i][1]*x[1] + ... <= a[i][n]
8 // max(a[m][0]*x[0] + a[m][1]*x[1] + ... - a[m][n])
9 // x[i] >= 0
10
11 void pivot(int r, int c)
12 {
13     swap(N[c], B[r]);
14     a[r][c] = 1 / a[r][c];
15     for (int i = 0; i <= n; ++i) if (i != c) a[r][i] *= a[r][c];
16     for (int i = 0; i <= m; ++i) if (i != r) {
17         for (int j = 0; j <= n; ++j) if (j != c) {
18             a[i][j] -= a[i][c] * a[r][j];
19         }
20         a[i][c] *= -a[r][c];
21     }
22 }
23
24 bool feasible()
25 {
26     for (;;) {

```

```

27         int r, c;
28         double p = inf;
29         for (int i = 1; i < m; ++i) if (a[i][n] < p) p = a[r = i][n];
30         if (p > -eps) return true;
31         p = 0;
32         for (int i = 1; i < n; ++i) if (a[r][i] < p) p = a[r][c = i];
33         if (p > -eps) return false;
34         p = a[r][n] / a[r][c];
35         for (int i = r + 1; i < m; ++i) if (a[i][c] > eps) {
36             double v = a[i][n] / a[i][c];
37             if (v < p) r = i, p = v;
38         }
39         pivot(r, c);
40     }
41 }
42
43 int simplex() // 0 - no solution, -1 - infinity, 1 - has a solution
44 {
45     for (int i = 0; i < n; ++i) N[i] = i;
46     for (int i = 0; i < m; ++i) B[i] = n + i;
47     if (!feasible()) return 0;
48     for (;;) {
49         int r, c;
50         double p = 0;
51         for (int i = 0; i < n; ++i) if (a[m][i] > p) p = a[m][c = i];
52         if (p < eps) break;
53         p = inf;
54         for (int i = 0; i < m; ++i) if (a[i][c] > eps) {
55             double v = a[i][n] / a[i][c];
56             if (v < p) r = i, p = v;
57         }
58         if (p == inf) return -1;
59         pivot(r, c);
60     }
61     for (int i = 0; i < n; ++i) if (N[i] < n) x[N[i]] = 0;
62     for (int i = 0; i < m; ++i) if (B[i] < n) x[B[i]] = a[i][n];
63     ans = -a[m][n];
64     return 1;
65 }

```

# Appendix A

## Snippets

```
1 | const int INF = 0x3f3f3f3f; // == 1061109567 < INT_MAX / 2
2 | const int INF = 0x2a2a2a2a; // == 707406378 < INT_MAX / 3
3 | const int INF = 0xc2c2c2c2; // == -1027423550 > INT_MIN / 2
4 | const int INF = 0xd5d5d5d5; // == -707406379 > INT_MIN / 3
5 |
6 | // gcc stack enlarging
7 | char *p = (char *)malloc(size) + size;
8 | __asm("movl□%0,%%esp□\n" :: "r"(p));
9 | // vc stack enlarging
10 | #pragma comment(linker, "/STACK:16777216")
```

# Appendix B

## Java Example

```
1 import java.util.Scanner;
2 import java.util.Arrays;
3 import java.math.BigInteger; // or BigDecimal
4
5 public class Main {
6     public static void main(String[] args) {
7         Scanner cin = new Scanner(System.in);
8         int n = cin.nextInt();
9         System.out.print(n);
10        System.out.println(n);
11
12        int[] arr = new int[5];
13        int[] arrr = {1, 2, 3, 4, 5};
14        int[][] f = new int[n][n];
15        Arrays.sort(arrr);
16
17        BigInteger a = cin.nextBigInteger();
18        BigInteger b = BigInteger.valueOf(2);
19        a = a.add(b);      a = a.subtract(b);      a = a.negate();
20        a = a.multiply(b); a = a.divide(b);        a = a.mod(b);
21        a = a.shiftLeft(1); a = a.shiftRight(1);
22        if (a.compareTo(b) < 0) System.out.println("a<b");
23    }
24 }
```



## Appendix C

# Vim Configuration

```
1 | se nocp nu cin ts=4 sw=4
2 | syn on
3 |
4 | se mp=g++\ -g\ -o\ %<\ %\ -Wall\ -std=c++11
5 | map mk :make<cr>
6 | map mr :!./%<<cr>
7 | map mw :!./%< < %<.in<cr>
8 | map mi :sp %<.in<cr>
```