

## Contents

<b>1</b>	<b>1 Templates</b>	<b>1</b>
1.1	Start	1
1.2	Template - C++	1
1.3	Template - Java	1
<b>2</b>	<b>2 Data Structures</b>	<b>2</b>
2.1	AVL Tree	2
2.2	BIT	2
2.3	KD Tree	2
2.4	LCA	3
2.5	Lazy Segment Tree	4
2.6	Segment Tree 2D	5
2.7	Segment Tree	6
2.8	Union Find	6
<b>3</b>	<b>3 Graph</b>	<b>7</b>
3.1	2-SAT	7
3.2	Articulation and Bridge	7
3.3	Eulerian Path	8
3.4	Heavy Light	8
3.5	Poset Width	8
3.6	SCC	9
3.7	Topological Sort	9
<b>4</b>	<b>4 Combinatorial Optimization</b>	<b>9</b>
4.1	Bipartite Graph	9
4.2	Max Flow - Dinic	10
4.3	Min Cost Matching	11
4.4	Min Cost Max Flow	12
4.5	Min Cut	12
<b>5</b>	<b>5 Geometry</b>	<b>13</b>
5.1	Convex Hull	13
5.2	Delaunay	13
5.3	Geometry	14
<b>6</b>	<b>6 Numerics</b>	<b>16</b>
6.1	Euclid	16
6.2	FFT	18
6.3	Gauss-Jordan	18
6.4	Matrix	19
6.5	Primes	20
6.6	Reduced Row Echelon Form	20
6.7	Simplex	20
<b>7</b>	<b>7 String</b>	<b>21</b>
7.1	Aho-Corasick	21
7.2	KMP	22
7.3	Suffix Arrays	22

<b>8</b>	<b>8 Misc</b>	<b>23</b>
8.1	IO	23
8.2	Longest Increasing Subsequence	23
8.3	Regular Expressions - Java	23
8.4	Tokenizer	24
8.5	Z Reference	24

## 1 1 Templates

### 1.1 Start

---

```

1 // .vimrc
2 syn on
3 set mouse=a sw=4 ts=4 ai si nu wrap
4 nnoremap ; :
5
6 // Terminal: comparing generated output to sample output
7 ./my_program < sample.in | diff sample.out -

```

---

### 1.2 Template - C++

---

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 typedef long long ll;
4 static bool DBG = 1;
5
6 ll mod(ll a, ll b) { return ((a%b)+b)%b; }
7
8 int main() {
9     ios_base::sync_with_stdio(0);
10    cout << fixed << setprecision(15);
11    int n;
12    cin >> n;
13    cout << n << endl;
14    return 0;
15 }

```

---

### 1.3 Template - Java

---

```

1 import java.util.*;
2 import java.math.*;
3 import java.io.*;
4
5 class modelo {
6     static final double EPS = 1.e-10;
7     static final boolean DBG = true;
8
9     private static int cmp(double x, double y = 0, double tol = EPS) {
10         return (x <= y + tol)? (x + tol < y)? -1 : 0 : 1;
11     }
12
13     public static void main(String[] argv) {
14         Scanner s = new Scanner(System.in);
15     }
16 }

```

---

## 2 Data Structures

### 2.1 AVL Tree

```

1 struct Node {
2     Node *l, *r; int h, size, key;
3     Node(int k) : l(0), r(0), h(1), size(1), key(k) {}
4     void u() { h=1+max(l?l->h:0, r?r->h:0);
5         size=(l?l->size:0)+1+(r?r->size:0); }
6 };
7 Node *rotl(Node *x) { Node *y=x->r; x->r=y->l; y->l=x; x->u(); y->u();
8     return y; }
9 Node *rotr(Node *x) { Node *y=x->l; x->l=y->r; y->r=x; x->u(); y->u();
10    return y; }
11 Node *rebalance(Node *x) {
12     x->u();
13     if (x->l->h > 1 + x->r->h) {
14         if (x->l->l->h < x->l->r->h) x->l = rotl(x->l);
15         x = rotr(x);
16     } else if (x->r->h > 1 + x->l->h) {
17         if (x->r->r->h < x->r->l->h) x->r = rotr(x->r);
18         x = rotl(x);
19     }
20     return x;
21 }
22 Node *insert(Node *x, int key) {
23     if (x == NULL) return new Node(key);
24     if (key < x->key) x->l = insert(x->l, key); else x->r = insert(x->r,
25         key);
26     return rebalance(x);
27 }

```

### 2.2 BIT

```

1 template<typename T> struct BIT{
2     int S;
3     vector<T> v;
4
5     BIT<T>(int _S){
6         S = _S;
7         v.resize(S+1);
8     }
9     void update(int i, T k){
10         for(i++; i<=S; i+=i&-i)
11             v[i] = v[i] + k;
12     }
13     T read(int i){
14         T sum = 0;
15         for(i++; i; i-=i&-i)
16             sum = sum + v[i];
17         return sum;
18     }
19     T read(int l, int r){
20         return read(r) - read(l-1);
21     }
22 };

```

### 2.3 KD Tree

```

1 // -----
2 // - constructs from n points in O(n lg^2 n) time
3 // - nearest-neighbor query in O(lg n) if points are well distributed
4 // - worst case nearest-neighbor may be linear
5 // -----
6
7 // number type for coordinates, and its maximum value
8 typedef long long ntype;
9 const ntype sentry = numeric_limits<ntype>::max();
10
11 // point structure for 2D-tree, can be extended to 3D
12 struct point {
13     ntype x, y;
14     point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
15 };
16 bool operator==(const point &a, const point &b) {
17     return a.x == b.x && a.y == b.y;
18 }
19 // sorts points on x-coordinate
20 bool on_x(const point &a, const point &b) {
21     return a.x < b.x;
22 }
23 // sorts points on y-coordinate
24 bool on_y(const point &a, const point &b) {
25     return a.y < b.y;
26 }
27 // squared distance between points
28 ntype pdist2(const point &a, const point &b) {
29     ntype dx = a.x-b.x, dy = a.y-b.y;
30     return dx*dx + dy*dy;
31 }
32 // bounding box for a set of points
33 struct bbox {
34     ntype x0, x1, y0, y1;
35     bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}
36     // computes bounding box from a bunch of points
37     void compute(const vector<point> &v) {
38         for (int i = 0; i < v.size(); ++i) {
39             x0 = min(x0, v[i].x); x1 = max(x1, v[i].x);
40             y0 = min(y0, v[i].y); y1 = max(y1, v[i].y);
41         }
42     }
43     // squared distance between a point and this bbox, 0 if inside
44     ntype distance(const point &p) {
45         if (p.x < x0) {
46             if (p.y < y0) return pdist2(point(x0, y0), p);
47             else if (p.y > y1) return pdist2(point(x0, y1), p);
48             else return pdist2(point(x0, p.y), p);
49         }
50         else if (p.x > x1) {
51             if (p.y < y0) return pdist2(point(x1, y0), p);
52             else if (p.y > y1) return pdist2(point(x1, y1), p);
53             else return pdist2(point(x1, p.y), p);
54         }
55         else {
56             if (p.y < y0) return pdist2(point(p.x, y0), p);
57             else if (p.y > y1) return pdist2(point(p.x, y1), p);

```

```

58     } else return 0;
59 }
60 }
61 };
62
63 // stores a single node of the kd-tree, either internal or leaf
64 struct kdnode {
65     bool leaf; // true if this is a leaf node (has one point)
66     point pt; // the single point of this is a leaf
67     bbox bound; // bounding box for set of points in children
68     kdnode *first, *second; // two children of this kd-node
69     kdnode() : leaf(false), first(0), second(0) {}
70     ~kdnode() { if (first) delete first; if (second) delete second; }
71
72     // intersect a point with this node (returns squared distance)
73     ntype intersect(const point &p) {
74         return bound.distance(p);
75     }
76     // recursively builds a kd-tree from a given cloud of points
77     void construct(vector<point> &vp) {
78         // compute bounding box for points at this node
79         bound.compute(vp);
80         // if we're down to one point, then we're a leaf node
81         if (vp.size() == 1) {
82             leaf = true;
83             pt = vp[0];
84         } else {
85             // split on x if the bbox is wider than high (not best
86             // heuristic...)
87             if (bound.x1-bound.x0 >= bound.y1-bound.y0)
88                 sort(vp.begin(), vp.end(), on_x);
89             // otherwise split on y-coordinate
90             else sort(vp.begin(), vp.end(), on_y);
91             // divide by taking half the array for each child
92             // (not best performance if many duplicates in the middle)
93             int half = vp.size()/2;
94             vector<point> vl(vp.begin(), vp.begin()+half);
95             vector<point> vr(vp.begin()+half, vp.end());
96             first = new kdnode(); first->construct(vl);
97             second = new kdnode(); second->construct(vr);
98         }
99     }
100 };
101
102 // simple kd-tree class to hold the tree and handle queries
103 struct kdtree {
104     kdnode *root;
105     // constructs a kd-tree from a points (copied here, as it sorts them)
106     kdtree(const vector<point> &vp) {
107         vector<point> v(vp.begin(), vp.end());
108         root = new kdnode();
109         root->construct(v);
110     }
111     ~kdtree() { delete root; }
112     // recursive search method returns squared distance to nearest point
113     ntype search(kdnode *node, const point &p) {
114         if (node->leaf) {
115             // commented special case tells a point not to find itself
116             // if (p == node->pt) return sentry;

```

```

116         // else
117         return pdist2(p, node->pt);
118     }
119     ntype bfirst = node->first->intersect(p);
120     ntype bsecond = node->second->intersect(p);
121     // choose the side with the closest bounding box to search first
122     // (note that the other side is also searched if needed)
123     if (bfirst < bsecond) {
124         ntype best = search(node->first, p);
125         if (bsecond < best)
126             best = min(best, search(node->second, p));
127         return best;
128     }
129     else {
130         ntype best = search(node->second, p);
131         if (bfirst < best)
132             best = min(best, search(node->first, p));
133         return best;
134     }
135 }
136 // squared distance to the nearest
137 ntype nearest(const point &p) {
138     return search(root, p);
139 }
140 };
141
142 // some basic test code here
143 int main() {
144     // generate some random points for a kd-tree
145     vector<point> vp;
146     for (int i = 0; i < 100000; ++i) {
147         vp.push_back(point(rand()%100000, rand()%100000));
148     }
149     kdtree tree(vp);
150     // query some points
151     for (int i = 0; i < 10; ++i) {
152         point q(rand()%100000, rand()%100000);
153         cout << "Closest squared distance to (" << q.x << ", " << q.y <<
154             "<< " is " << tree.nearest(q) << endl;
155     }
156 }

```

## 2.4 LCA

```

1 struct lca {
2     int L, N;
3     vector<int> depth, size, link;
4
5     lca(){}
6     lca(const vvi &graph, int root = 0) {
7         N = graph.size();
8         for (L = 0; (1 << L) <= N; L++);
9         depth.resize(N);
10        size.resize(N);
11        link.resize(L*N);
12        init(root, root, graph);
13    }

```

```

14 void init(int loc, int par, const vvi &graph) {
15     link[loc] = par;
16     for (int l = 1; l < L; l++)
17         link[l*N + loc] = link[(l-1)*N + link[(l-1)*N + loc]];
18     for (int nbr : graph[loc]) {
19         if (nbr == par) continue;
20         depth[nbr] = depth[loc] + 1;
21         init(nbr, loc, graph);
22         size[loc] += size[nbr];
23     }
24     size[loc]++;
25 }
26 int above(int loc, int dist) {
27     for (int l = 0; l < L; l++)
28         if ((dist >> l)&1)
29             loc = link[l*N + loc];
30     return loc;
31 }
32 int find(int u, int v) {
33     if (depth[u] > depth[v]) swap(u, v);
34     v = above(v, depth[v] - depth[u]);
35     if (u == v) return u;
36     for (int l = L - 1; l >= 0; l--) {
37         if (link[l*N + u] != link[l*N + v])
38             u = link[l*N + u], v = link[l*N + v];
39     }
40     return link[u];
41 }
42 };

```

## 2.5 Lazy Segment Tree

```

1 // Modular lazy segment tree
2 // It takes a type T for vertex values and a type U for update
3 // operations. Type T should have an operator '+' specifying how to
4 // combine vertices. Type U should have an operator '()' specifying how
5 // to apply updates to vertices and an operator '+' for combining two
6 // updates. Example below.
7 template<typename T, typename U> struct seg_tree_lazy {
8     int S, H;
9     T zero;
10    vector<T> value;
11    U noop;
12    vector<bool> dirty;
13    vector<U> prop;
14
15    seg_tree_lazy<T, U>(int _S, T _zero = T(), U _noop = U()) {
16        zero = _zero, noop = _noop;
17        for (S = 1, H = 1; S < _S; ) S *= 2, H++;
18        value.resize(2*S, zero);
19        dirty.resize(2*S, false);
20        prop.resize(2*S, noop);
21    }
22    void set_leaves(vector<T> &leaves) {
23        copy(leaves.begin(), leaves.end(), value.begin() + S);
24        for (int i = S - 1; i > 0; i--)
25            value[i] = value[2 * i] + value[2 * i + 1];
26    }

```

```

27 void apply(int i, U &update) {
28     value[i] = update(value[i]);
29     if (i < S) {
30         prop[i] = prop[i] + update;
31         dirty[i] = true;
32     }
33 }
34 void rebuild(int i) {
35     for (int l = i/2; l; l /= 2) {
36         T combined = value[2*l] + value[2*l+1];
37         value[l] = prop[l](combined);
38     }
39 }
40 void propagate(int i) {
41     for (int h = H; h > 0; h--) {
42         int l = i >> h;
43         if (dirty[l]) {
44             apply(2*l, prop[l]);
45             apply(2*l+1, prop[l]);
46             prop[l] = noop;
47             dirty[l] = false;
48         }
49     }
50 }
51 void upd(int i, int j, U update) {
52     i += S, j += S;
53     propagate(i), propagate(j);
54     for (int l = i, r = j; l <= r; l /= 2, r /= 2) {
55         if ((l&1) == 1) apply(l++, update);
56         if ((r&1) == 0) apply(r--, update);
57     }
58     rebuild(i), rebuild(j);
59 }
60 T query(int i, int j){
61     i += S, j += S;
62     propagate(i), propagate(j);
63     T res_left = zero, res_right = zero;
64     for (; i <= j; i /= 2, j /= 2){
65         if ((i&1) == 1) res_left = res_left + value[i++];
66         if ((j&1) == 0) res_right = value[j--] + res_right;
67     }
68     return res_left + res_right;
69 }
70 };
71
72 // Example that supports following operations:
73 // 1: Add amount V to the values in range [L,R].
74 // 2: Reset the values in range [L,R] to value V.
75 // 3: Query for the sum of the values in range [L,R].
76
77 // Here's what T looks like:
78 struct node {
79     int sum, width;
80     node operator+(const node &n) {
81         return { sum + n.sum, width + n.width }
82     }
83 };
84 // Here's what U looks like:
85 struct update {

```

```

86 bool type; // 0 for add, 1 for reset
87 int value;
88 node operator()(const node &n) {
89     if(type) return { n.width * value, n.width };
90     else return { n.sum + n.width * value, n.width };
91 }
92 update operator+(const update &u) {
93     if(u.type) return u;
94     return { type, value + u.value };
95 }
96 };
97 int main() {
98     int N = 100;
99     node zero = {0,0};
100     update noop = {false, 0};
101     vector<node> leaves(N, {0,1});
102     seg_tree_lazy<node, update> st(N, zero, noop);
103     st.set_leaves(leaves);
104 }

```

## 2.6 Segment Tree 2D

```

1 #define Max 506
2 #define INF (1 << 30)
3 int P[Max][Max]; // container for 2D grid
4
5 /* 2D Segment Tree node */
6 struct Point {
7     int x, y, mx;
8     Point() {}
9     Point(int x, int y, int mx) : x(x), y(y), mx(mx) {}
10
11     bool operator < (const Point& other) const {
12         return mx < other.mx;
13     }
14 };
15
16 struct Segtree2d {
17     Point T[2 * Max * Max];
18     int n, m;
19     // initialize and construct segment tree
20     void init(int n, int m) {
21         this->n = n;
22         this->m = m;
23         build(1, 1, 1, n, m);
24     }
25     // build a 2D segment tree from data [ (a1, b1), (a2, b2) ]
26     // Time: O(n logn)
27     Point build(int node, int a1, int b1, int a2, int b2) {
28         // out of range
29         if (a1 > a2 or b1 > b2)
30             return def();
31
32         // if it is only a single index, assign value to node
33         if (a1 == a2 and b1 == b2)
34             return T[node] = Point(a1, b1, P[a1][b1]);
35
36         // split the tree into four segments

```

```

37     T[node] = def();
38     T[node] = maxNode(T[node], build(4 * node - 2, a1, b1, (a1 + a2) /
39         2, (b1 + b2) / 2 ));
40     T[node] = maxNode(T[node], build(4 * node - 1, (a1 + a2) / 2 + 1,
41         b1, a2, (b1 + b2) / 2 ));
42     T[node] = maxNode(T[node], build(4 * node + 0, a1, (b1 + b2) / 2 +
43         1, (a1 + a2) / 2, b2 ));
44     T[node] = maxNode(T[node], build(4 * node + 1, (a1 + a2) / 2 + 1,
45         (b1 + b2) / 2 + 1, a2, b2 ));
46     return T[node];
47 }
48 // helper function for query(int, int, int, int);
49 Point query(int node, int a1, int b1, int a2, int b2, int x1, int y1,
50     int x2, int y2) {
51     // if we out of range, return dummy
52     if (x1 > a2 or y1 > b2 or x2 < a1 or y2 < b1 or a1 > a2 or b1 > b2)
53         return def();
54     // if it is within range, return the node
55     if (x1 <= a1 and y1 <= b1 and a2 <= x2 and b2 <= y2)
56         return T[node];
57     // split into four segments
58     Point mx = def();
59     mx = maxNode(mx, query(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 +
60         b2) / 2, x1, y1, x2, y2 ));
61     mx = maxNode(mx, query(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2,
62         (b1 + b2) / 2, x1, y1, x2, y2 ));
63     mx = maxNode(mx, query(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 +
64         a2) / 2, b2, x1, y1, x2, y2 ));
65     mx = maxNode(mx, query(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2)
66         / 2 + 1, a2, b2, x1, y1, x2, y2));
67     // return the maximum value
68     return mx;
69 }
70 // query from range [ (x1, y1), (x2, y2) ]
71 // Time: O(logn)
72 Point query(int x1, int y1, int x2, int y2) {
73     return query(1, 1, 1, n, m, x1, y1, x2, y2);
74 }
75
76 // helper function for update(int, int, int);
77 Point update(int node, int a1, int b1, int a2, int b2, int x, int y,
78     int value) {
79     if (a1 > a2 or b1 > b2)
80         return def();
81     if (x > a2 or y > b2 or x < a1 or y < b1)
82         return T[node];
83     if (x == a1 and y == b1 and x == a2 and y == b2)
84         return T[node] = Point(x, y, value);
85     Point mx = def();
86     mx = maxNode(mx, update(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 +
87         b2) / 2, x, y, value ));
88     mx = maxNode(mx, update(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2,
89         (b1 + b2) / 2, x, y, value));
90     mx = maxNode(mx, update(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 +
91         a2) / 2, b2, x, y, value));
92     mx = maxNode(mx, update(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2)
93         / 2 + 1, a2, b2, x, y, value ));
94     return T[node] = mx;
95 }

```

```

82 // update the value of (x, y) index to 'value'
83 // Time: O(logn)
84 Point update(int x, int y, int value) {
85     return update(1, 1, 1, n, m, x, y, value);
86 }
87 // utility functions; these functions are virtual because they will
88 // be overridden in child class
89 virtual Point maxNode(Point a, Point b) {
90     return max(a, b);
91 }
92 // dummy node
93 virtual Point def() {
94     return Point(0, 0, -INF);
95 }
96 };
97 /* 2D Segment Tree for range minimum query; a override of Segtree2d class
98 */
99 struct Segtree2dMin : Segtree2d {
100     // overload maxNode() function to return minimum value
101     Point maxNode(Point a, Point b) { return min(a, b); }
102     Point def() { return Point(0, 0, INF); }
103 };
104 // initialize class objects
105 Segtree2d Tmax;
106 Segtree2dMin Tmin;
107
108 /* Drier program */
109 int main(void) {
110     int n, m;
111     // input
112     scanf("%d %d", &n, &m);
113     for(int i = 1; i <= n; i++)
114         for(int j = 1; j <= m; j++)
115             scanf("%d", &P[i][j]);
116     // initialize
117     Tmax.init(n, m);
118     Tmin.init(n, m);
119     // query
120     int x1, y1, x2, y2;
121     scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
122     Tmax.query(x1, y1, x2, y2).mx;
123     Tmin.query(x1, y1, x2, y2).mx;
124     // update
125     int x, y, v;
126     scanf("%d %d %d", &x, &y, &v);
127     Tmax.update(x, y, v);
128     Tmin.update(x, y, v);
129     return 0;
130 }

```

## 2.7 Segment Tree

```

1 template<typename T> struct seg_tree {
2     int S;
3     T zero;
4     vector<T> value;

```

```

5
6     seg_tree<T>(int _S, T _zero = T()) {
7         S = _S, zero = _zero;
8         value.resize(2*S+1, zero);
9     }
10    void set_leaves(vector<T> &leaves) {
11        copy(leaves.begin(), leaves.end(), value.begin() + S);
12        for (int i = S - 1; i > 0; i--)
13            value[i] = value[2 * i] + value[2 * i + 1];
14    }
15    void upd(int i, T v) {
16        i += S;
17        value[i] = v;
18        while(i>1){
19            i/=2;
20            value[i] = value[2*i] + value[2*i+1];
21        }
22    }
23    T query(int i, int j) {
24        T res_left = zero, res_right = zero;
25        for(i += S, j += S; i <= j; i /= 2, j /= 2){
26            if((i&1) == 1) res_left = res_left + value[i++];
27            if((j&1) == 0) res_right = value[j--] + res_right;
28        }
29        return res_left + res_right;
30    }
31 };

```

## 2.8 Union Find

```

1 // (struct) also keeps track of sizes
2 struct union_find {
3     vector<int> P,S;
4
5     union_find(int N) {
6         P.resize(N), S.resize(N, 1);
7         for(int i = 0; i < N; i++) P[i] = i;
8     }
9     int rep(int i) {return (P[i] == i) ? i : P[i] = rep(P[i]);}
10    bool unio(int a, int b) {
11        a = rep(a), b = rep(b);
12        if(a == b) return false;
13        P[b] = a;
14        S[a] += S[b];
15        return true;
16    }
17 };
18 // (Shorter) union-find set: the vector/array contains the parent of each
19 // node
20 int find(vector<int> &C, int x){return (C[x]==x) ? x : C[x]=find(C,
21     C[x]);} //C++
22 int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);} //C

```

## 3 3 Graph

### 3.1 2-SAT

```

1 struct two_sat {
2     int N;
3     vector<vector<int>>> impl;
4
5     two_sat(int _N) {
6         N = _N;
7         impl.resize(2 * N);
8     }
9     void add_impl(int var1, bool neg1, int var2, bool neg2) {
10         impl[2 * var1 + neg1].push_back(2 * var2 + neg2);
11         impl[2 * var2 + !neg2].push_back(2 * var1 + !neg1);
12     }
13     void add_clause(int var1, bool neg1, int var2, bool neg2) {
14         add_impl(var1, !neg1, var2, neg2);
15     }
16     void add_clause(int var1, bool neg1) {
17         add_clause(var1, neg1, var1, neg1);
18     }
19
20     int V, L, C;
21     stack<int> view;
22
23     int dfs(int loc) {
24         visit[loc] = V;
25         label[loc] = L++;
26         int low = label[loc];
27         view.push(loc);
28         in_view[loc] = true;
29         for (int nbr : impl[loc]) {
30             if(!visit[nbr]) low = min(low, dfs(nbr));
31             else if(in_view[nbr]) low = min(low, label[nbr]);
32         }
33         if(low == label[loc]) {
34             while (true) {
35                 int mem = view.top();
36                 comp[mem] = C;
37                 in_view[mem] = false;
38                 view.pop();
39                 if(mem == loc) break;
40             }
41             C++;
42         }
43         return low;
44     }
45
46     vector<int> visit, label, comp, in_view;
47
48     void reset(vector<int> &v) {
49         v.resize(2 * N);
50         fill(v.begin(), v.end(), 0);
51     }
52     bool consistent() {
53         V = 0, L = 0, C = 0;
54         reset(visit), reset(label), reset(comp), reset(in_view);
55         for (int i = 0; i < 2 * N; i++) {

```

```

56             if(!visit[i]) {
57                 V++;
58                 dfs(i);
59             }
60         }
61         for (int i = 0; i < N; i++)
62             if(comp[2 * i] == comp[2 * i + 1]) return false;
63         return true;
64     }
65 };

```

### 3.2 Articulation and Bridge

```

1 // Not well-tested. Use at your own risk.
2 typedef pair<int,int> ii;
3 typedef vector<int> vi;
4 typedef vector<ii> vii;
5 typedef vector<vii> vvii;
6
7 struct ArticulationAndBridge {
8     int N, root, root_children, dfs_ctr;
9     vi dfs_low, dfs_num, dfs_parent, artic;
10    vii bridge;
11    vvii adj_list;
12
13    ArticulationAndBridge (int N) : N(N), dfs_num(N, -1), dfs_low(N, 0),
14        dfs_parent(N, 0), artic(N, 0), adj_list(N) {}
15    void add_edge(int u, int v) {
16        adj_list[u].push_back({v, 0});
17        adj_list[v].push_back({u, 0});
18    }
19    void apab(int u) {
20        dfs_low[u] = dfs_num[u] = dfs_ctr++; // dfs_low[u] <=
21        dfs_num[u]
22        for(auto v : adj_list[u]) {
23            if(dfs_num[v.first] == -1) { // tree edge
24                dfs_parent[v.first] = u;
25                if(u == root) root_children++; // special case u is root
26                apab(v.first);
27                if(dfs_low[v.first] >= dfs_num[u]) // articulation point
28                    artic[u] = true;
29                if(dfs_low[v.first] > dfs_num[u]) // bridge
30                    bridge.push_back(make_pair(u, v.first));
31                dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
32            } else if (v.first != dfs_parent[u]) // back-edge
33                dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
34        }
35    }
36    void run() {
37        dfs_ctr = 0;
38        for(int i = 0; i < N; i++) {
39            if(dfs_num[i] == -1) {
40                root = i; root_children = 0; apab(i);
41                artic[root] = (root_children > 1);
42            }
43        }
44    };

```



### 3.3 Eulerian Path

```

1 struct Edge;
2 typedef list<Edge>::iterator iter;
3
4 struct Edge {
5     int next_vertex;
6     iter reverse_edge;
7     Edge(int next_vertex) : next_vertex(next_vertex) {}
8 };
9
10 const int max_vertices = 100;
11 int num_vertices;
12 list<Edge> adj[max_vertices]; // adjacency list
13 vector<int> path;
14
15 void find_path(int v) {
16     while(adj[v].size() > 0) {
17         int vn = adj[v].front().next_vertex;
18         adj[vn].erase(adj[v].front().reverse_edge);
19         adj[v].pop_front();
20         find_path(vn);
21     }
22     path.push_back(v);
23 }
24 void add_edge(int a, int b) {
25     adj[a].push_front(Edge(b));
26     iter ita = adj[a].begin();
27     adj[b].push_front(Edge(a));
28     iter itb = adj[b].begin();
29     ita->reverse_edge = itb;
30     itb->reverse_edge = ita;
31 }

```

### 3.4 Heavy Light

```

1 template<typename T> struct heavy_light {
2     lca links;
3     seg_tree<T> st;
4     vector<int> preorder, index, jump;
5
6     heavy_light(const vvi &graph, int root) {
7         links = lca(graph, 0);
8         st = seg_tree<T>(graph.size());
9         index.resize(graph.size()), jump.resize(graph.size());
10        dfs(root, root, root, graph);
11    }
12    void dfs(int loc, int par, int lhv, const vvi &graph) {
13        jump[loc] = lhv;
14        index[loc] = preorder.size();
15        preorder.push_back(loc);
16        vector<int> ch = graph[loc];
17        sort(ch.begin(), ch.end(), [&](int i, int j) {
18            return links.size[i] > links.size[j]; });
19        if (loc != par) ch.erase(ch.begin());
20        for (int c = 0; c < ch.size(); c++)
21            dfs(ch[c], loc, c ? ch[c] : lhv, graph);
22    }

```

```

23 void assign(int loc, T value) {
24     st.upd(index[loc], value);
25 }
26 T __sum(int u, int r) {
27     T res;
28     while (u != r) {
29         int go = max(index[r] + 1, index[jump[u]]);
30         res = res + st.query(go, index[u]);
31         u = links.link[preorder[go]];
32     }
33     return res;
34 }
35 T sum(int u, int v) {
36     int r = links.find(u, v);
37     return st.query(index[r], index[r]) + __sum(u, r) + __sum(v, r);
38 }
39 };

```

### 3.5 Poset Width

```

1 // requires bipartite graph (4.1)
2 vector<int> width(vector<vector<int>> poset) {
3     int N = poset.size();
4     bipartite_graph g(N, N);
5     for (int i = 0; i < N; i++) {
6         for (int j : poset[i])
7             g.edge(j, i);
8     }
9     g.matching();
10    vector<bool> vis[2];
11    vis[false].resize(2 * N, false);
12    vis[true].resize(2 * N, false);
13    for (int i = 0; i < N; i++) {
14        if (g.match[i] != -1) continue;
15        if (vis[false][i]) continue;
16        queue<pair<bool, int>> bfs;
17        bfs.push(make_pair(false, i));
18        vis[false][i] = true;
19        while (!bfs.empty()) {
20            bool inn = bfs.front().first;
21            int loc = bfs.front().second;
22            bfs.pop();
23            for (int nbr : g.adj[loc]) {
24                if (vis[!inn][nbr]) continue;
25                if ((g.match[loc] == nbr) ^ inn) continue;
26                vis[!inn][nbr] = true;
27                bfs.push(make_pair(!inn, nbr));
28            }
29        }
30    }
31    vector<bool> inz(2 * N, false);
32    for (int i = 0; i < 2 * N; i++)
33        inz[i] = vis[true][i] || vis[false][i];
34    vector<bool> ink(N, false);
35    for (int i = 0; i < N; i++)
36        if (!inz[i])
37            ink[i] = true;
38    for (int i = N; i < 2 * N; i++)

```



```

39     if (inz[i])
40         ink[i - N] = true;
41     vector<int> res;
42     for (int i = 0; i < N; i++) {
43         if (!ink[i])
44             res.push_back(i);
45     }
46     return res;
47 }

```

### 3.6 SCC

```

1 struct edge{int e, nxt;};
2 int V, E;
3 edge e[MAXE], er[MAXE];
4 int sp[MAXV], spr[MAXV];
5 int group_cnt, group_num[MAXV];
6 bool v[MAXV];
7 int stk[MAXV];
8 void fill_forward(int x) {
9     int i;
10    v[x]=true;
11    for(i=sp[x];i; i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
12    stk[++stk[0]]=x;
13 }
14 void fill_backward(int x) {
15     int i;
16     v[x]=false;
17     group_num[x]=group_cnt;
18     for(i=spr[x];i; i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
19 }
20 void add_edge(int v1, int v2) { //add edge v1->v2
21     e[++E].e=v2; e[E].nxt=sp[v1]; sp[v1]=E;
22     er[E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
23 }
24 void SCC() {
25     int i;
26     stk[0]=0;
27     memset(v, false, sizeof(v));
28     for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
29     group_cnt=0;
30     for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++;
31         fill_backward(stk[i]);}

```

### 3.7 Topological Sort

```

1 // This function uses performs a non-recursive topological sort.
2 //
3 // Running time:  $O(|V|^2)$ . If you use adjacency lists (vector<map<int> >),
4 // the running time is reduced to  $O(|E|)$ .
5 //
6 // INPUT: w[i][j] = 1 if i should come before j, 0 otherwise
7 // OUTPUT: a permutation of 0,...,n-1 (stored in a vector)
8 // which represents an ordering of the nodes which
9 // is consistent with w
10 //

```

```

11 // If no ordering is possible, false is returned.
12
13 typedef double TYPE;
14 typedef vector<TYPE> VT;
15 typedef vector<VT> VVT;
16 typedef vector<int> VI;
17 typedef vector<VI> VVI;
18
19 bool TopologicalSort (const VVI &w, VI &order){
20     int n = w.size();
21     VI parents (n);
22     queue<int> q;
23     order.clear();
24     for (int i = 0; i < n; i++){
25         for (int j = 0; j < n; j++)
26             if (w[j][i]) parents[i]++;
27         if (parents[i] == 0) q.push (i);
28     }
29     while (q.size() > 0){
30         int i = q.front();
31         q.pop();
32         order.push_back (i);
33         for (int j = 0; j < n; j++) if (w[i][j]){
34             parents[j]--;
35             if (parents[j] == 0) q.push (j);
36         }
37     }
38     return (order.size() == n);
39 }

```

## 4 Combinatorial Optimization

### 4.1 Bipartite Graph

```

1 struct bipartite_graph {
2     int A, B;
3     vector<vector<int>> adj;
4
5     bipartite_graph(int _A, int _B) {
6         A = _A, B = _B;
7         adj.resize(A + B);
8     }
9     void edge(int i, int j) {
10         adj[i].push_back(A+j);
11         adj[A+j].push_back(i);
12     }
13
14     vector<int> visit, match;
15
16     bool augment(int loc, int run) {
17         if(visit[loc] == run) return false;
18         visit[loc] = run;
19         for (int nbr : adj[loc]) {
20             if (match[nbr] == -1 || augment(match[nbr], run)) {
21                 match[loc] = nbr, match[nbr] = loc;
22                 return true;
23             }

```

```

24     }
25     return false;
26 }
27 int matching() {
28     visit = vector<int>(A+B, -1);
29     match = vector<int>(A+B, -1);
30     int ans = 0;
31     for (int i = 0; i < A; i++)
32         ans += augment(i, i);
33     return ans;
34 }
35 vector<bool> vertex_cover() {
36     vector<bool> res(A + B, false);
37     queue<int> bfs;
38     for (int i = 0; i < A; i++) {
39         if (match[i] == -1) bfs.push(i);
40         else res[i] = true;
41     }
42     while (!bfs.empty()) {
43         int loc = bfs.front();
44         bfs.pop();
45         for (int nbr : adj[loc]) {
46             if (res[nbr]) continue;
47             res[nbr] = true;
48             int loc2 = match[nbr];
49             if (loc2 == -1) continue;
50             res[loc2] = false;
51             bfs.push(loc2);
52         }
53     }
54     return res;
55 }
56 };

```

## 4.2 Max Flow - Dinic

```

1 // Adjacency list implementation of Dinic's blocking flow algorithm.
2 // This is very fast in practice, and only loses to push-relabel flow.
3 //
4 // Running time:
5 //  $O(|V|^2 |E|)$ 
6 //
7 // INPUT:
8 // - graph, constructed using AddEdge()
9 // - source and sink
10 //
11 // OUTPUT:
12 // - maximum flow value
13 // - To obtain actual flow values, look at edges with capacity > 0
14 // (zero capacity edges are residual edges).
15
16 struct Edge {
17     int from, to, cap, flow, index;
18     Edge(int from, int to, int cap, int flow, int index) :
19         from(from), to(to), cap(cap), flow(flow), index(index) {}
20     ll rcap() { return cap - flow; }
21 };
22

```

```

23 struct Dinic {
24     int N;
25     vector<vector<Edge>> G;
26     vector<vector<Edge*>> Lf;
27     vector<int> layer;
28     vector<int> Q;
29
30     Dinic(int N) : N(N), G(N), Q(N) {}
31
32     void AddEdge(int from, int to, int cap) {
33         if (from == to) return;
34         G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
35         G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
36     }
37
38     ll BlockingFlow(int s, int t) {
39         layer.clear(); layer.resize(N, -1);
40         Lf.clear(); Lf.resize(N);
41         int head = 0, tail = 0;
42         Q[tail++] = s;
43         while (head < tail) {
44             int x = Q[head++];
45             for (int i = 0; i < G[x].size(); i++) {
46                 Edge &e = G[x][i]; if (e.rcap() <= 0) continue;
47                 if (layer[e.to] == -1) {
48                     layer[e.to] = layer[e.from] + 1;
49                     Q[tail++] = e.to;
50                 }
51                 if (layer[e.to] > layer[e.from]) {
52                     Lf[e.from].push_back(&e);
53                 }
54             }
55         }
56         if (layer[t] == -1) return 0;
57         ll totflow = 0;
58         vector<Edge*> P;
59         while (!Lf[s].empty()) {
60             int curr = P.empty() ? s : P.back()->to;
61             if (curr == t) { // Augment
62                 ll amt = P.front()->rcap();
63                 for (int i = 0; i < P.size(); ++i) {
64                     amt = min(amt, P[i]->rcap());
65                 }
66                 totflow += amt;
67                 for (int i = P.size() - 1; i >= 0; --i) {
68                     P[i]->flow += amt;
69                     G[P[i]->to][P[i]->index].flow -= amt;
70                     if (P[i]->rcap() <= 0) {
71                         Lf[P[i]->from].pop_back();
72                         P.resize(i);
73                     }
74                 }
75             } else if (Lf[curr].empty()) { // Retreat
76                 P.pop_back();
77                 for (int i = 0; i < N; ++i)
78                     for (int j = 0; j < Lf[i].size(); ++j)
79                         if (Lf[i][j]->to == curr)
80                             Lf[i].erase(Lf[i].begin() + j);
81             } else { // Advance

```

```

82         P.push_back(Lf[curr].back());
83     }
84 }
85 return totflow;
86 }
87 ll GetMaxFlow(int s, int t) {
88     ll totflow = 0;
89     while (ll flow = BlockingFlow(s, t))
90         totflow += flow;
91     return totflow;
92 }
93 };

```

### 4.3 Min Cost Matching

```

1  ////////////////////////////////////////////////////
2  // Min cost bipartite matching via shortest augmenting paths
3  //
4  // This is an  $O(n^3)$  implementation of a shortest augmenting path
5  // algorithm for finding min cost perfect matchings in dense
6  // graphs. In practice, it solves 1000x1000 problems in around 1
7  // second.
8  //
9  // cost[i][j] = cost for pairing left node i with right node j
10 // Lmate[i] = index of right node that left node i pairs with
11 // Rmate[j] = index of left node that right node j pairs with
12 //
13 // The values in cost[i][j] may be positive or negative. To perform
14 // maximization, simply negate the cost[][] matrix.
15 ////////////////////////////////////////////////////
16
17 using namespace std;
18
19 typedef vector<double> VD;
20 typedef vector<VD> VVD;
21 typedef vector<int> VI;
22
23 double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
24     int n = int(cost.size());
25     // construct dual feasible solution
26     VD u(n);
27     VD v(n);
28     for (int i = 0; i < n; i++) {
29         u[i] = cost[i][0];
30         for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
31     }
32     for (int j = 0; j < n; j++) {
33         v[j] = cost[0][j] - u[0];
34         for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
35     }
36     // construct primal solution satisfying complementary slackness
37     Lmate = VI(n, -1);
38     Rmate = VI(n, -1);
39     int mated = 0;
40     for (int i = 0; i < n; i++) {
41         for (int j = 0; j < n; j++) {
42             if (Rmate[j] != -1) continue;
43             if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {

```

```

44                 Lmate[i] = j;
45                 Rmate[j] = i;
46                 mated++;
47                 break;
48             }
49         }
50     }
51     VD dist(n);
52     VI dad(n);
53     VI seen(n);
54     // repeat until primal solution is feasible
55     while (mated < n) {
56         // find an unmatched left node
57         int s = 0;
58         while (Lmate[s] != -1) s++;
59         // initialize Dijkstra
60         fill(dad.begin(), dad.end(), -1);
61         fill(seen.begin(), seen.end(), 0);
62         for (int k = 0; k < n; k++)
63             dist[k] = cost[s][k] - u[s] - v[k];
64         int j = 0;
65         while (true) {
66             // find closest
67             j = -1;
68             for (int k = 0; k < n; k++) {
69                 if (seen[k]) continue;
70                 if (j == -1 || dist[k] < dist[j]) j = k;
71             }
72             seen[j] = 1;
73             // termination condition
74             if (Rmate[j] == -1) break;
75             // relax neighbors
76             const int i = Rmate[j];
77             for (int k = 0; k < n; k++) {
78                 if (seen[k]) continue;
79                 const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
80                 if (dist[k] > new_dist) {
81                     dist[k] = new_dist;
82                     dad[k] = j;
83                 }
84             }
85         }
86         // update dual variables
87         for (int k = 0; k < n; k++) {
88             if (k == j || !seen[k]) continue;
89             const int i = Rmate[k];
90             v[k] += dist[k] - dist[j];
91             u[i] -= dist[k] - dist[j];
92         }
93         u[s] += dist[j];
94         // augment along path
95         while (dad[j] >= 0) {
96             const int d = dad[j];
97             Rmate[j] = Rmate[d];
98             Lmate[Rmate[j]] = j;
99             j = d;
100         }
101         Rmate[j] = s;
102         Lmate[s] = j;

```

```

103     mated++;
104 }
105 double value = 0;
106 for (int i = 0; i < n; i++)
107     value += cost[i][Lmate[i]];
108 return value;
109 }

```

## 4.4 Min Cost Max Flow

```

1 // Min cost max flow algorithm using an adjacency matrix. If you
2 // want just regular max flow, setting all edge costs to 1 gives
3 // running time  $O(|E|^2 |V|)$ .
4 //
5 // Running time:  $O(\min(|V|^2 * \text{totflow}, |V|^3 * \text{totcost}))$ 
6 //
7 // INPUT: cap -- a matrix such that cap[i][j] is the capacity of
8 //         a directed edge from node i to node j
9 //         cost -- a matrix such that cost[i][j] is the (positive)
10 //        cost of sending one unit of flow along a
11 //        directed edge from node i to node j
12 //         source -- starting node
13 //         sink -- ending node
14 //
15 // OUTPUT: max flow and min cost; the matrix flow will contain
16 //         the actual flow values (note that unlike in the MaxFlow
17 //         code, you don't need to ignore negative flow values -- there
18 //         shouldn't be any)
19 typedef vector<ll> vll;
20 typedef vector<vll> vvll;
21 const ll INF = 1LL << 60;
22
23 struct MCMF {
24     int N;
25     vll found, dad, dist, pi;
26     vvll cap, flow, cost;
27
28     MCMF(int N) : N(N), cap(N, vll(N)), flow(cap), cost(cap),
29     dad(N), found(N), pi(N), dist(N+1) {};
30
31     void add_edge(int from, int to, ll ca, ll co) {
32         cap[from][to] = ca; cost[from][to] = co; }
33     bool search(int source, int sink) {
34         fill(found.begin(), found.end(), 0);
35         fill(dist.begin(), dist.end(), INF);
36         dist[source] = 0;
37         while(source != N) {
38             int best = N;
39             found[source] = 1;
40             for(int k = 0; k < N; k++) {
41                 if(found[k]) continue;
42                 if(flow[k][source]) {
43                     ll val = dist[source] + pi[source] - pi[k] -
44                         cost[k][source];
45                     if(dist[k] > val) {
46                         dist[k] = val;
47                         dad[k] = source;
48                     }
49                 }
50             }
51             source = best;
52         }
53         return found[sink];
54     }
55     pair<ll, ll> mcmf(int source, int sink) {
56         ll totflow = 0, totcost = 0;
57         while(search(source, sink)) {
58             ll amt = INF;
59             for(int x = sink; x != source; x = dad[x])
60                 amt = min(amt, (ll)(flow[x][dad[x]] != 0 ?
61                     flow[x][dad[x]] : cap[dad[x]][x] -
62                     flow[dad[x]][x]));
63             for(int x = sink; x != source; x = dad[x]) {
64                 if(flow[x][dad[x]] != 0) {
65                     flow[x][dad[x]] -= amt;
66                     totcost -= amt * cost[x][dad[x]];
67                 } else {
68                     flow[dad[x]][x] += amt;
69                     totcost += amt * cost[dad[x]][x];
70                 }
71             }
72             totflow += amt;
73         }
74         return {totflow, totcost};
75     }
76 };

```

```

48     }
49     if(flow[source][k] < cap[source][k]) {
50         ll val = dist[source] + pi[source] - pi[k] +
51             cost[source][k];
52         if(dist[k] > val) {
53             dist[k] = val;
54             dad[k] = source;
55         }
56     }
57     if(dist[k] < dist[best]) best = k;
58 }
59 source = best;
60 }
61 for(int k = 0; k < N; k++)
62     pi[k] = min((ll)(pi[k] + dist[k]), INF);
63 return found[sink];
64 }
65 pair<ll, ll> mcmf(int source, int sink) {
66     ll totflow = 0, totcost = 0;
67     while(search(source, sink)) {
68         ll amt = INF;
69         for(int x = sink; x != source; x = dad[x])
70             amt = min(amt, (ll)(flow[x][dad[x]] != 0 ?
71                 flow[x][dad[x]] : cap[dad[x]][x] -
72                 flow[dad[x]][x]));
73         for(int x = sink; x != source; x = dad[x]) {
74             if(flow[x][dad[x]] != 0) {
75                 flow[x][dad[x]] -= amt;
76                 totcost -= amt * cost[x][dad[x]];
77             } else {
78                 flow[dad[x]][x] += amt;
79                 totcost += amt * cost[dad[x]][x];
80             }
81         }
82         totflow += amt;
83     }
84     return {totflow, totcost};
85 }

```

## 4.5 Min Cut

```

1 // Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
2 //
3 // Running time:
4 //  $O(|V|^3)$ 
5 //
6 // INPUT:
7 // - graph, constructed using AddEdge()
8 //
9 // OUTPUT:
10 // - (min cut value, nodes in half of min cut)
11 typedef vector<int> VI;
12 typedef vector<VI> VVI;
13 const int INF = 1000000000;
14
15 pair<int, VI> GetMinCut(VVI &weights) {
16     int N = weights.size();

```

```

17 VI used(N), cut, best_cut;
18 int best_weight = -1;
19 for (int phase = N-1; phase >= 0; phase--) {
20     VI w = weights[0];
21     VI added = used;
22     int prev, last = 0;
23     for (int i = 0; i < phase; i++) {
24         prev = last;
25         last = -1;
26         for (int j = 1; j < N; j++)
27             if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
28         if (i == phase-1) {
29             for (int j = 0; j < N; j++) weights[prev][j] +=
30                 weights[last][j];
31             for (int j = 0; j < N; j++) weights[j][prev] =
32                 weights[j][last];
33             used[last] = true;
34             cut.push_back(last);
35             if (best_weight == -1 || w[last] < best_weight) {
36                 best_cut = cut;
37                 best_weight = w[last];
38             }
39         } else {
40             for (int j = 0; j < N; j++)
41                 w[j] += weights[last][j];
42             added[last] = true;
43         }
44     }
45     return make_pair(best_weight, best_cut);
46 }

```

## 5 5 Geometry

### 5.1 Convex Hull

```

1 // Compute the 2D convex hull of a set of points using the monotone chain
2 // algorithm. Eliminate redundant points from the hull if
3 // REMOVE_REDUNDANT is
4 // #defined.
5 // Running time: O(n log n)
6 //
7 // INPUT: a vector of input points, unordered.
8 // OUTPUT: a vector of points in the convex hull, counterclockwise,
9 // starting
10 // with bottommost/leftmost point
11 #define REMOVE_REDUNDANT
12 typedef double T;
13 const T EPS = 1e-7;
14 struct PT {
15     T x, y;
16     PT() {}
17     PT(T x, T y) : x(x), y(y) {}
18     bool operator<(const PT &rhs) const { return make_pair(y,x) <
19         make_pair(rhs.y,rhs.x); }

```

```

19     bool operator==(const PT &rhs) const { return make_pair(y,x) ==
20         make_pair(rhs.y,rhs.x); }
21 };
22 T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
23 T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }
24 #ifndef REMOVE_REDUNDANT
25 bool between(const PT &a, const PT &b, const PT &c) {
26     return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 &&
27         (a.y-b.y)*(c.y-b.y) <= 0);
28 }
29 #endif
30 void ConvexHull(vector<PT> &pts) {
31     sort(pts.begin(), pts.end());
32     pts.erase(unique(pts.begin(), pts.end()), pts.end());
33     vector<PT> up, dn;
34     for (int i = 0; i < pts.size(); i++) {
35         while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i])
36             >= 0) up.pop_back();
37         while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i])
38             <= 0) dn.pop_back();
39         up.push_back(pts[i]);
40         dn.push_back(pts[i]);
41     }
42     pts = dn;
43     for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);
44 #ifndef REMOVE_REDUNDANT
45     if (pts.size() <= 2) return;
46     dn.clear();
47     dn.push_back(pts[0]);
48     dn.push_back(pts[1]);
49     for (int i = 2; i < pts.size(); i++) {
50         if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i]))
51             dn.pop_back();
52         dn.push_back(pts[i]);
53     }
54     if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
55         dn[0] = dn.back();
56         dn.pop_back();
57     }
58     pts = dn;
59 #endif
60 }

```

### 5.2 Delaunay

```

1 // Slow but simple Delaunay triangulation. Does not handle
2 // degenerate cases (from O'Rourke, Computational Geometry in C)
3 //
4 // Running time: O(n^4)
5 //
6 // INPUT: x[] = x-coordinates
7 //        y[] = y-coordinates
8 //
9 // OUTPUT: triples = a vector containing m triples of indices
10 //           corresponding to triangle vertices
11 typedef double T;
12
13 struct triple {

```

```

14     int i, j, k;
15     triple() {}
16     triple(int i, int j, int k) : i(i), j(j), k(k) {}
17 };
18 vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
19     int n = x.size();
20     vector<T> z(n);
21     vector<triple> ret;
22     for (int i = 0; i < n; i++)
23         z[i] = x[i] * x[i] + y[i] * y[i];
24     for (int i = 0; i < n-2; i++) {
25         for (int j = i+1; j < n; j++) {
26             for (int k = i+1; k < n; k++) {
27                 if (j == k) continue;
28                 double xn = (y[j]-y[i])*(z[k]-z[i]) -
29                     (y[k]-y[i])*(z[j]-z[i]);
30                 double yn = (x[k]-x[i])*(z[j]-z[i]) -
31                     (x[j]-x[i])*(z[k]-z[i]);
32                 double zn = (x[j]-x[i])*(y[k]-y[i]) -
33                     (x[k]-x[i])*(y[j]-y[i]);
34                 bool flag = zn < 0;
35                 for (int m = 0; flag && m < n; m++)
36                     flag = flag && ((x[m]-x[i])*xn +
37                         (y[m]-y[i])*yn +
38                         (z[m]-z[i])*zn <= 0);
39                 if (flag) ret.push_back(triple(i, j, k));
40             }
41         }
42     }
43     return ret;
44 }
45 int main() {
46     T xs[]={0, 0, 1, 0.9};
47     T ys[]={0, 1, 0, 0.9};
48     vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
49     vector<triple> tri = delaunayTriangulation(x, y);
50     //expected: 0 1 3
51     //          0 3 2
52     int i;
53     for(i = 0; i < tri.size(); i++)
54         printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
55     return 0;
56 }

```

### 5.3 Geometry

```

1 // C++ routines for computational geometry.
2 double INF = 1e100;
3 double EPS = 1e-12;
4
5 struct PT {
6     double x, y;
7     PT() {}
8     PT(double x, double y) : x(x), y(y) {}
9     PT(const PT &p) : x(p.x), y(p.y) {}
10    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
11    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
12    PT operator * (double c) const { return PT(x*c, y*c); }

```

```

13    PT operator / (double c) const { return PT(x/c, y/c); }
14 };
15
16 double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
17 double dist2(PT p, PT q) { return dot(p-q,p-q); }
18 double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
19 ostream &operator<<(ostream &os, const PT &p) {
20     os << "(" << p.x << "," << p.y << ")";
21 }
22 // rotate a point CCW or CW around the origin
23 PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
24 PT RotateCW90(PT p) { return PT(p.y,-p.x); }
25 PT RotateCCW(PT p, double t) {
26     return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
27 }
28 // project point c onto line through a and b
29 // assuming a != b
30 PT ProjectPointLine(PT a, PT b, PT c) {
31     return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
32 }
33 // project point c onto line segment through a and b
34 PT ProjectPointSegment(PT a, PT b, PT c) {
35     double r = dot(b-a,b-a);
36     if (fabs(r) < EPS) return a;
37     r = dot(c-a, b-a)/r;
38     if (r < 0) return a;
39     if (r > 1) return b;
40     return a + (b-a)*r;
41 }
42 // compute distance from c to segment between a and b
43 double DistancePointSegment(PT a, PT b, PT c) {
44     return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
45 }
46 // compute distance between point (x,y,z) and plane ax+by+cz=d
47 double DistancePointPlane(double x, double y, double z,
48     double a, double b, double c, double d) {
49     return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
50 }
51 // determine if lines from a to b and c to d are parallel or collinear
52 bool LinesParallel(PT a, PT b, PT c, PT d) {
53     return fabs(cross(b-a, c-d)) < EPS;
54 }
55 bool LinesCollinear(PT a, PT b, PT c, PT d) {
56     return LinesParallel(a, b, c, d)
57         && fabs(cross(a-b, a-c)) < EPS
58         && fabs(cross(c-d, c-a)) < EPS;
59 }
60 // determine if line segment from a to b intersects with
61 // line segment from c to d
62 bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
63     if (LinesCollinear(a, b, c, d)) {
64         if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
65             dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
66         if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
67             return false;
68         return true;
69     }
70     if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
71     if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;

```



```

72     return true;
73 }
74 // compute intersection of line passing through a and b
75 // with line passing through c and d, assuming that unique
76 // intersection exists; for segment intersection, check if
77 // segments intersect first
78 PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
79     b=b-a; d=c-d; c=c-a;
80     assert(dot(b, b) > EPS && dot(d, d) > EPS);
81     return a + b*cross(c, d)/cross(b, d);
82 }
83 // compute center of circle given three points
84 PT ComputeCircleCenter(PT a, PT b, PT c) {
85     b=(a+b)/2;
86     c=(a+c)/2;
87     return ComputeLineIntersection(b, b+RotateCW90(a-b), c,
88                                     c+RotateCW90(a-c));
89 }
90 // determine if point is in a possibly non-convex polygon (by William
91 // Randolph Franklin); returns 1 for strictly interior points, 0 for
92 // strictly exterior points, and 0 or 1 for the remaining points.
93 // Note that it is possible to convert this into an *exact* test using
94 // integer arithmetic by taking care of the division appropriately
95 // (making sure to deal with signs properly) and then by writing exact
96 // tests for checking point on polygon boundary
97 bool PointInPolygon(const vector<PT> &p, PT q) {
98     bool c = 0;
99     for (int i = 0; i < p.size(); i++){
100         int j = (i+1)%p.size();
101         if ((p[i].y <= q.y && q.y < p[j].y ||
102             p[j].y <= q.y && q.y < p[i].y) &&
103             q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) /
104                 (p[j].y - p[i].y))
105             c = !c;
106     }
107     return c;
108 }
109 // determine if point is on the boundary of a polygon
110 bool PointOnPolygon(const vector<PT> &p, PT q) {
111     for (int i = 0; i < p.size(); i++)
112         if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) <
113             EPS)
114             return true;
115     return false;
116 }
117 // compute intersection of line through points a and b with
118 // circle centered at c with radius r > 0
119 vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
120     vector<PT> ret;
121     b = b-a;
122     a = a-c;
123     double A = dot(b, b);
124     double B = dot(a, b);
125     double C = dot(a, a) - r*r;
126     double D = B*B - A*C;
127     if (D < -EPS) return ret;
128     ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
129     if (D > EPS)
130         ret.push_back(c+a+b*(-B-sqrt(D))/A);

```

```

128     return ret;
129 }
130 // compute intersection of circle centered at a with radius r
131 // with circle centered at b with radius R
132 vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
133     vector<PT> ret;
134     double d = sqrt(dist2(a, b));
135     if (d > r+R || d+min(r, R) < max(r, R)) return ret;
136     double x = (d*d-R*R+r*r)/(2*d);
137     double y = sqrt(r*r-x*x);
138     PT v = (b-a)/d;
139     ret.push_back(a+v*x + RotateCCW90(v)*y);
140     if (y > 0)
141         ret.push_back(a+v*x - RotateCCW90(v)*y);
142     return ret;
143 }
144 // This code computes the area or centroid of a (possibly nonconvex)
145 // polygon, assuming that the coordinates are listed in a clockwise or
146 // counterclockwise fashion. Note that the centroid is often known as
147 // the "center of gravity" or "center of mass".
148 double ComputeSignedArea(const vector<PT> &p) {
149     double area = 0;
150     for(int i = 0; i < p.size(); i++) {
151         int j = (i+1) % p.size();
152         area += p[i].x*p[j].y - p[j].x*p[i].y;
153     }
154     return area / 2.0;
155 }
156 double ComputeArea(const vector<PT> &p) {
157     return fabs(ComputeSignedArea(p));
158 }
159 PT ComputeCentroid(const vector<PT> &p) {
160     PT c(0,0);
161     double scale = 6.0 * ComputeSignedArea(p);
162     for (int i = 0; i < p.size(); i++){
163         int j = (i+1) % p.size();
164         c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
165     }
166     return c / scale;
167 }
168 // tests whether or not a given polygon (in CW or CCW order) is simple
169 bool IsSimple(const vector<PT> &p) {
170     for (int i = 0; i < p.size(); i++) {
171         for (int k = i+1; k < p.size(); k++) {
172             int j = (i+1) % p.size();
173             int l = (k+1) % p.size();
174             if (i == 1 || j == k) continue;
175             if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
176                 return false;
177         }
178     }
179     return true;
180 }
181 // Plane distance between parallel planes aX + bY + cZ + d1 = 0 and
182 // aX + bY + cZ + d2 = 0 is abs(d1 - d2) / sqrt(a*a + b*b + c*c)
183 // distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
184 // (or ray, or segment; in the case of the ray, the endpoint is the
185 // first point)
186 public static final int LINE = 0;

```



```

187 public static final int SEGMENT = 1;
188 public static final int RAY = 2;
189 public static double ptLineDistSq(double x1, double y1, double z1,
190     double x2, double y2, double z2, double px, double py, double pz,
191     int type) {
192     double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);
193     double x, y, z;
194     if (pd2 == 0) {
195         x = x1;
196         y = y1;
197         z = z1;}
198     else {
199         double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) /
200             pd2;
201         x = x1 + u * (x2 - x1);
202         y = y1 + u * (y2 - y1);
203         z = z1 + u * (z2 - z1);
204         if (type != LINE && u < 0) {
205             x = x1;
206             y = y1;
207             z = z1;}
208         if (type == SEGMENT && u > 1.0) {
209             x = x2;
210             y = y2;
211             z = z2;}
212     }
213     return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
214 }
215 int main() {
216     // expected: (-5,2)
217     cerr << RotateCCW90(PT(2,5)) << endl;
218     // expected: (5,-2)
219     cerr << RotateCW90(PT(2,5)) << endl;
220     // expected: (-5,2)
221     cerr << RotateCCW(PT(2,5),M_PI/2) << endl;
222     // expected: (5,2)
223     cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;
224     // expected: (5,2) (7.5,3) (2.5,1)
225     cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
226         << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
227         << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;
228     // expected: 6.78903
229     cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
230     // expected: 1 0 1
231     cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
232         << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
233         << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
234     // expected: 0 0 1
235     cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
236         << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
237         << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
238     // expected: 1 1 1 0
239     cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
240         << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
241         << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
242         << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;
243     // expected: (1,2)

```

```

244     cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3))
245         << endl;
246     // expected: (1,1)
247     cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;
248     vector<PT> v;
249     v.push_back(PT(0,0));
250     v.push_back(PT(5,0));
251     v.push_back(PT(5,5));
252     v.push_back(PT(0,5));
253     // expected: 1 1 1 0 0
254     cerr << PointInPolygon(v, PT(2,2)) << " "
255         << PointInPolygon(v, PT(2,0)) << " "
256         << PointInPolygon(v, PT(0,2)) << " "
257         << PointInPolygon(v, PT(5,2)) << " "
258         << PointInPolygon(v, PT(2,5)) << endl;
259     // expected: 0 1 1 1 1
260     cerr << PointOnPolygon(v, PT(2,2)) << " "
261         << PointOnPolygon(v, PT(2,0)) << " "
262         << PointOnPolygon(v, PT(0,2)) << " "
263         << PointOnPolygon(v, PT(5,2)) << " "
264         << PointOnPolygon(v, PT(2,5)) << endl;
265     // expected:(1,6)
266     // (5,4) (4,5)
267     // blank line
268     // (4,5) (5,4)
269     // blank line
270     // (4,5) (5,4)
271     vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
272     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
273     u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
274     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
275     u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
276     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
277     u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
278     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
279     u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
280     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
281     u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
282     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
283     // area should be 5.0
284     // centroid should be (1.1666666, 1.1666666)
285     PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
286     vector<PT> p(pa, pa+4);
287     PT c = ComputeCentroid(p);
288     cerr << "Area: " << ComputeArea(p) << endl;
289     cerr << "Centroid: " << c << endl;
290 }

```

## 6 Numerics

### 6.1 Euclid

```

1 // This is a collection of useful code for solving problems that
2 // involve modular linear equations. Note that all of the
3 // algorithms described here work on nonnegative integers.
4 typedef vector<int> VI;
5 typedef pair<int,int> PII;

```

```

6 // return a % b (positive value)
7 int mod(int a, int b) {
8     return ((a%b)+b)%b;
9 }
10 // computes gcd(a,b)
11 int gcd(int a, int b) {
12     int tmp;
13     while(b){a%=b; tmp=a; a=b; b=tmp;}
14     return a;
15 }
16 // computes lcm(a,b)
17 int lcm(int a, int b) {
18     return a/gcd(a,b)*b;
19 }
20 // returns d = gcd(a,b); finds x,y such that d = ax + by
21 int extended_euclid(int a, int b, int &x, int &y) {
22     int xx = y = 0;
23     int yy = x = 1;
24     while (b) {
25         int q = a/b;
26         int t = b; b = a%b; a = t;
27         t = xx; xx = x-q*xx; x = t;
28         t = yy; yy = y-q*yy; y = t;
29     }
30     return a;
31 }
32 // finds all solutions to ax = b (mod n)
33 VI modular_linear_equation_solver(int a, int b, int n) {
34     int x, y;
35     VI solutions;
36     int d = extended_euclid(a, n, x, y);
37     if (!(b%d)) {
38         x = mod(x*(b/d), n);
39         for (int i = 0; i < d; i++)
40             solutions.push_back(mod(x + i*(n/d), n));
41     }
42     return solutions;
43 }
44 // computes b such that ab = 1 (mod n), returns -1 on failure
45 int mod_inverse(int a, int n) {
46     int x, y;
47     int d = extended_euclid(a, n, x, y);
48     if (d > 1) return -1;
49     return mod(x,n);
50 }
51 // Chinese remainder theorem (special case): find z such that
52 // z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
53 // Return (z,M). On failure, M = -1.
54 PII chinese_remainder_theorem(int x, int a, int y, int b) {
55     int s, t;
56     int d = extended_euclid(x, y, s, t);
57     if (a%d != b%d) return make_pair(0, -1);
58     return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
59 }
60 // Chinese remainder theorem: find z such that
61 // z % x[i] = a[i] for all i. Note that the solution is
62 // unique modulo M = lcm_i (x[i]). Return (z,M). On
63 // failure, M = -1. Note that we do not require the a[i]'s

```

```

65 // to be relatively prime.
66 PII chinese_remainder_theorem(const VI &x, const VI &a) {
67     PII ret = make_pair(a[0], x[0]);
68     for (int i = 1; i < x.size(); i++) {
69         ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
70         if (ret.second == -1) break;
71     }
72     return ret;
73 }
74 // computes x and y such that ax + by = c; on failure, x = y = -1
75 void linear_diophantine(int a, int b, int c, int &x, int &y) {
76     int d = gcd(a,b);
77     if (c%d) {
78         x = y = -1;
79     } else {
80         x = c/d * mod_inverse(a/d, b/d);
81         y = (c-a*x)/b;
82     }
83 }
84 // computes n^k (mod m)
85 long long power(long long n, long long k, long long m = LLONG_MAX) {
86     long long ret = 1;
87     while (k) {
88         if (k & 1) ret = (ret * n) % m;
89         n = (n * n) % m;
90         k >>= 1;
91     }
92     return ret;
93 }
94 // computes nCm
95 long long binomial(int n, int m) {
96     if (n > m || n < 0) return 0;
97     long long ans = 1, ans2 = 1;
98     for (int i = 0; i < m; i++) {
99         ans *= n - i;
100        ans2 *= i + 1;
101    }
102    return ans / ans2;
103 }
104 // computes the nth Catalan number
105 long long catalan_number(int n) {
106     return binomial(n * 2, n) / (n + 1);
107 }
108 // computes phi(n) (use euler_totient)
109 long long euler_totient2(long long n, long long ps) {
110     for (long long i = ps; i * i <= n; i++) {
111         if (n % i == 0) {
112             long long p = 1;
113             while (n % i == 0) {
114                 n /= i;
115                 p *= i;
116             }
117             return (p - p / i) * euler_totient2(n, i + 1);
118         }
119         if (i > 2) i++;
120     }
121     return n - 1;
122 }
123 long long euler_totient(long long n) {

```

```

124     return euler_totient2(n, 2);
125 }
126
127 int main() {
128     // expected: 2
129     cout << gcd(14, 30) << endl;
130     // expected: 2 -2 1
131     int x, y;
132     int d = extended_euclid(14, 30, x, y);
133     cout << d << " " << x << " " << y << endl;
134     // expected: 95 45
135     VI sols = modular_linear_equation_solver(14, 30, 100);
136     for (int i = 0; i < (int) sols.size(); i++) cout << sols[i] << " ";
137     cout << endl;
138     // expected: 8
139     cout << mod_inverse(8, 9) << endl;
140     // expected: 23 56
141     //          11 12
142     int xs[] = {3, 5, 7, 4, 6};
143     int as[] = {2, 3, 2, 3, 5};
144     PII ret = chinese_remainder_theorem(VI(xs, xs+3), VI(as, as+3));
145     cout << ret.first << " " << ret.second << endl;
146     ret = chinese_remainder_theorem(VI(xs+3, xs+5), VI(as+3, as+5));
147     cout << ret.first << " " << ret.second << endl;
148     // expected: 5 -15
149     linear_diophantine(7, 2, 5, x, y);
150     cout << x << " " << y << endl;
151 }

```

## 6.2 FFT

```

1 namespace fft {
2     struct cnum {
3         double a, b;
4         cnum operator+(const cnum &c) { return { a + c.a, b + c.b }; }
5         cnum operator-(const cnum &c) { return { a - c.a, b - c.b }; }
6         cnum operator*(const cnum &c) { return { a*c.a - b*c.b, a*c.b +
7             b*c.a }; }
8         cnum operator/(double d) { return { a / d, b / d }; }
9     };
10
11     const double PI = 2 * atan2(1, 0);
12     int deg;
13     vector<int> rev;
14
15     void set_degree(int _deg) {
16         assert(__builtin_popcount(_deg) == 1);
17         deg = _deg;
18         rev.resize(deg);
19         for (int i = 1, j = 0; i < deg; i++) {
20             int bit = deg / 2;
21             for (; j >= bit; bit /= 2)
22                 j -= bit;
23             j += bit;
24             rev[i] = j;
25         }
26     }
27
28     void transform(vector<cnum> &poly, bool invert) {

```

```

27         if(deg != poly.size()) set_degree(poly.size());
28         for (int i = 1; i < deg; i++)
29             if(rev[i] > i)
30                 swap(poly[i], poly[rev[i]]);
31         for (int len = 2; len <= deg; len *= 2) {
32             double ang = 2 * PI / len * (invert ? -1 : 1);
33             cnum base = { cos(ang), sin(ang) };
34             for (int i = 0; i < deg; i += len) {
35                 cnum w = {1, 0};
36                 for (int j = 0; j < len / 2; j++) {
37                     cnum u = poly[i+j];
38                     cnum v = w * poly[i+j+len/2];
39                     poly[i+j] = u + v;
40                     poly[i+j+len/2] = u - v;
41                     w = w * base;
42                 }
43             }
44         }
45         if(invert) {
46             for (int i = 0; i < deg; i++)
47                 poly[i] = poly[i] / double(deg);
48         }
49     }
50 }

```

## 6.3 Gauss-Jordan

```

1 // Gauss-Jordan elimination with full pivoting.
2 //
3 // Uses:
4 // (1) solving systems of linear equations (AX=B)
5 // (2) inverting matrices (AX=I)
6 // (3) computing determinants of square matrices
7 //
8 // Running time: O(n^3)
9 //
10 // INPUT: a[] [] = an nxn matrix
11 //         b[] [] = an nxm matrix
12 //
13 // OUTPUT: X = an nxm matrix (stored in b[] [])
14 //          A^{-1} = an nxn matrix (stored in a[] [])
15 //          returns determinant of a[] []
16 typedef vector<int> VI;
17 typedef double T;
18 typedef vector<T> VT;
19 typedef vector<VT> VVT;
20 const double EPS = 1e-10;
21
22 T GaussJordan(VVT &a, VVT &b) {
23     const int n = a.size();
24     const int m = b[0].size();
25     VI irow(n), icol(n), ipiv(n);
26     T det = 1;
27
28     for (int i = 0; i < n; i++) {
29         int pj = -1, pk = -1;
30         for (int j = 0; j < n; j++) if (!ipiv[j])
31             for (int k = 0; k < n; k++) if (!ipiv[k])

```

```

32         if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j;
33             pk = k; }
34         if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." <<
35             endl; exit(0); }
36         ipiv[pk]++;
37         swap(a[pj], a[pk]);
38         swap(b[pj], b[pk]);
39         if (pj != pk) det *= -1;
40         irow[i] = pj;
41         icol[i] = pk;
42         T c = 1.0 / a[pk][pk];
43         det *= a[pk][pk];
44         a[pk][pk] = 1.0;
45         for (int p = 0; p < n; p++) a[pk][p] *= c;
46         for (int p = 0; p < m; p++) b[pk][p] *= c;
47         for (int p = 0; p < n; p++) if (p != pk) {
48             c = a[p][pk];
49             a[p][pk] = 0;
50             for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
51             for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
52         }
53     }
54     for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
55         for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
56     }
57     return det;
58 }
59 int main() {
60     const int n = 4;
61     const int m = 2;
62     double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
63     double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
64     VVT a(n), b(n);
65     for (int i = 0; i < n; i++) {
66         a[i] = VT(A[i], A[i] + n);
67         b[i] = VT(B[i], B[i] + m);
68     }
69     double det = GaussJordan(a, b);
70     // expected: 60
71     cout << "Determinant: " << det << endl;
72     // expected: -0.233333 0.166667 0.133333 0.0666667
73     // 0.166667 0.166667 0.333333 -0.333333
74     // 0.233333 0.833333 -0.133333 -0.0666667
75     // 0.05 -0.75 -0.1 0.2
76     cout << "Inverse: " << endl;
77     for (int i = 0; i < n; i++) {
78         for (int j = 0; j < n; j++)
79             cout << a[i][j] << ' ';
80         cout << endl;
81     }
82     // expected: 1.63333 1.3
83     // -0.166667 0.5
84     // 2.36667 1.7
85     // -1.85 -1.35
86     cout << "Solution: " << endl;
87     for (int i = 0; i < n; i++) {
88         for (int j = 0; j < m; j++)

```

```

89         cout << b[i][j] << ' ';
90         cout << endl;
91     }
92 }

```

---

## 6.4 Matrix

```

1 template<typename T> struct matrix {
2     int N;
3     vector<T> dat;
4
5     matrix<T> (int _N, T fill = T(0), T diag = T(0)) {
6         N = _N;
7         dat.resize(N * N, fill);
8         for (int i = 0; i < N; i++)
9             (*this)(i, i) = diag;
10    }
11    T& operator()(int i, int j) {
12        return dat[N * i + j];
13    }
14    matrix<T> operator*(matrix<T> &b){
15        matrix<T> r(N);
16        for(int i=0; i<N; i++)
17            for(int j=0; j<N; j++)
18                for(int k=0; k<N; k++)
19                    r(i, j) = r(i, j) + (*this)(i, k) * b(k, j);
20        return r;
21    }
22    matrix<T> pow(ll expo){
23        if(!expo) return matrix<T>(N, T(0), T(1));
24        matrix<T> r = (*this * *this).pow(expo/2);
25        return expo&1 ? r * *this : r;
26    }
27    friend ostream& operator<<(ostream &os, matrix<T> &m){
28        os << "{";
29        for(int i=0; i<m.N; i++){
30            if(i) os << "],\n ";
31            os << "{";
32            for(int j=0; j<m.N; j++){
33                if(j) os << ", ";
34                os << setw(10) << m(i, j) << setw(0);
35            }
36            os << "}]";
37        }
38        return os << "}";
39    }
40 };
41 struct mll {
42     const int MOD;
43     ll val;
44     mll(ll _val = 0) {
45         val = _val % MOD;
46         if (val < 0) val += MOD;
47     }
48     mll operator+(const mll &o) {
49         return mll((val + o.val) % MOD);
50     }
51     mll operator*(const mll &o) {

```

```

52     return mll((val * o.val) % MOD);
53 }
54 friend ostream& operator<<(ostream &os, mll &m) {
55     return os << m.val;
56 }
57 };

```

## 6.5 Primes

```

1 // O(sqrt(x)) Exhaustive Primality Test
2 #define EPS 1e-7
3 typedef long long LL;
4 bool IsPrimeSlow (LL x) {
5     if(x<=1) return false;
6     if(x<=3) return true;
7     if (!(x%2) || !(x%3)) return false;
8     LL s=(LL)(sqrt((double)(x))+EPS);
9     for(LL i=5;i<=s;i+=6)
10         if (!(x%i) || !(x%(i+2))) return false;
11     return true;
12 }
13 // Primes less than 1000:
14 //  2  3  5  7 11 13 17 19 23 29 31 37
15 // 41 43 47 53 59 61 67 71 73 79 83 89
16 // 97 101 103 107 109 113 127 131 137 139 149 151
17 // 157 163 167 173 179 181 191 193 197 199 211 223
18 // 227 229 233 239 241 251 257 263 269 271 277 281
19 // 283 293 307 311 313 317 331 337 347 349 353 359
20 // 367 373 379 383 389 397 401 409 419 421 431 433
21 // 439 443 449 457 461 463 467 479 487 491 499 503
22 // 509 521 523 541 547 557 563 569 571 577 587 593
23 // 599 601 607 613 617 619 631 641 643 647 653 659
24 // 661 673 677 683 691 701 709 719 727 733 739 743
25 // 751 757 761 769 773 787 797 809 811 821 823 827
26 // 829 839 853 857 859 863 877 881 883 887 907 911
27 // 919 929 937 941 947 953 967 971 977 983 991 997
28 // Other primes:
29 // The largest prime smaller than 10^x:
30 // 7 97 997 9973 99991 999983 9999991 99999989 999999937 9999999967
31 // 9999999977 99999999989 999999999971 9999999999973 99999999999989
32 // 999999999999937 999999999999997 9999999999999989

```

## 6.6 Reduced Row Echelon Form

```

1 // Reduced row echelon form via Gauss-Jordan elimination
2 // with partial pivoting. This can be used for computing
3 // the rank of a matrix.
4 //
5 // Running time: O(n^3)
6 //
7 // INPUT: a[] [] = an nxn matrix
8 //
9 // OUTPUT: rref[] [] = an nxm matrix (stored in a[] [])
10 // returns rank of a[] []
11 typedef vector<double> VD;
12 typedef vector<VD> VVD;
13 const double EPSILON = 1e-7;

```

```

14
15 // returns rank
16 int rref (VVD &a){
17     int i,j,r,c;
18     int n = a.size();
19     int m = a[0].size();
20     for (r=c=0;c<m;c++){
21         j=r;
22         for (i=r+1;i<n;i++) if (fabs(a[i][c])>fabs(a[j][c])) j = i;
23         if (fabs(a[j][c])<EPSILON) continue;
24         for (i=0;i<m;i++) swap(a[j][i],a[r][i]);
25         double s = a[r][c];
26         for (j=0;j<m;j++) a[r][j] /= s;
27         for (i=0;i<n;i++) if (i != r){
28             double t = a[i][c];
29             for (j=0;j<m;j++) a[i][j] -= t*a[r][j];
30         }
31         r++;
32     }
33     return r;
34 }

```

## 6.7 Simplex

```

1 // Two-phase simplex algorithm for solving linear programs of the form
2 //
3 // maximize c^T x
4 // subject to Ax <= b
5 // x >= 0
6 //
7 // INPUT: A -- an m x n matrix
8 // b -- an m-dimensional vector
9 // c -- an n-dimensional vector
10 // x -- a vector where the optimal solution will be stored
11 //
12 // OUTPUT: value of the optimal solution (infinity if unbounded
13 // above, nan if infeasible)
14 //
15 // To use this code, create an LPSolver object with A, b, and c as
16 // arguments. Then, call Solve(x).
17 typedef long double DOUBLE;
18 typedef vector<DOUBLE> VD;
19 typedef vector<VD> VVD;
20 typedef vector<int> VI;
21 const DOUBLE EPS = 1e-9;
22
23 struct LPSolver {
24     int m, n;
25     VI B, N;
26     VVD D;
27
28     LPSolver(const VVD &A, const VD &b, const VD &c) :
29         m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
30         for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
31             D[i][j] = A[i][j];
32         for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1;
33             D[i][n + 1] = b[i]; }
34         for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }

```

```

33     N[n] = -1; D[m + 1][n] = 1;
34 }
35 void Pivot(int r, int s) {
36     for (int i = 0; i < m + 2; i++) if (i != r)
37         for (int j = 0; j < n + 2; j++) if (j != s)
38             D[i][j] -= D[r][j] * D[i][s] / D[r][s];
39     for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
40     for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
41     D[r][s] = 1.0 / D[r][s];
42     swap(B[r], N[s]);
43 }
44 bool Simplex(int phase) {
45     int x = phase == 1 ? m + 1 : m;
46     while (true) {
47         int s = -1;
48         for (int j = 0; j <= n; j++) {
49             if (phase == 2 && N[j] == -1) continue;
50             if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] &&
51                 N[j] < N[s]) s = j;
52         }
53         if (D[x][s] > -EPS) return true;
54         int r = -1;
55         for (int i = 0; i < m; i++) {
56             if (D[i][s] < EPS) continue;
57             if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] /
58                 D[r][s] ||
59                 (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s])
60                 && B[i] < B[r]) r = i;
61         }
62         if (r == -1) return false;
63         Pivot(r, s);
64     }
65 }
66 DOUBLE Solve(VD &x) {
67     int r = 0;
68     for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
69     if (D[r][n + 1] < -EPS) {
70         Pivot(r, n);
71         if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return
72             -numeric_limits<DOUBLE>::infinity();
73         for (int i = 0; i < m; i++) if (B[i] == -1) {
74             int s = -1;
75             for (int j = 0; j <= n; j++)
76                 if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s]
77                     && N[j] < N[s]) s = j;
78             Pivot(i, s);
79         }
80     }
81     if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
82     x = VD(n);
83     for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
84     return D[m][n + 1];
85 }
86 int main() {
87     const int m = 4;
88     const int n = 3;
89     DOUBLE _A[m][n] = {

```

```

87     { 6, -1, 0 },
88     { -1, -5, 0 },
89     { 1, 5, 1 },
90     { -1, -5, -1 }
91 };
92 DOUBLE _b[m] = { 10, -4, 5, -5 };
93 DOUBLE _c[n] = { 1, -1, 0 };
94 VVD A(m);
95 VD b(_b, _b + m);
96 VD c(_c, _c + n);
97 for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);
98
99 LPSolver solver(A, b, c);
100 VD x;
101 DOUBLE value = solver.Solve(x);
102 cerr << "VALUE: " << value << endl; // VALUE: 1.29032
103 cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
104 for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
105 cerr << endl;
106 return 0;
107 }

```

## 7 7 String

### 7.1 Aho-Corasick

```

1 namespace aho_corasick {
2     const int SIGMA = 2;
3     const int TOTL = 1e7 + 100;
4
5     struct node {
6         int link[SIGMA];
7         int suff, dict, patt;
8         node() {
9             suff = 0, dict = 0, patt = -1;
10             memset(link, 0, sizeof(link));
11         }
12         // link[]: contains trie links + failure links
13         // suff: link to longest proper suffix that exists in the trie
14         // dict: link to longest suffix that exists in the dictionary
15         // patt: index of this node's word in the dictionary
16     };
17
18     int tail = 1;
19     vector<node> trie(TOTL);
20     vector<string> patterns;
21
22     void add_pattern(string &s) {
23         int loc = 0;
24         for (char c : s) {
25             int &nloc = trie[loc].link[c - 'a'];
26             if (!nloc) nloc = tail++;
27             loc = nloc;
28         }
29         trie[loc].dict = loc;
30         trie[loc].patt = patterns.size();
31         patterns.push_back(s);

```



```

32 }
33 void calc_links() {
34     queue<int> bfs({0});
35     while (!bfs.empty()) {
36         int loc = bfs.front(); bfs.pop();
37         int fail = trie[loc].suff;
38         if (!trie[loc].dict)
39             trie[loc].dict = trie[fail].dict;
40         for (int c = 0; c < SIGMA; c++) {
41             int &succ = trie[loc].link[c];
42             if (succ) {
43                 trie[succ].suff = loc ? trie[fail].link[c] : 0;
44                 bfs.push(succ);
45             } else succ = trie[fail].link[c];
46         }
47     }
48 }
49 void match(string &s, vector<bool> &matches) {
50     int loc = 0;
51     for (char c : s) {
52         loc = trie[loc].link[c-'a'];
53         for (int dm = trie[loc].dict; dm; dm =
54             trie[trie[dm].suff].dict) {
55             if (matches[trie[dm].patt]) break;
56             matches[trie[dm].patt] = true;
57         }
58     }
59 }

```

## 7.2 KMP

```

1 template<typename T> struct kmp {
2     int M;
3     vector<T> needle;
4     vector<int> succ;
5
6     kmp(vector<T> _needle) {
7         needle = _needle;
8         M = needle.size();
9         succ.resize(M + 1);
10        succ[0] = -1, succ[1] = 0;
11        int cur = 0;
12        for (int i = 2; i <= M; ) {
13            if (needle[i-1] == needle[cur]) succ[i++] = ++cur;
14            else if (cur == succ[cur])
15                succ[i++] = 0;
16        }
17    }
18    vector<bool> find(vector<T> &haystack) {
19        int N = haystack.size(), i = 0;
20        vector<bool> res(N);
21        for (int m = 0; m + i < N; ) {
22            if (i < M && needle[i] == haystack[m + i]) {
23                if (i == M - 1) res[m] = true;
24                i++;
25            } else if (succ[i] != -1) {
26                m = m + i - succ[i];

```

```

27         i = succ[i];
28     } else {
29         i = 0;
30         m++;
31     }
32 }
33 return res;
34 }
35 };

```

## 7.3 Suffix Arrays

```

1 // Suffix array construction in  $O(L \log^2 L)$  time. Routine for
2 // computing the length of the longest common prefix of any two
3 // suffixes in  $O(\log L)$  time.
4 //
5 // INPUT: string s
6 //
7 // OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)
8 //         of substring s[i..L-1] in the list of sorted suffixes.
9 //         That is, if we take the inverse of the permutation suffix[],
10 //         we get the actual suffix array.
11 struct SuffixArray {
12     const int L;
13     string s;
14     vector<vector<int>> > P;
15     vector<pair<pair<int,int>,int>> > M;
16
17     SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L,
18         0)), M(L) {
19         for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
20         for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
21             P.push_back(vector<int>(L, 0));
22             for (int i = 0; i < L; i++)
23                 M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ?
24                     P[level-1][i + skip] : -1000), i);
25             sort(M.begin(), M.end());
26             for (int i = 0; i < L; i++)
27                 P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ?
28                     P[level][M[i-1].second] : i;
29         }
30     }
31     vector<int> GetSuffixArray() { return P.back(); }
32
33     // returns the length of the longest common prefix of s[i..L-1] and
34     // s[j..L-1]
35     int LongestCommonPrefix(int i, int j) {
36         int len = 0;
37         if (i == j) return L - i;
38         for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
39             if (P[k][i] == P[k][j]) {
40                 i += 1 << k;
41                 j += 1 << k;
42                 len += 1 << k;
43             }
44         }
45         return len;

```



```

43 }
44 };
45
46 int main() {
47     // bobocel is the 0'th suffix
48     // obocel is the 5'th suffix
49     // bocel is the 1'st suffix
50     // ocel is the 6'th suffix
51     // cel is the 2'nd suffix
52     // el is the 3'rd suffix
53     // l is the 4'th suffix
54     SuffixArray suffix("bobocel");
55     vector<int> v = suffix.GetSuffixArray();
56     // Expected output: 0 5 1 6 2 3 4
57     //      2
58     for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
59     cout << endl;
60     cout << suffix.LongestCommonPrefix(0, 2) << endl;
61 }

```

## 8 8 Misc

### 8.1 IO

```

1 int main() {
2     // Ouput a specific number of digits past the decimal point,
3     // in this case 5
4     cout.setf(ios::fixed); cout << setprecision(5);
5     cout << 100.0/7.0 << endl;
6     cout.unsetf(ios::fixed);
7     // Output the decimal point and trailing zeros
8     cout.setf(ios::showpoint);
9     cout << 100.0 << endl;
10    cout.unsetf(ios::showpoint);
11    // Output a '+' before positive values
12    cout.setf(ios::showpos);
13    cout << 100 << " " << -100 << endl;
14    cout.unsetf(ios::showpos);
15    // Output numerical values in hexadecimal
16    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
17 }

```

### 8.2 Longest Increasing Subsequence

```

1 // Given a list of numbers of length n, this routine extracts a
2 // longest increasing subsequence.
3 //
4 // Running time: O(n log n)
5 //
6 // INPUT: a vector of integers
7 // OUTPUT: a vector containing the longest increasing subsequence
8 typedef vector<int> VI;
9 typedef pair<int,int> PII;
10 typedef vector<PII> VPPII;
11
12 #define STRICTLY_INCREASNG

```

```

13 VI LongestIncreasingSubsequence(VI v) {
14     VPPII best;
15     VI dad(v.size(), -1);
16     for (int i = 0; i < v.size(); i++) {
17         #ifndef STRICTLY_INCREASNG
18             PII item = make_pair(v[i], 0);
19             VPPII::iterator it = lower_bound(best.begin(), best.end(), item);
20             item.second = i;
21         #else
22             PII item = make_pair(v[i], i);
23             VPPII::iterator it = upper_bound(best.begin(), best.end(), item);
24         #endif
25         if (it == best.end()) {
26             dad[i] = (best.size() == 0 ? -1 : best.back().second);
27             best.push_back(item);
28         } else {
29             dad[i] = dad[it->second];
30             *it = item;
31         }
32     }
33     VI ret;
34     for (int i = best.back().second; i >= 0; i = dad[i])
35         ret.push_back(v[i]);
36     reverse(ret.begin(), ret.end());
37     return ret;
38 }

```

### 8.3 Regular Expressions - Java

```

1 // Code which demonstrates the use of Java's regular expression libraries.
2 // This is a solution for
3 //
4 // Loglan: a logical language
5 // http://acm.uva.es/p/v1/134.html
6 //
7 // In this problem, we are given a regular language, whose rules can be
8 // inferred directly from the code. For each sentence in the input, we
9 // must
10 // determine whether the sentence matches the regular expression or not.
11 // The
12 // code consists of (1) building the regular expression (which is fairly
13 // complex) and (2) using the regex to match sentences.
14
15 import java.util.*;
16 import java.util.regex.*;
17
18 public class LogLan {
19     public static String BuildRegex () {
20         String space = " ";
21         String A = "[aeiou]";
22         String C = "[a-z&&[^aeiou]]";
23         String MOD = "(g" + A + ")";
24         String BA = "(b" + A + ")";
25         String DA = "(d" + A + ")";
26         String LA = "(l" + A + ")";
27         String NAM = "([a-z]*" + C + ")";

```

```

28 String PREDA = "(" + C + C + A + C + A + "|" + C + A + C + C + A +
29     ")";
30 String predstring = "(" + PREDA + "(" + space + PREDA + ")*";
31 String predname = "(" + LA + space + predstring + "|" + NAM + ")";
32 String preds = "(" + predstring + "(" + space + A + space +
33     predstring + ")*";
34 String predclaim = "(" + predname + space + BA + space + preds + "|"
35     + DA + space +
36     preds + ")";
37 String verbpred = "(" + MOD + space + predstring + ")";
38 String statement = "(" + predname + space + verbpred + space +
39     predname + "|" +
40     predname + space + verbpred + ")";
41 String sentence = "(" + statement + "|" + predclaim + ")";
42
43 return "^" + sentence + "$";
44 }
45
46 public static void main (String args[]){
47
48     String regex = BuildRegex();
49     Pattern pattern = Pattern.compile (regex);
50
51     Scanner s = new Scanner(System.in);
52     while (true) {
53
54         // In this problem, each sentence consists of multiple lines, where
55         // the last
56         // line is terminated by a period. The code below reads lines until
57         // encountering a line whose final character is a '.'. Note the use
58         // of
59         // s.length() to get length of string
60         // s.charAt() to extract characters from a Java string
61         // s.trim() to remove whitespace from the beginning and end of Java
62         // string
63         // Other useful String manipulation methods include
64         // s.compareTo(t) < 0 if s < t, lexicographically
65         // s.indexOf("apple") returns index of first occurrence of "apple"
66         // in s
67         // s.lastIndexOf("apple") returns index of last occurrence of
68         // "apple" in s
69         // s.replace(c,d) replaces occurrences of character c with d
70         // s.startsWith("apple") returns (s.indexOf("apple") == 0)
71         // s.toLowerCase() / s.toUpperCase() returns a new lower/uppercased
72         // string
73         // Integer.parseInt(s) converts s to an integer (32-bit)
74         // Long.parseLong(s) converts s to a long (64-bit)
75         // Double.parseDouble(s) converts s to a double
76
77         String sentence = "";
78         while (true){
79             sentence = (sentence + " " + s.nextLine()).trim();
80             if (sentence.equals("#")) return;
81             if (sentence.charAt(sentence.length()-1) == '.') break;
82         }
83     }
84 }

```

```

77 }
78
79 // now, we remove the period, and match the regular expression
80
81 String removed_period = sentence.substring(0,
82     sentence.length()-1).trim();
83 if (pattern.matcher (removed_period).find()){
84     System.out.println ("Good");
85 } else {
86     System.out.println ("Bad!");
87 }
88 }
89 }

```

## 8.4 Tokenizer

```

1 // char tokens
2 vector<string> &split(const string &s, char delim, vector<string> &elems){
3     stringstream ss(s);
4     string item;
5     while (getline(ss, item, delim)) {
6         elems.push_back(item);
7     }
8     return elems;
9 }
10 vector<string> split(const string &s, char delim) {
11     vector<string> elems;
12     split(s, delim, elems);
13     return elems;
14 }
15 // string tokens
16 vector<string> split(string str, string token) {
17     vector<string> result;
18     int next;
19     while((next = str.find(token)) != string::npos) {
20         result.push_back(str.substr(0, next));
21         str = str.substr(next + token.size());
22     }
23     return result;
24 }
25
26 int main() {
27     string test = "Hello, this is a string that we might, like, like to
28         parse! ";
29     auto v_space = split(test, ' ');
30     auto v_comma = split(test, ',');
31     cout << "|"; for(auto s : v_space) cout << s << "|"; cout << endl;
32     cout << "|"; for(auto s : v_comma) cout << s << "|"; cout << endl;
33     // |Hello,|this|is|a|string|that|we|might,|like,|like|to|parse!|
34     // |Hello| this is a string that we might| like| like to parse! |
35     return 0;
36 }

```

## 8.5 Z Reference

```

1 // STL Quick Reference

```

```

2 // most have begin(), end(), empty(), clear(), size()
3 // rbegin(), rend() => array, vector, string, set, multiset, map, deque
4 // lower/upper_bound(), equal_range() => set, multiset, map
5 int a1[10], a2[10] = {0}, a3[10] = {1,2,3,4,5,6,7,8,9,10}; // not STL
6 array<int, 10> a4 = {1,2,3,4,5,6,7,8,9,10};
7 pair<int, int> p1, p2 = {1,2}, p3 = make_pair(1,2);
8 tuple<int, char> tp1(10, 'x'), tp2 = make_tuple(30, 'y');
9     get<0>(tp1) = 3, tie(ignore, mychar) = tp1 // unpack into mychar
10     tuple_size<decltype(tp1)>, tuple_cat(tp1, tp2),
11     forward_as_tuple(1, 'a') // use as something to pass to functions
12 vector<int> v1, v2(100), v3(100, 0), v4(a3, a3 + 10), v5 = {1,3};
13     v2.push_back(1), v2.pop_back(), v2[2], v2.empty(), v2.clear()
14     front(), back(), reserve()
15     find(v5.begin(), v5.end(), 3) != v5.end();
16     v5.insert(find(v5.begin(), v5.end(), 1), 2); // => {2,1,3}
17     v5.erase(find(v5.begin(), v5.end(), 3)); // => {2,1}
18 string str1, str2("hello"), str3(str2), str4(str2,1,3), str5(10,'x');
19 // has the functions vector has, but also has:
20     c_str(), find(), rfind(), substr(), compare(), (+)
21     str2.find("el") != string::npos;
22 set<int> s1, s2 = {1,2}, s3(v.begin(), v.end()); set<int, classcomp> s4;
23     insert(el), erase(it), erase(value), find(el), count(el),
24     s2.lower_bound("p"), s2.upper_bound("p"), s2.equal_range("p")
25     pair<it, bool unique> ret = s2.emplace(3), s2.emplace_hint(it2, 3);
26 multiset<int> ms1 // same as set, except stores multiple copies
27     it ret = ms1.emplace(it2, 3);
28 unordered_set<int> us1 // same as set, except without ordering
29     bucket_count(), bucket_size(), bucket(el), reserve(10),
30     hash_function(el), max_load_factor(max_load_factor()/2.0)
31 map<string, int> m1, m2 = {"zero", 0}, {"one", 1}, {"two", 2};
32     m1["0"] = 0, m1.erase("0"), m.clear(),
33     m2.lower_bound("p"), m2.upper_bound("p"), m2.equal_range("p")
34     m1.find("0") != m1.end();
35     for(auto p : m1) cout << p1.first << " " << p1.second << endl;
36     auto it = m2.lower_bound()
37 unordered_map<string, int> um1 // same as map, except without ordering
38 // also has same utilities as unordered_set regarding hashing
39 queue<int> q1, q2(other stl object)
40     front(), back(), push(), pop() // NO iterators begin(), end(), etc
41 priority_queue<int[, vector<int>, greater<int>]> pq1, pq2(a1, a1+5);
42     top(), push(), pop() // NO iterators begin(), end(), etc
43 deque<int> dq1, dq2(4,100), dq3(dq2.begin(), dq2.end()), dq4(dq3);
44     front(), back(), push_front(), push_back(), pop_front(), pop_back()
45 list<int> l1; // same constructors and functions as deque, but also:
46     insert(it, 4), insert(it, 4, 10), insert(it, a1, a1+5),
47     l1.splice(it, l2), l1.remove(89), l1.unique(), merge(), reverse()
48 bitset<> bs1, bs2(0xfa2), bs3("0101");
49     none(), any(), test(3), count() // # set
50     set(), set(1), set(1, 0), reset(), reset(1), flip(), flip(1),
51     to_string(), to_ulong(), to_ullong()
52 numeric_limits<int>::min(), max(), is_signed, digits, has_infinity
53
54 // Bounds (equal_range gives pair {lower_bound, upper_bound})
55 1 4 5 6 6 6 8 9 14 // sample multiset
56     ^-----^ // lower_bound(6) => 6, upper_bound(6) => 8
57     ^-^ // lower_bound(5) => 5, upper_bound(5) => 6
58     ^ // lower_bound(7) => 8, upper_bound(7) => 8
59     ^ // lower_bound(0) => 1, upper_bound(0) => 1
60     ^ // lower_bound(20) => (end), upper_bound(20) => (end)

```

```

61
62 // Comparisons
63 bool myfunction (int i,int j) { return (i<j); } // function version
64 struct myclass {
65     bool operator() (int i,int j) { return (i<j);} // object version
66 } myobject;
67 int myints[] = {32,71,12,45,26,80,53,33};
68 std::vector<int> myvector (myints, myints+8);
69 sort(v.begin(), v.end(), myfunction);
70 sort(v.begin(), v.end(), myobject);
71
72 // Misc.
73 for(auto i : v) { cout << i << endl; }
74 for(auto &i : v) { i *= 2; }
75 sort(v.begin(), v.end(),
76     [&](const int& a, const& int b) -> bool {return a < b;});
77 unique(v.begin(), v.end());
78 all_of(v.begin(), v.end(), [](int i){return i % 2 == 0;});
79 all_of, none_of, any_of, find_if_not, copy_if, minmax, minmax_element
80
81 // Random
82 unsigned seed = chrono::system_clock::now().time_since_epoch().count();
83 shuffle(foo.begin(), foo.end(), std::default_random_engine(seed));
84 random_shuffle(myvector.begin(), myvector.end());
85 next_permutation(v.begin(), v.end());
86
87 // Regex
88 bool equals = regex_match("subject", regex("(sub)(.*)"));
89
90 // Timing
91 auto start = high_resolution_clock::now();
92 auto end = high_resolution_clock::now();
93 cout<<duration_cast<milliseconds>(end-start).count()<<endl;
94
95 // Ratios: add, subtract, multiply, divide,
96 // equal, not_equal, less, less_equal, greater, greater_equal
97 using sum = ratio_add<ratio<1,2>, ratio<2,3>>;
98 cout << "sum = " << sum::num << "/" << sum::den;
99
100 // Functional
101 [capture] (args) {return func();} // (capture = &) captures all
102 int RandomNumber() { return rand() % 100; }
103 generate(v.begin(), v.end(), RandomNumber); // like map
104 int myfunction (int x, int y) {return x + 2*y;}
105 int init = 100;
106 accumulate(v.begin(), v.end(), init, myfunction); // like fold
107 for_each(v.begin(), v.end(), [&](int x) {cout << x << endl;});

```