

Code Template for ACM-ICPC

wodesuck
SYSU_ArthasMenethil
Sun Yat-sen University

September 17, 2014

Contents

| | | |
|----------|---|-----------|
| 1 | Graph/Tree Theory | 3 |
| 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 |
| 1.6 | Matching | 7 |
| 1.6.1 | Maximum Bipartite Matching(Hungarian) | 7 |
| 1.6.2 | Maximum Weight Perfect Biparite Matching(KM) | 7 |
| 1.6.3 | Maximum Matching on General Graph(Blossom Algorithm) | 8 |
| 1.6.4 | Maximum Weight Perfect Matching on General Graph(Randomize Greedy Matching) | 9 |
| 1.7 | 2-SAT | 10 |
| 1.8 | Divide and Conquer for Tree | 12 |
| 1.9 | Heavy-Light Decomposition | 14 |
| 2 | Data Structures | 16 |
| 2.1 | Segment Tree | 16 |
| 2.1.1 | zkw Segment Tree | 16 |
| 2.1.2 | Functional Segment Tree | 16 |
| 2.2 | Self-balancing BST | 17 |
| 2.2.1 | Size Balanced Tree | 17 |
| 2.2.2 | Splay | 18 |
| 2.2.3 | Functional Treap | 19 |
| 2.2.4 | Functional Treap(Range Operation) | 19 |
| 2.3 | Leftist Tree | 20 |
| 2.4 | Dynamic Tree | 21 |
| 2.4.1 | Link-cut Tree | 21 |
| 2.4.2 | Euler Tour Tree | 23 |
| 2.5 | KD Tree | 24 |
| 2.6 | Sparse Table | 25 |
| 3 | Stringology | 26 |
| 3.1 | KMP Algorithm | 26 |
| 3.2 | Extend-KMP Algorithm | 26 |
| 3.3 | Aho-Corasick Automation | 27 |
| 3.4 | Suffix Array | 28 |
| 3.5 | Suffix Automation | 29 |
| 3.6 | Longest Palindorme Substring(Manacher) | 30 |
| 3.7 | Minimum Representation | 30 |

| | | |
|----------|---|-----------|
| 4 | Computational Geometry | 31 |
| 4.1 | Basic Operations | 31 |
| 4.1.1 | Line | 31 |
| 4.1.2 | Triangle | 32 |
| 4.1.3 | Circle | 33 |
| 4.2 | Point in Polygon Problem | 34 |
| 4.3 | Convex Hull(Graham) | 35 |
| 4.4 | Half-plane Intersection | 35 |
| 4.5 | Closest Pair(Divide and Conquer) | 35 |
| 4.6 | Farthest Pair(Rotating Caliper) | 36 |
| 4.7 | Minimum Distance Between Convex Hull(Rotating Caliper) | 36 |
| 4.8 | Union Area of a Circle and a Polygon | 37 |
| 4.9 | Union Area of Circles | 37 |
| 4.10 | Union Area of Polygons | 38 |
| 4.11 | Minimum Enclosing Circle(Randomized Incremental Method) | 38 |
| 4.12 | 3D Computational Geometry | 39 |
| 4.12.1 | Line | 40 |
| 4.12.2 | Sphere | 41 |
| 4.13 | Convex Hull in 3D | 41 |
| 5 | Number Theory | 43 |
| 5.1 | Fast Fourier Transform | 43 |
| 5.2 | Primality Test(Miller-Rabin) | 43 |
| 5.3 | Integer Factorization(Pollard's ρ Algorithm) | 44 |
| 5.4 | Extended Euclid's Algorithm | 44 |
| 5.5 | Euler's φ Function | 44 |
| 6 | Others | 46 |
| 6.1 | Exact Cover(DLX) | 46 |
| 6.2 | Fuzzy Cover(DLX) | 47 |
| 6.3 | Power of Matrix | 48 |
| 6.4 | Cantor Pairing Function | 48 |
| 6.5 | Adaptive Simpson's Method | 49 |
| A | Snippets | 50 |
| B | Java Example | 51 |
| C | Vim Configuration | 52 |

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 f)
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 != f) {
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             grp[v] = cnt;
20         } while (v != u);
21         ++cnt;
22     }
23 }

```

1.4 Lowest Common Ancestor(Tarjan)

```

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

```

```

27 Usage:
28     memset(h, 0, sizeof(h));
29     memset(vh, 0, sizeof(vh));
30     vh[0] = esz;
31     while (h[s] != esz) flow += aug(s, INT_MAX);
32 */

```

1.5.2 Minimum Cost Maximum Flow(Primal-Dual)

```

1  int esz, psz, s, t;
2  int cost, dist, d[MAXV];
3  bool v[MAXV];
4
5  int aug(int p, int m)
6  {
7      if (p == t) return cost += dist * m, m;
8      int d = m; v[p] = true;
9      for (edge *i = e[p]; i; i = i->next) {
10         if (i->u && !i->c && !v[i->t]) {
11             int f = aug(i->t, min(d, i->u));
12             i->u -= f; i->pair->u += f; d -= f;
13             if (!d) break;
14         }
15     }
16     return m - d;
17 }
18
19 bool modlabel()
20 {
21     deque<int> q;
22     memset(v, 0, sizeof(v));
23     memset(d, 0x3f, sizeof(d));
24     q.push_back(s); d[s] = 0; v[s] = true;
25     while (!q.empty()) {
26         int u = q.front(); q.pop_front(); v[u] = false;
27         for (edge *i = e[u]; i; i = i->next) {
28             if (i->u && d[u] + i->c < d[i->t]) {
29                 d[i->t] = d[u] + i->c;
30                 if (!v[i->t]) {
31                     if (q.empty() || d[i->t] < d[q.front()])
32                         {
33                             q.push_front(i->t);
34                         } else {
35                             q.push_back(i->t);
36                         }
37                     v[i->t] = true;
38                 }
39             }
40         }
41     }
42     if (d[t] == inf) return false;
43     for (int i = 0; i < esz; ++i) {
44         for (edge *j = e[i]; j; j = j->next) {
45             j->c += d[i] - d[j->t];
46         }
47     }
48     dist += d[t];
49     return true;
50 }
51 /*

```

```

52 Initialize:
53     psz = 0; memset(e, 0, sizeof(e));
54     cost = dist = 0;
55 Usage:
56     while (modlabel()) {
57         do memset(v, 0, sizeof(v));
58         while (aug(s, INT_MAX));
59     }
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 }
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 | int n; // n vars
2 | struct edge {

```

```

3         int t;
4         edge *next;
5     }epool[CONDITIONS * 2], *e[MAXN * 2], *e2[MAXN * 2];
6     // sat variable Bi and !Bi are encoded as i<<1 and i<<1^1
7     int psz;
8     int stamp, dfn[MAXN * 2], low[MAXN * 2];
9     int top, sta[MAXN * 2];
10    bool ins[MAXN * 2];
11    int cnt, grp[MAXN * 2];
12    int deg[MAXN * 2], con[MAXN * 2], mrk[MAXN * 2];
13    int head, tail, que[MAXN * 2];
14    // Bi is true if mrk[grp[i << 1]] == 1
15
16    bool chk()
17    {
18        cnt = 0; stamp = 0; top = 0;
19        memset(dfn, 0, sizeof(dfn));
20        memset(low, 0, sizeof(low));
21        memset(ins, 0, sizeof(ins));
22        memset(grp, 0, sizeof(grp));
23        for (int i = 0; i < (n<<1); ++i) if (!dfn[i]) tarjan(i);
24        for (int i = 0; i < n; ++i) {
25            int u = grp[i<<1], v = grp[(i<<1)^1];
26            if (u == v) return false;
27            con[u] = v; con[v] = u;
28        }
29        return true;
30    }
31
32    void group()
33    {
34        memset(e2, 0, sizeof(e2));
35        memset(deg, 0, sizeof(deg));
36        for (int i = 0; i < (n<<1); ++i) {
37            for (edge *j = e[i]; j; j = j->next) {
38                int x = grp[i], y = grp[j->t];
39                if (x == y) continue;
40                add_edge2(y, x);
41                ++deg[x];
42            }
43        }
44    }
45
46    void tsort()
47    {
48        memset(mrk, 0, sizeof(mrk));
49        for (int i = 0; i < cnt; ++i) if (!deg[i]) que[tail++] = i;
50        while (head < tail) {
51            int u = que[head++];
52            if (!mrk[u]) mrk[u] = 1, mrk[con[u]] = -1;
53            for (edge *i = e2[u]; i; i = i->next) {
54                if (!--deg[i->t]) que[tail++] = i->t;
55            }
56        }
57    }
58
59    /*
60    Initialize:
61        psz = 0; memset(e, 0, sizeof(e));
62        add_edge(i<<1, j<<1^1); for Bi -> !Bj
63    Usage:

```

```

64         if (chk()) {
65             group();
66             tsort();
67         } else {
68             // No solution
69         }
70 */

```

1.8 Divide and Conquer for Tree

poj 1741 Tree

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <algorithm>
4
5  using namespace std;
6
7  int n, k;
8  const int MAXN = 10001;
9  struct edge {
10     int t, w;
11     edge *next;
12 }epool[MAXN * 2], *e[MAXN];
13 int psz;
14 int rec[MAXN], tot;
15 int size[MAXN], msize[MAXN], dist[MAXN];
16 bool del[MAXN];
17 int w[MAXN], c[MAXN];
18 int ans;
19
20 void getsize(int p, int fa)
21 {
22     rec[tot++] = p; size[p] = 1; msize[p] = 0;
23     for (edge *i = e[p]; i; i = i->next) {
24         if (i->t == fa || del[i->t]) continue;
25         getsize(i->t, p);
26         size[p] += size[i->t];
27         msize[p] = max(msize[p], size[i->t]);
28     }
29 }
30
31 int centroid(int p)
32 {
33     tot = 0;
34     getsize(p, 0);
35     int k = rec[0];
36     for (int i = 0; i < tot; ++i) {
37         int t = rec[i];
38         msize[t] = max(msize[t], tot - size[t]);
39         if (msize[t] < msize[k]) k = t;
40     }
41     return k;
42 }
43
44 void getdist(int p, int fa)
45 {
46     for (edge *i = e[p]; i; i = i->next) {
47         if (i->t == fa || del[i->t]) continue;
48         w[tot++] = dist[i->t] = dist[p] + i->w;
49         getdist(i->t, p);

```

```

50     }
51 }
52
53 void add(int p, int x)
54 {
55     for (; p <= tot; p += p & -p) c[p] += x;
56 }
57
58 int sum(int p)
59 {
60     int s = 0;
61     for (; p; p -= p & -p) s += c[p];
62     return s;
63 }
64
65 void dfs1(int p, int fa)
66 {
67     ans += sum(upper_bound(w, w + tot, k - dist[p]) - w);
68     for (edge *i = e[p]; i; i = i->next) {
69         if (i->t != fa && !del[i->t]) dfs1(i->t, p);
70     }
71 }
72
73 void dfs2(int p, int fa)
74 {
75     add(upper_bound(w, w + tot, dist[p]) - w, 1);
76     for (edge *i = e[p]; i; i = i->next) {
77         if (i->t != fa && !del[i->t]) dfs2(i->t, p);
78     }
79 }
80
81 void solve(int p)
82 {
83     p = centroid(p);
84     del[p] = true;
85
86     dist[p] = 0; tot = 0;
87     getdist(p, 0);
88     sort(w, w + tot);
89     tot = unique(w, w + tot) - w;
90     fill(c, c + tot + 1, 0);
91
92     for (edge *i = e[p]; i; i = i->next) {
93         if (del[i->t]) continue;
94         dfs1(i->t, p);
95         dfs2(i->t, p);
96     }
97     ans += sum(upper_bound(w, w + tot, k) - w);
98
99     for (edge *i = e[p]; i; i = i->next) {
100         if (!del[i->t]) solve(i->t);
101     }
102 }
103
104 void add_edge(int u, int v, int w)
105 {
106     edge *tmp = epool + psz++;
107     tmp->t = v; tmp->w = w; tmp->next = e[u]; e[u] = tmp;
108 }
109
110 void init()

```

```

111 {
112     psz = 0; ans = 0;
113     memset(e, 0, sizeof(e));
114     memset(del, 0, sizeof(del));
115     for (int i = 1; i < n; ++i) {
116         int u, v, w;
117         scanf("%d%d%d", &u, &v, &w);
118         add_edge(u, v, w);
119         add_edge(v, u, w);
120     }
121 }
122
123 int main()
124 {
125     while (scanf("%d%d", &n, &k) != EOF && (n || k)) {
126         init();
127         solve(1);
128         printf("%d\n", ans);
129     }
130 }

```

1.9 Heavy-Light Decomposition

```

1 void dfs1(int p)
2 {
3     size[p] = 1; h[p] = -1;
4     for (edge *i = e[p]; i; i = i->next) {
5         if (i->t == f[p]) continue;
6         f[i->t] = p;
7         d[i->t] = d[p] + 1;
8         dfs1(i->t);
9         size[p] += size[i->t];
10        if (h[p] < 0 || size[i->t] > size[h[p]]) h[p] = i->t;
11    }
12 }
13
14 void dfs2(int p, int anc)
15 {
16     top[p] = anc;
17     if (h[p] >= 0) dfs2(h[p], anc);
18     for (edge *i = e[p]; i; i = i->next) {
19         if (i->t != f[p] && i->t != h[p]) dfs2(i->t, i->t);
20     }
21 }
22
23 int lca(int u, int v)
24 {
25     while (top[u] != top[v]) {
26         if (d[top[u]] < d[top[v]]) swap(u, v);
27         u = f[top[u]];
28     }
29     if (d[u] > d[v]) swap(u, v);
30     return u;
31 }
32
33 /*
34 Data:
35 f[]      -- father
36 d[]      -- depth
37 size[]   -- subtree size

```

```

38 |         h[]    -- heavy child
39 |         top[]  -- head node of chain
40 | Initialize:
41 |         f[1] = -1;
42 |         d[1] = 0;
43 | */

```


Chapter 2

Data Structures

2.1 Segment Tree

2.1.1 zkw Segment Tree

```
1 // M should be 2^k, and should satisfy M >= n + 2,
2 // where the range of query indexes is [1, n]
3 int T[M<<1];
4
5 void change(int p, int x)
6 {
7     v[p += M] = x;
8     while (p>>=1) T[p] = max(T[p<<1], T[p<<1^1]);
9 }
10
11 int query(int s, int t)
12 {
13     int l = INT_MIN, r = INT_MIN;
14     for (s += m - 1, t += m + 1; s^t^1; s>>=1, t>>=1) {
15         if (~s&1) l = max(w, T[s^1]);
16         if (t&1) r = max(w, T[t^1]);
17     }
18     return max(l, r);
19 }
```

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 }
42
43 int erase(sbt *&t, int v)
44 {
45     --t->sz;
46     if (v == t->k || t->ch[v > t->k] == null) {
47         v = t->k;
48         if (t->ch[0] == null) t = t->ch[1];
49         else if (t->ch[1] == null) t = t->ch[0];
50         else t->k = erase(t->ch[0], v + 1);
51         return v;
52     }
53     return erase(t->ch[v > t->k], v);
54 }
55
56 sbt *find(sbt *t, int v)
57 {
58     if (t == null) return 0;
59     if (v == t->k) return t;
60     return find(t->ch[v > t->k], v);
61 }
62
63 int rank(sbt *t, int v)
64 {
65     if (t == null) return 0;
66     else if (v < t->k) return rank(t->ch[0], v);
67     else return t->ch[0]->sz + 1 + rank(t->ch[1], v);
68 }
69
70 sbt *select(int t, int k)
71 {
72     if (k == t->ch[0]->sz + 1) return t;
73     else if (k <= t->ch[0]->sz) return select(t->ch[0], k);
74     else return select(t->ch[1], k - t->ch[0]->sz - 1);
75 }

```

2.2.2 Splay

```

1 void zig(int t)
2 {
3     int p = parent[t], g = parent[p];
4     if (right[t]) parent[right[t]] = p;
5     left[p] = right[t]; right[t] = p;
6     parent[p] = t; parent[t] = g;
7     update(p); update(t);
8     if (g) p == left[g] ? left[g] = t : right[g] = t;
9 }
10
11 void zag(int t)
12 {
13     int p = parent[t], g = parent[p];

```

```

14     if (left[t]) parent[left[t]] = p;
15     right[p] = left[t]; left[t] = p;
16     parent[p] = t; parent[t] = g;
17     update(p); update(t);
18     if (g) p == left[g] ? left[g] = t : right[g] = t;
19 }
20
21 void splay(int t, int header = 0) // header = parent[root]
22 {
23     int p = parent[t], g = parent[p];
24     for (; p != header; p = parent[t], g = parent[p]) {
25         if (g == header) t == left[p] ? zig(t) : zag(t);
26         else if (p == left[g]) t == left[p] ? zig(p) : zag(t), zig(t);
27         else t == right[p] ? zag(p) : zig(t), zag(t);
28     }
29 }

```

2.2.3 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 }

```

2.2.4 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 | int left[MAXN], right[MAXN], parent[MAXN], size[MAXN];
2 | bool rev[MAXN];
3 |
4 | void update(int t)
5 | {
6 |     size[t] = size[left[t]] + size[right[t]] + 1;
7 | }
8 |
9 | void revsub(int t)
10 | {
11 |     swap(left[t], right[t]);
12 |     rev[t] ^= 1;
13 | }
14 |
15 | void sink(int t)
16 | {
17 |     if (rev[t]) {
18 |         if (left[t]) revsub(left[t]);
19 |         if (right[t]) revsub(right[t]);
20 |         rev[t] = false;
21 |     }
22 | }
23 |
24 | void sinkdown(int t)
25 | {
26 |     static int path[MAXN];
27 |     int n = 0;
28 |     for (;;) {
29 |         path[n++] = t;
30 |         int p = parent[t];
31 |         if (t != left[p] && t != right[p]) break;
32 |         t = p;
33 |     }
34 |     while (n) sink(path[--n]);
35 | }
36 |
37 | void zig(int t)
38 | {
39 |     int p = parent[t], g = parent[p];
40 |     if (right[t]) parent[right[t]] = p;
41 |     left[p] = right[t]; right[t] = p;
42 |     parent[p] = t; parent[t] = g;

```

```

43     update(p); update(t);
44     if (p == left[g]) left[g] = t;
45     else if (p == right[g]) right[g] = t;
46 }
47
48 void zag(int t)
49 {
50     int p = parent[t], g = parent[p];
51     if (left[t]) parent[left[t]] = p;
52     right[p] = left[t]; left[t] = p;
53     parent[p] = t; parent[t] = g;
54     update(p); update(t);
55     if (p == left[g]) left[g] = t;
56     else if (p == right[g]) right[g] = t;
57 }
58
59 void splay(int t)
60 {
61     sinkdown(t);
62     for (;;) {
63         int p = parent[t], g = parent[p];
64         if (t == left[p]) {
65             if (p == left[g]) zig(p), zig(t);
66             else if (p == right[g]) zig(t), zag(t);
67             else zig(t);
68         } else if (t == right[p]) {
69             if (p == left[g]) zag(t), zig(t);
70             else if (p == right[g]) zag(p), zag(t);
71             else zag(t);
72         } else break;
73     }
74 }
75
76 int expose(int t)
77 {
78     int u = 0;
79     for (; t; u = t, t = parent[t]) {
80         splay(t);
81         right[t] = u;
82         update(t);
83     }
84     return u;
85 }
86
87 void link(int t, int p) // link subtree t to p
88 {
89     parent[expose(t)] = p;
90     expose(t);
91 }
92
93 void cut(int t)
94 {
95     expose(t);
96     splay(t);
97     parent[left[t]] = 0;
98     left[t] = 0;
99     update(t);
100 }
101
102 void setroot(int t)
103 {

```

```

104         revsub(expose(t));
105     }
106
107 void query(int u, int v) // query path u -> v
108 {
109     expose(u);
110     int t = expose(v); // t == lca(u, v)
111     // analysis node t
112     // analysis right[t]
113     if (u != t) {
114         splay(u);
115         // analysis u
116     }
117 }
118
119 void change(int u, int v) // path change
120 {
121     expose(u);
122     int t = expose(v); // t == lca(u, v)
123     // change node t
124     if (right[t]) // change right[t]
125     if (u != t) {
126         splay(u);
127         // change u
128     }
129 }

```

2.4.2 Euler Tour Tree

```

1  int n;
2  int left[MAXN * 2], right[MAXN * 2], parent[MAXN * 2], size[MAXN * 2];
3  // each node split into 2 nodes. i --> (i<<1) && (i<<1)^1
4
5  void update(int t) { size[t] = size[left[t]] + size[right[t]] + 1; }
6
7  int root(int t) { while (parent[t]) t = parent[t]; return t; }
8  int minnode(int t) { while (left[t]) t = left[t]; return t; }
9  int maxnode(int t) { while (right[t]) t = right[t]; return t; }
10
11 int prev(int t)
12 {
13     if (left[t]) return maxnode(left[t]);
14     int p = parent[t];
15     while (p && t == left[p]) t = p, p = parent[t];
16     return p;
17 }
18
19 int succ(int t)
20 {
21     if (right[t]) return minnode(right[t]);
22     int p = parent[t];
23     while (p && t == right[p]) t = p, p = parent[t];
24     return p;
25 }
26
27 void cut(int t)
28 {
29     int x = prev(t<<1), y = succ(t<<1^1);
30     splay(x); splay(y, x);
31     parent[left[y]] = 0; left[y] = 0;
32     update(y); update(x);

```



```

33 }
34
35 void link(int t, int p) // link subtree t to p
36 {
37     p <= 1; t = root(t<<1);
38     splay(p); splay(minnode(right[p]), p);
39     parent[t] = right[p]; left[right[p]] = t;
40     update(right[p]); update(p);
41 }

```

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 sqr(double n) { return n * n; }
8
9  double dis(kd a, kd b)
10 {
11     double s = 0;
12     for (int i = 0; i < K; ++i) s += sqr(a.x[i] - b.x[i]);
13     return sqrt(s);
14 }
15
16 struct cmpk {
17     int k;
18     cmpk(int k): k(k) {}
19     bool operator()(const kd &a, const kd &b)
20     { return a.x[k] < b.x[k]; }
21 };
22
23 void build(int l, int r, int d)
24 {
25     if (r - l <= 1) return;
26     int mid = (l + r) >> 1;
27     nth_element(t + l, t + mid, t + r, cmpk(d));
28     if (++d == K) d = 0;
29     build(l, mid, d); build(mid + 1, r, d);
30 }
31
32 typedef priority_queue<pair<double, int> > heap;
33 void knn(int l, int r, int d, kd p, size_t k, heap &h)
34 {
35     if (r - l < 1) return;
36     int mid = (l + r) >> 1;
37     h.push(make_pair(dis(p, t[mid]), t[mid].id));
38     if (h.size() > k) h.pop();
39     double dx = p.x[d] - t[mid].x[d];
40     if (++d == K) d = 0;
41     if (dx < 0) {
42         knn(l, mid, d, p, k, h);
43         if (h.top().first > dx) knn(mid + 1, r, d, p, k, h);
44     } else {
45         knn(mid + 1, r, d, p, k, h);
46         if (h.top().first > dx) knn(l, mid, d, p, k, h);
47     }
48 }

```

```

49
50 /*
51 Usage:
52     build(0, n, 0);
53     knn(0, n, 0, pos, ans_heap);
54 */

```

2.6 Sparse Table

```

1  int f[MAXN][LOGN];
2
3  void st_init(int *a, int n)
4  {
5      for (int i = 0; i < n; ++i) f[i][0] = a[i];
6      for (int j = 1; (1<<j) <= n; ++j) {
7          for (int i = 0; i + (1<<j) <= n; ++i) {
8              f[i][j] = min(f[i][j - 1], f[i + (1<<(j-1))][j - 1]);
9          }
10     }
11 }
12
13 int rmq(int l, int r)
14 {
15     int k = 0;
16     while ((1<<(k+1)) <= r - l + 1) ++k;
17     return min(d[l][k], d[r - (1<<k) + 1][k]);
18 }

```

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->f->ch[i];
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; i = i->last) {
47         //     sum += i->val, i->val = 0;
48         // } # method 2
49     }
50     return sum;
51 }
52
53 /*
54 Initialize:
55     psz = 1; memset(dict = pool, 0, sizeof(trie));
56 Method 1: counting appearance times
57 Method 2: counting appear patterns
58 ** pattern that appear more than once counted 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 // REMEMBER: 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 // Initialize: 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 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(Gramham)

```
1 bool cmpx(point a, point b) { return dcmp(a.X - b.X) ? a.X < b.X : a.Y < b.Y; }
2
3 int gramham(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 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     int head = 0, tail = 0, m = 0;
9     for (int i = 0; i < n; ++i) q[i].a = arg(dir(q[i]));
10    sort(ln, ln + n, cmpang);
11    for (int i = 1; i < n; ++i) {
12        if (!dcmp(q[i].a - q[i - 1].a)) continue;
13        while (head < tail && !inhp(h[tail - 1], q[i])) --tail;
14        while (head < tail && !inhp(h[head], q[i])) ++head;
15        q[++tail] = q[i];
16        if (head < tail) h[tail - 1] = inter(q[tail - 1], q[tail]);
17    }
18    while (head < tail && !inhp(h[tail - 1], q[head])) --tail;
19    if (head < tail) h[tail] = inter(q[tail], q[head]);
20    for (int i = head; i <= tail; ++i) h[m++] = h[i];
21    return m;
22 }
23
24 line makehp(double a, double b, double c) // ax + by + c > 0
25 {
26     point p1 = fabs(a) > fabs(b) ? point(-c / a, 0) : point(0, -c / b);
27     point p2 = p1 + vec(b, -a);
28     return line(p1, p2);
29 }
```

4.5 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 */

```

4.6 Farthest Pair(Rotating Caliper)

```

1 double maxdis(point *p, int n)
2 {
3     gramham(p, n, h, m);
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 }

```

4.7 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 = n + n; 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 }

```

4.8 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 }
```

4.9 Union Area of Circles

```
1 bool incir(circle a, circle b)
2 { return dcmp(abs(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     memset(tot, 0, sizeof(tot));
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 (incir(c[i], c[j])) { ++k; continue; }
13            if (!inter(c[i], c[j], a1, a2)) continue;
14            a[m++] = make_pair(a1, 1);
15            a[m++] = make_pair(a2, -1);
16            if (a1 > a2) ++k;
17        }
18        sort(a, a + m);
19        double a1 = a[m - 1].first - 2 * pi, a2, rad;
20        for (int j = 0; j < m; ++j) {
21            a2 = a[j].first, rad = a2 - a1;
22            tot[k] += .5 * sqr(c[i].r) * (rad - sin(rad));
23            tot[k] += .5 * cross(c[i].p(a1), c[i].p(a2));
24            k += a[j].second;
25            a1 = a2;
26        }
27        if (!m) tot[k] += pi * sqr(c[i].r);
28    }
29 }
30
31 /*
32 tot[0] = the area of union
```

```

33 | tot[n-1]           = the aera of intersection
34 | tot[k-1] - tot[k] = the aera covered k times
35 | */

```

4.10 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 |     memset(tot, 0, sizeof(tot));
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 | */

```

4.11 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 }

```

4.12 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 }

```

4.12.1 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 | }
```

4.12.2 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 | }
```

4.13 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 |     }
```

Chapter 5

Number Theory

5.1 Fast Fourier Transform

```
1 typedef complex<double> cp;
2 void fft(cp *a, int n, int f)
3 {
4     static cp b[MAXN];
5     double arg = pi;
6     for (int k = n >> 1; k >= 1, arg *= 0.5) {
7         cp wm = polar(1.0, f * arg), w(1, 0);
8         for (int i = 0; i < n; i += k, w *= wm) {
9             int p = i << 1;
10            if (p >= n) p -= n;
11            for (int j = 0; j < k; ++j) {
12                b[i + j] = a[p + j] + w * a[p + k + j];
13            }
14        }
15        for (int i = 0; i < n; ++i) a[i] = b[i];
16    }
17 }
18
19 /*
20 Usage:
21 fft(a, n, 1); -- dft
22 fft(a, n, -1); for(i) a[i]/=n; -- idft
23 n should be 2^k
24 */
```

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 ρ 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 φ 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 * ...
25 | // phi[n] = n / p1 * (p1 - 1) / p2 * (p2 - 1) ...

```

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 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 }

```

6.4 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 }

```

6.5 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 }

```

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 | se go= bs=2 "␣for␣gvim
3 | syn␣on
```