Contents

1 1 Templates

1.1 Start

```
// .vimrc
syn on
set mouse=a sw=4 ts=4 ai si nu wrap
nnoremap ; :

// Terminal: comparing generated output to sample output
./my_program < sample.in | diff sample.out -
```

1.2 Template - C++

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
static bool DBG = 1;

ll mod(ll a, ll b) { return ((a%b)+b)%b; }

int main() {
    ios_base::sync_with_stdio(0);
    cout << fixed << setprecision(15);
    int n;
    cin >> n;
    cout << n << endl;
    return 0;
}
```

1.3 Template - Java

```java
import java.util.*;
import java.math.*;
import java.io.*;

class modelo {
    static final double EPS = 1.e-10;
    static final boolean DBG = true;

    private static int cmp(double x, double y = 0, double tol = EPS) {
        return (x <= y + tol)? (x + tol < y)? -1 : 0 : 1;
    }

    public static void main(String[] argv) {
        Scanner s = new Scanner(System.in);
    }
}
```

2   2 Data Structures

2.1   AVL Tree

```
struct Node {
    Node *l, *r; int h, size, key;
    Node(int k) : l(0), r(0), h(1), size(1), key(k) {}
    void u() { h=1+max(l?l->h:0, r?r->h:0);
        size=(l?l->size:0)+1+(r?r->size:0); }
};
Node *rotl(Node *x) { Node *y=x->r; x->r=y->l; y->l=x; x->u(); y->u();
    return y; }
Node *rotr(Node *x) { Node *y=x->l; x->l=y->r; y->r=x; x->u(); y->u();
    return y; }
Node *rebalance(Node *x) {
    x->u();
    if (x->l->h > 1 + x->r->h) {
        if (x->l->l->h < x->l->r->h) x->l = rotl(x->l);
        x = rotr(x);
    } else if (x->r->h > 1 + x->l->h) {
        if (x->r->r->h < x->r->l->h) x->r = rotr(x->r);
        x = rotl(x);
    }
    return x;
}
Node *insert(Node *x, int key) {
    if (x == NULL) return new Node(key);
    if (key < x->key) x->l = insert(x->l, key); else x->r = insert(x->r,
        key);
    return rebalance(x);
}
```

2.2   BIT

```
template<typename T> struct BIT{
    int S;
    vector<T> v;

    BIT<T>(int _S){
        S = _S;
        v.resize(S+1);
    }
    void update(int i, T k){
        for(i++; i<=S; i+=i&-i)
            v[i] = v[i] + k;
    }
    T read(int i){
        T sum = 0;
        for(i++; i; i-=i&-i)
            sum = sum + v[i];
        return sum;
    }
    T read(int l, int r){
        return read(r) - read(l-1);
    }
};
```

2.3   KD Tree

```
// ----------------------------------------------------------------
// - constructs from n points in O(n lg^2 n) time
// - nearest-neighbor query in O(lg n) if points are well distributed
// - worst case nearest-neighbor may be linear
// ----------------------------------------------------------------

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};
bool operator==(const point &a, const point &b) {
    return a.x == b.x && a.y == b.y;
}
// sorts points on x-coordinate
bool on_x(const point &a, const point &b) {
    return a.x < b.x;
}
// sorts points on y-coordinate
bool on_y(const point &a, const point &b) {
    return a.y < b.y;
}
// squared distance between points
ntype pdist2(const point &a, const point &b) {
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}
// bounding box for a set of points
struct bbox {
    ntype x0, x1, y0, y1;
    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}
    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x); x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y); y1 = max(y1, v[i].y);
        }
    }
    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            if (p.y < y0)      return pdist2(point(x0, y0), p);
            else if (p.y > y1) return pdist2(point(x0, y1), p);
            else          return pdist2(point(x0, p.y), p);
        }
        else if (p.x > x1) {
            if (p.y < y0)      return pdist2(point(x1, y0), p);
            else if (p.y > y1) return pdist2(point(x1, y1), p);
            else          return pdist2(point(x1, p.y), p);
        }
        else {
            if (p.y < y0)      return pdist2(point(p.x, y0), p);
            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;
```

```
16              //      else
17              return pdist2(p, node->pt);
18          }
19          ntype bfirst = node->first->intersect(p);
20          ntype bsecond = node->second->intersect(p);
21          // choose the side with the closest bounding box to search first
22          // (note that the other side is also searched if needed)
23          if (bfirst < bsecond) {
24              ntype best = search(node->first, p);
25              if (bsecond < best)
26                  best = min(best, search(node->second, p));
27              return best;
28          }
29          else {
30              ntype best = search(node->second, p);
31              if (bfirst < best)
32                  best = min(best, search(node->first, p));
33              return best;
34          }
35      }
36      // squared distance to the nearest
37      ntype nearest(const point &p) {
38          return search(root, p);
39      }
40  };
41
42  // some basic test code here
43  int main() {
44      // generate some random points for a kd-tree
45      vector<point> vp;
46      for (int i = 0; i < 100000; ++i) {
47          vp.push_back(point(rand()%100000, rand()%100000));
48      }
49      kdtree tree(vp);
50      // query some points
51      for (int i = 0; i < 10; ++i) {
52          point q(rand()%100000, rand()%100000);
53          cout << "Closest squared distance to (" << q.x << ", " << q.y <<
                 ")"
54              << " is " << tree.nearest(q) << endl;
55      }
56  }
```

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

```
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
            be overridden in child class
88      virtual Point maxNode(Point a, Point b) {
89          return max(a, b);
90      }
91      // dummy node
92      virtual Point def() {
93          return Point(0, 0, -INF);
94      }
95  };
96
97  /* 2D Segment Tree for range minimum query; a override of Segtree2d class
        */
98  struct Segtree2dMin : Segtree2d {
99      // overload maxNode() function to return minimum value
100     Point maxNode(Point a, Point b) { return min(a, b); }
101     Point def() { return Point(0, 0, INF); }
102 };
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      seg_tree<T>(int _S, T _zero = T()) {
6          S = _S, zero = _zero;
7          value.resize(2*S+1, zero);
8      }
9      void set_leaves(vector<T> &leaves) {
10         copy(leaves.begin(), leaves.end(), value.begin() + S);
11         for (int i = S - 1; i > 0; i--)
12             value[i] = value[2 * i] + value[2 * i + 1];
13     }
14     void upd(int i, T v) {
15         i += S;
16         value[i] = v;
17         while(i>1){
18             i/=2;
19             value[i] = value[2*i] + value[2*i+1];
20         }
21     }
22     T query(int i, int j) {
23         T res_left = zero, res_right = zero;
24         for(i += S, j += S; i <= j; i /= 2, j /= 2){
25             if((i&1) == 1) res_left = res_left + value[i++];
26             if((j&1) == 0) res_right = value[j--] + res_right;
27         }
28         return res_left + res_right;
29     }
30 };
```

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
       node
19 int find(vector <int>& C, int x){return (C[x]==x) ? x : C[x]=find(C,
       C[x]);} //C++
20 int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);} //C
```

3 3 Graph

3.1 2-SAT

```
struct two_sat {
    int N;
    vector<vector<int>> impl;

    two_sat(int _N) {
        N = _N;
        impl.resize(2 * N);
    }
    void add_impl(int var1, bool neg1, int var2, bool neg2) {
        impl[2 * var1 + neg1].push_back(2 * var2 + neg2);
        impl[2 * var2 + !neg2].push_back(2 * var1 + !neg1);
    }
    void add_clause(int var1, bool neg1, int var2, bool neg2) {
        add_impl(var1, !neg1, var2, neg2);
    }
    void add_clause(int var1, bool neg1) {
        add_clause(var1, neg1, var1, neg1);
    }

    int V, L, C;
    stack<int> view;

    int dfs(int loc) {
        visit[loc] = V;
        label[loc] = L++;
        int low = label[loc];
        view.push(loc);
        in_view[loc] = true;
        for (int nbr : impl[loc]) {
            if(!visit[nbr]) low = min(low, dfs(nbr));
            else if(in_view[nbr]) low = min(low, label[nbr]);
        }
        if(low == label[loc]) {
            while (true) {
                int mem = view.top();
                comp[mem] = C;
                in_view[mem] = false;
                view.pop();
                if(mem == loc) break;
            }
            C++;
        }
        return low;
    }

    vector<int> visit, label, comp, in_view;

    void reset(vector<int> &v) {
        v.resize(2 * N);
        fill(v.begin(), v.end(), 0);
    }
    bool consistent() {
        V = 0, L = 0, C = 0;
        reset(visit), reset(label), reset(comp), reset(in_view);
        for (int i = 0; i < 2 * N; i++) {
            if(!visit[i]) {
                V++;
                dfs(i);
            }
        }
        for (int i = 0; i < N; i++)
            if(comp[2 * i] == comp[2 * i + 1]) return false;
        return true;
    }
};
```

3.2 Articulation and Bridge

```
// Not well-tested. Use at your own risk.
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vii> vvii;

struct ArticulationAndBridge {
    int N, root, root_children, dfs_ctr;
    vi dfs_low, dfs_num, dfs_parent, artic;
    vii bridge;
    vvii adj_list;

    ArticulationAndBridge (int N) : N(N), dfs_num(N, -1), dfs_low(N, 0),
        dfs_parent(N, 0), artic(N, 0), adj_list(N) {}
    void add_edge(int u, int v) {
        adj_list[u].push_back({v, 0});
        adj_list[v].push_back({u, 0});
    }
    void apab(int u) {
        dfs_low[u] = dfs_num[u] = dfs_ctr++;      // dfs_low[u] <=
            dfs_num[u]
        for(auto v : adj_list[u]) {
            if(dfs_num[v.first] == -1) {          // tree edge
                dfs_parent[v.first] = u;
                if(u == root) root_children++;    // special case u is root
                apab(v.first);
                if(dfs_low[v.first] >= dfs_num[u]) // articulation point
                    artic[u] = true;
                if(dfs_low[v.first] > dfs_num[u]) // bridge
                    bridge.push_back(make_pair(u, v.first));
                dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
            } else if (v.first != dfs_parent[u]) // back-edge
                dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
        }
    }
    void run() {
        dfs_ctr = 0;
        for(int i = 0; i < N; i++) {
            if(dfs_num[i] == -1) {
                root = i; root_children = 0; apab(i);
                artic[root] = (root_children > 1);
            }
        }
    }
};
```

3.3 Dense Dijkstra

```cpp
void Dijkstra (const VVT &w, VT &dist, VI &prev, int start) {
    int n = w.size();
    VI found (n);
    prev = VI(n, -1);
    dist = VT(n, 1000000000);
    dist[start] = 0;
    while (start != -1){
        found[start] = true;
        int best = -1;
        for (int k = 0; k < n; k++) if (!found[k]) {
            if (dist[k] > dist[start] + w[start][k]) {
                dist[k] = dist[start] + w[start][k];
                prev[k] = start;
            }
            if (best == -1 || dist[k] < dist[best]) best = k;
        }
        start = best;
    }
}
```

3.4 Dijkstra

```cpp
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)
typedef pair<int,int> PII;
const int INF = 2000000000;

int main(){
    int N, s, t;
    scanf ("%d%d%d", &N, &s, &t);
    vector<vector<PII> > edges(N);
    for (int i = 0; i < N; i++){
        int M;
        scanf ("%d", &M);
        for (int j = 0; j < M; j++){
            int vertex, dist;
            scanf ("%d%d", &vertex, &dist);
            edges[i].push_back (make_pair (dist, vertex)); // note order
                of arguments here
        }
    }

    // use priority queue in which top element has the "smallest" priority
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    vector<int> dist(N, INF), dad(N, -1);
    Q.push (make_pair (0, s));
    dist[s] = 0;
    while (!Q.empty()){
        PII p = Q.top();
        if (p.second == t) break;
        Q.pop();
        int here = p.second;
        for (vector<PII>::iterator it=edges[here].begin();
                it!=edges[here].end(); it++){
            if (dist[here] + it->first < dist[it->second]){
                dist[it->second] = dist[here] + it->first;
                dad[it->second] = here;
                Q.push (make_pair (dist[it->second], it->second));
            }
        }
    }
    printf ("%d\n", dist[t]);
    if (dist[t] < INF)
        for(int i=t;i!=-1;i=dad[i])
            printf ("%d%c", i, (i==s?'\n':' '));
    return 0;
}
```

3.5 Eulerian Path

```cpp
struct Edge;
typedef list<Edge>::iterator iter;

struct Edge {
    int next_vertex;
    iter reverse_edge;
    Edge(int next_vertex) : next_vertex(next_vertex) {}
};

const int max_vertices = 100;
int num_vertices;
list<Edge> adj[max_vertices]; // adjacency list
vector<int> path;

void find_path(int v) {
    while(adj[v].size() > 0) {
        int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}
void add_edge(int a, int b) {
    adj[a].push_front(Edge(b));
    iter ita = adj[a].begin();
    adj[b].push_front(Edge(a));
    iter itb = adj[b].begin();
    ita->reverse_edge = itb;
    itb->reverse_edge = ita;
}
```

3.6 Heavy Light

```cpp
template<typename T> struct heavy_light {
    lca links;
    seg_tree<T> st;
    vector<int> preorder, index, jump;

    heavy_light(const vvi &graph, int root) {
        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 };
```

3.7   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 inm = bfs.front().first;
21             int loc = bfs.front().second;
22             bfs.pop();
23             for (int nbr : g.adj[loc]) {
24                 if (vis[!inm][nbr]) continue;
25                 if ((g.match[loc] == nbr) ^ inm) continue;
26                 vis[!inm][nbr] = true;
27                 bfs.push(make_pair(!inm, 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 }
```

3.8   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]);}
```

```
31    }
```

3.9   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   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      ll BlockingFlow(int s, int t) {
38          layer.clear(); layer.resize(N, -1);
39          layer[s] = 0;
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 * totflow, |V|^3 * 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] -
                            cost[k][source];
44                    if(dist[k] > val) {
45                        dist[k] = val;
46                        dad[k] = source;
47                    }
48                }
49                if(flow[source][k] < cap[source][k]) {
50                    ll val = dist[source] + pi[source] - pi[k] +
                            cost[source][k];
51                    if(dist[k] > val) {
52                        dist[k] = val;
53                        dad[k] = source;
54                    }
55                }
56                if(dist[k] < dist[best]) best = k;
57            }
58            source = best;
59        }
60        for(int k = 0; k < N; k++)
61            pi[k] = min((ll)(pi[k] + dist[k]), INF);
62        return found[sink];
63    }
64    pair<ll,ll> mcmf(int source, int sink) {
65        ll totflow = 0, totcost = 0;
66        while(search(source, sink)) {
67            ll amt = INF;
68            for(int x = sink; x != source; x = dad[x])
69                amt = min(amt, (ll)(flow[x][dad[x]] != 0 ?
70                        flow[x][dad[x]] : cap[dad[x]][x] -
                        flow[dad[x]][x]));
71            for(int x = sink; x != source; x = dad[x]) {
72                if(flow[x][dad[x]] != 0) {
73                    flow[x][dad[x]] -= amt;
74                    totcost -= amt * cost[x][dad[x]];
75                } else {
76                    flow[dad[x]][x] += amt;
77                    totcost += amt * cost[dad[x]][x];
78                }
79            }
80            totflow += amt;
81        }
82        return {totflow, totcost};
83    }
84 };
```

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] +=
                        weights[last][j];
30                 for (int j = 0; j < N; j++) weights[j][prev] =
                        weights[prev][j];
31                 used[last] = true;
32                 cut.push_back(last);
33                 if (best_weight == -1 || w[last] < best_weight) {
34                     best_cut = cut;
35                     best_weight = w[last];
36                 }
37             } else {
38                 for (int j = 0; j < N; j++)
39                     w[j] += weights[last][j];
40                 added[last] = true;
41             }
42         }
43     }
44     return make_pair(best_weight, best_cut);
45 }
```

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 REMOVE_REDUNDANT is
3 // #defined.
4 //
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,
        starting
9   //      with bottommost/leftmost point
10
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) <
            make_pair(rhs.y,rhs.x); }
19      bool operator==(const PT &rhs) const { return make_pair(y,x) ==
            make_pair(rhs.y,rhs.x); }
20  };
21  T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
22  T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }
23  #ifdef REMOVE_REDUNDANT
24  bool between(const PT &a, const PT &b, const PT &c) {
25      return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 &&
            (a.y-b.y)*(c.y-b.y) <= 0);
26  }
27  #endif
28  void ConvexHull(vector<PT> &pts) {
29      sort(pts.begin(), pts.end());
30      pts.erase(unique(pts.begin(), pts.end()), pts.end());
31      vector<PT> up, dn;
32      for (int i = 0; i < pts.size(); i++) {
33          while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i])
                >= 0) up.pop_back();
34          while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i])
                <= 0) dn.pop_back();
35          up.push_back(pts[i]);
36          dn.push_back(pts[i]);
37      }
38      pts = dn;
39      for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);
40  #ifdef REMOVE_REDUNDANT
41      if (pts.size() <= 2) return;
42      dn.clear();
43      dn.push_back(pts[0]);
44      dn.push_back(pts[1]);
45      for (int i = 2; i < pts.size(); i++) {
46          if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i]))
                dn.pop_back();
47          dn.push_back(pts[i]);
48      }
49      if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
50          dn[0] = dn.back();
51          dn.pop_back();
52      }
53      pts = dn;
54  #endif
55  }
```

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

5.3 Geometry

```cpp
// C++ routines for computational geometry.
double INF = 1e100;
double EPS = 1e-12;

struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c ); }
    PT operator / (double c) const { return PT(x/c, y/c ); }
};

double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p) { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
    return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b-a,b-a);
    if (fabs(r) < EPS) return a;
    r = dot(c-a, b-a)/r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
        double a, double b, double c, double d) {
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
        && fabs(cross(a-b, a-c)) < EPS
        && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
                dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
        if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
            return false;
        return true;
    }
    if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
    if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
    return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=c-d; c=c-a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return ComputeLineIntersection(b, b+RotateCW90(a-b), c,
        c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i+1)%p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
                p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) /
                (p[j].y - p[i].y))
            c = !c;
```

```cpp
    }
    return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) <
            EPS)
            return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b-a;
    a = a-c;
    double A = dot(b, b);
    double B = dot(a, b);
    double C = dot(a, a) - r*r;
    double D = B*B - A*C;
    if (D < -EPS) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
    if (D > EPS)
        ret.push_back(c+a+b*(-B-sqrt(D))/A);
    return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r+R || d+min(r, R) < max(r, R)) return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = sqrt(r*r-x*x);
    PT v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if (y > 0)
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
```

```cpp
    return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);
    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == l || j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

// Plane distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0 is abs(d1 - d2) / sqrt(a*a + b*b + c*c)

// distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
// (or ray, or segment; in the case of the ray, the endpoint is the
// first point)
public static final int LINE = 0;
public static final int SEGMENT = 1;
public static final int RAY = 2;
public static double ptLineDistSq(double x1, double y1, double z1,
        double x2, double y2, double z2, double px, double py, double pz,
        int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);
    double x, y, z;
    if (pd2 == 0) {
        x = x1;
        y = y1;
        z = z1;}
    else {
        double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) /
            pd2;
        x = x1 + u * (x2 - x1);
        y = y1 + u * (y2 - y1);
        z = z1 + u * (z2 - z1);
        if (type != LINE && u < 0) {
            x = x1;
            y = y1;
            z = z1;}
        if (type == SEGMENT && u > 1.0) {
            x = x2;
            y = y2;
            z = z2;}
```

```
231        }
232        return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
233 }
234
235 int main() {
236     // expected: (-5,2)
237     cerr << RotateCCW90(PT(2,5)) << endl;
238     // expected: (5,-2)
239     cerr << RotateCW90(PT(2,5)) << endl;
240     // expected: (-5,2)
241     cerr << RotateCCW(PT(2,5),M_PI/2) << endl;
242     // expected: (5,2)
243     cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;
244     // expected: (5,2) (7.5,3) (2.5,1)
245     cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
246         << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
247         << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;
248     // expected: 6.78903
249     cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
250     // expected: 1 0 1
251     cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
252         << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
253         << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
254     // expected: 0 0 1
255     cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
256         << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
257         << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
258     // expected: 1 1 1 0
259     cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
260         << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
261         << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
262         << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;
263     // expected: (1,2)
264     cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3))
265         << endl;
265     // expected: (1,1)
266     cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;
267     vector<PT> v;
268     v.push_back(PT(0,0));
269     v.push_back(PT(5,0));
270     v.push_back(PT(5,5));
271     v.push_back(PT(0,5));
272     // expected: 1 1 1 0 0
273     cerr << PointInPolygon(v, PT(2,2)) << " "
274         << PointInPolygon(v, PT(2,0)) << " "
275         << PointInPolygon(v, PT(0,2)) << " "
276         << PointInPolygon(v, PT(5,2)) << " "
277         << PointInPolygon(v, PT(2,5)) << endl;
278     // expected: 0 1 1 1 1
279     cerr << PointOnPolygon(v, PT(2,2)) << " "
280         << PointOnPolygon(v, PT(2,0)) << " "
281         << PointOnPolygon(v, PT(0,2)) << " "
282         << PointOnPolygon(v, PT(5,2)) << " "
283         << PointOnPolygon(v, PT(2,5)) << endl;
284     // expected: (1,6)
285     //          (5,4) (4,5)
286     //          blank line
287     //          (4,5) (5,4)
288     //          blank line
```

```
289     //          (4,5) (5,4)
290     vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
291     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
292     u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
293     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
294     u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
295     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
296     u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
297     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
298     u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
299     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
300     u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
301     for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
302     // area should be 5.0
303     // centroid should be (1.1666666, 1.166666)
304     PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
305     vector<PT> p(pa, pa+4);
306     PT c = ComputeCentroid(p);
307     cerr << "Area: " << ComputeArea(p) << endl;
308     cerr << "Centroid: " << c << endl;
309 }
```

6   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
7  // return a % b (positive value)
8  int mod(int a, int b) {
9      return ((a%b)+b)%b;
10 }
11 // computes gcd(a,b)
12 int gcd(int a, int b) {
13     int tmp;
14     while(b){a%=b; tmp=a; a=b; b=tmp;}
15     return a;
16 }
17 // computes lcm(a,b)
18 int lcm(int a, int b) {
19     return a/gcd(a,b)*b;
20 }
21 // returns d = gcd(a,b); finds x,y such that d = ax + by
22 int extended_euclid(int a, int b, int &x, int &y) {
23     int xx = y = 0;
24     int yy = x = 1;
25     while (b) {
26         int q = a/b;
27         int t = b; b = a%b; a = t;
28         t = xx; xx = x-q*xx; x = t;
29         t = yy; yy = y-q*yy; y = t;
30     }
31     return a;
```

```
32      }
33      // finds all solutions to ax = b (mod n)
34      VI modular_linear_equation_solver(int a, int b, int n) {
35          int x, y;
36          VI solutions;
37          int d = extended_euclid(a, n, x, y);
38          if (!(b%d)) {
39              x = mod (x*(b/d), n);
40              for (int i = 0; i < d; i++)
41                  solutions.push_back(mod(x + i*(n/d), n));
42          }
43          return solutions;
44      }
45      // computes b such that ab = 1 (mod n), returns -1 on failure
46      int mod_inverse(int a, int n) {
47          int x, y;
48          int d = extended_euclid(a, n, x, y);
49          if (d > 1) return -1;
50          return mod(x,n);
51      }
52      // Chinese remainder theorem (special case): find z such that
53      // z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
54      // Return (z,M). On failure, M = -1.
55      PII chinese_remainder_theorem(int x, int a, int y, int b) {
56          int s, t;
57          int d = extended_euclid(x, y, s, t);
58          if (a%d != b%d) return make_pair(0, -1);
59          return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
60      }
61      // Chinese remainder theorem: find z such that
62      // z % x[i] = a[i] for all i. Note that the solution is
63      // unique modulo M = lcm_i (x[i]). Return (z,M). On
64      // 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;
00              ans2 *= i + 1;
01          }
02          return ans / ans2;
03      }
04      // computes the nth Catalan number
05      long long catalan_number(int n) {
06          return binomial(n * 2, n) / (n + 1);
07      }
08      // computes phi(n) (use euler_totient)
09      long long euler_totient2(long long n, long long ps) {
10          for (long long i = ps ; i * i <= n ; i++) {
11              if (n % i == 0) {
12                  long long p = 1;
13                  while (n % i == 0) {
14                      n /= i;
15                      p *= i;
16                  }
17                  return (p - p / i) * euler_totient2(n, i + 1);
18              }
19              if (i > 2) i++;
20          }
21          return n - 1;
22      }
23      long long euler_totient(long long n) {
24          return euler_totient2(n, 2);
25      }
26
27      int main() {
28          // expected: 2
29          cout << gcd(14, 30) << endl;
30          // expected: 2 -2 1
31          int x, y;
32          int d = extended_euclid(14, 30, x, y);
33          cout << d << " " << x << " " << y << endl;
34          // expected: 95 45
35          VI sols = modular_linear_equation_solver(14, 30, 100);
36          for (int i = 0; i < (int) sols.size(); i++) cout << sols[i] << " ";
37          cout << endl;
38          // expected: 8
39          cout << mod_inverse(8, 9) << endl;
40          // expected: 23 56
41          //          11 12
42          int xs[] = {3, 5, 7, 4, 6};
43          int as[] = {2, 3, 2, 3, 5};
44          PII ret = chinese_remainder_theorem(VI (xs, xs+3), VI(as, as+3));
45          cout << ret.first << " " << ret.second << endl;
46          ret = chinese_remainder_theorem (VI(xs+3, xs+5), VI(as+3, as+5));
47          cout << ret.first << " " << ret.second << endl;
48          // expected: 5 -15
49          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 +
                b*c.a }; }
7           cnum operator/(double d) { return { a / d, b / d }; }
8       };
9
10      const double PI = 2 * atan2(1, 0);
11      int deg;
12      vector<int> rev;
13
14      void set_degree(int _deg) {
15          assert(__builtin_popcount(_deg) == 1);
16          deg = _deg;
17          rev.resize(deg);
18          for (int i = 1, j = 0; i < deg; i++) {
19              int bit = deg / 2;
20              for (; j >= bit; bit /= 2)
21                  j -= bit;
22              j += bit;
23              rev[i] = j;
24          }
25      }
26      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;
                        pk = k; }
33          if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." <<
                endl; exit(0); }
34          ipiv[pk]++;
35          swap(a[pj], a[pk]);
36          swap(b[pj], b[pk]);
37          if (pj != pk) det *= -1;
38          irow[i] = pj;
39          icol[i] = pk;
40          T c = 1.0 / a[pk][pk];
41          det *= a[pk][pk];
42          a[pk][pk] = 1.0;
43          for (int p = 0; p < n; p++) a[pk][p] *= c;
44          for (int p = 0; p < m; p++) b[pk][p] *= c;
45          for (int p = 0; p < n; p++) if (p != pk) {
46              c = a[p][pk];
47              a[p][pk] = 0;
48              for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
49              for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
50          }
51      }
52      for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
53          for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
54      }
55      return det;
```

```
56  }
57
58  int main() {
59      const int n = 4;
60      const int m = 2;
61      double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
62      double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
63      VVT a(n), b(n);
64      for (int i = 0; i < n; i++) {
65          a[i] = VT(A[i], A[i] + n);
66          b[i] = VT(B[i], B[i] + m);
67      }
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          }
37          return os << "}}";
38      }
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 //  99999999977 999999999989 9999999999971 99999999999973 999999999999989
32 //  9999999999999937 99999999999999997 999999999999999989
```

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]; }
35         N[n] = -1; D[m + 1][n] = 1;
36     }
37     void Pivot(int r, int s) {
38         for (int i = 0; i < m + 2; i++) if (i != r)
39             for (int j = 0; j < n + 2; j++) if (j != s)
40                 D[i][j] -= D[r][j] * D[i][s] / D[r][s];
41         for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
42         for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
43         D[r][s] = 1.0 / D[r][s];
44         swap(B[r], N[s]);
45     }
46     bool Simplex(int phase) {
47         int x = phase == 1 ? m + 1 : m;
48         while (true) {
49             int s = -1;
50             for (int j = 0; j <= n; j++) {
51                 if (phase == 2 && N[j] == -1) continue;
52                 if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] &&
53                     N[j] < N[s]) s = j;
54             }
55             if (D[x][s] > -EPS) return true;
56             int r = -1;
57             for (int i = 0; i < m; i++) {
58                 if (D[i][s] < EPS) continue;
```

```
56              if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] /
                    D[r][s] ||
57                      (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s])
                        && B[i] < B[r]) r = i;
58          }
59          if (r == -1) return false;
60          Pivot(r, s);
61        }
62      }
63      DOUBLE Solve(VD &x) {
64          int r = 0;
65          for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
66          if (D[r][n + 1] < -EPS) {
67              Pivot(r, n);
68              if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return
                    -numeric_limits<DOUBLE>::infinity();
69              for (int i = 0; i < m; i++) if (B[i] == -1) {
70                  int s = -1;
71                  for (int j = 0; j <= n; j++)
72                      if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s]
                            && N[j] < N[s]) s = j;
73                  Pivot(i, s);
74              }
75          }
76          if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
77          x = VD(n);
78          for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
79          return D[m][n + 1];
80      }
81  };
82
83  int main() {
84      const int m = 4;
85      const int n = 3;
86      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 =
                    trie[trie[dm].suff].dict) {
54                  if (matches[trie[dm].patt]) break;
```

```
55              matches[trie[dm].patt] = true;
56          }
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) cur = succ[cur];
15             else 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,
           0)), M(L) {
18         for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
19         for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
20             P.push_back(vector<int>(L, 0));
21             for (int i = 0; i < L; i++)
22                 M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ?
                       P[level-1][i + skip] : -1000), i);
23             sort(M.begin(), M.end());
24             for (int i = 0; i < L; i++)
25                 P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ?
                       P[level][M[i-1].second] : i;
26         }
27     }
28
29     vector<int> GetSuffixArray() { return P.back(); }
30
31     // returns the length of the longest common prefix of s[i...L-1] and
           s[j...L-1]
32     int LongestCommonPrefix(int i, int j) {
33         int len = 0;
34         if (i == j) return L - i;
35         for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
36             if (P[k][i] == P[k][j]) {
37                 i += 1 << k;
38                 j += 1 << k;
39                 len += 1 << k;
40             }
41         }
42         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

```cpp
int main() {
  // Ouput a specific number of digits past the decimal point,
  // in this case 5
  cout.setf(ios::fixed); cout << setprecision(5);
  cout << 100.0/7.0 << endl;
  cout.unsetf(ios::fixed);
  // Output the decimal point and trailing zeros
  cout.setf(ios::showpoint);
  cout << 100.0 << endl;
  cout.unsetf(ios::showpoint);
  // Output a '+' before positive values
  cout.setf(ios::showpos);
  cout << 100 << " " << -100 << endl;
  cout.unsetf(ios::showpos);
  // Output numerical values in hexadecimal
  cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

8.2 Longest Increasing Subsequence

```cpp
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing subsequence
typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG
VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);
  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.begin(), best.end(), item);
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator it = upper_bound(best.begin(), best.end(), item);
#endif
    if (it == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.back().second);
      best.push_back(item);
    } else {
      dad[i] = dad[it->second];
      *it = item;
    }
  }
  VI ret;
  for (int i = best.back().second; i >= 0; i = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```

8.3 Regular Expressions - Java

```java
// Code which demonstrates the use of Java's regular expression libraries.
// This is a solution for
//
//   Loglan: a logical language
//   http://acm.uva.es/p/v1/134.html
//
// In this problem, we are given a regular language, whose rules can be
// inferred directly from the code. For each sentence in the input, we must
// determine whether the sentence matches the regular expression or not. The
// code consists of (1) building the regular expression (which is fairly
// complex) and (2) using the regex to match sentences.

import java.util.*;
import java.util.regex.*;

public class LogLan {

  public static String BuildRegex (){
    String space = " +";

    String A = "([aeiou])";
    String C = "([a-z&&[^aeiou]])";
    String MOD = "(g" + A + ")";
    String BA = "(b" + A + ")";
    String DA = "(d" + A + ")";
    String LA = "(l" + A + ")";
    String NAM = "([a-z]*" + C + ")";
    String PREDA = "(" + C + C + A + C + A + "|" + C + A + C + C + A +
        ")";

    String predstring = "(" + PREDA + "(" + space + PREDA + ")*)";
    String predname = "(" + LA + space + predstring + "|" + NAM + ")";
    String preds = "(" + predstring + "(" + space + A + space +
        predstring + ")*)";
    String predclaim = "(" + predname + space + BA + space + preds + "|"
        + DA + space +
      preds + ")";
    String verbpred = "(" + MOD + space + predstring + ")";
    String statement = "(" + predname + space + verbpred + space +
        predname + "|" +
      predname + space + verbpred + ")";
    String sentence = "(" + statement + "|" + predclaim + ")";

    return "^" + sentence + "$";
  }

  public static void main (String args[]){

    String regex = BuildRegex();
```

```
46      Pattern pattern = Pattern.compile (regex);
47
48      Scanner s = new Scanner(System.in);
49      while (true) {
50
51        // In this problem, each sentence consists of multiple lines, where
               the last
52        // line is terminated by a period. The code below reads lines until
53        // encountering a line whose final character is a '.'. Note the use
               of
54        //
55        //  s.length() to get length of string
56        //  s.charAt() to extract characters from a Java string
57        //  s.trim() to remove whitespace from the beginning and end of Java
               string
58        //
59        // Other useful String manipulation methods include
60        //
61        //  s.compareTo(t) < 0 if s < t, lexicographically
62        //  s.indexOf("apple") returns index of first occurrence of "apple"
               in s
63        //  s.lastIndexOf("apple") returns index of last occurrence of
               "apple" in s
64        //  s.replace(c,d) replaces occurrences of character c with d
65        //  s.startsWith("apple) returns (s.indexOf("apple") == 0)
66        //  s.toLowerCase() / s.toUpperCase() returns a new lower/uppercased
               string
67        //
68        //  Integer.parseInt(s) converts s to an integer (32-bit)
69        //  Long.parseLong(s) converts s to a long (64-bit)
70        //  Double.parseDouble(s) converts s to a double
71
72        String sentence = "";
73        while (true){
74          sentence = (sentence + " " + s.nextLine()).trim();
75          if (sentence.equals("#")) return;
76          if (sentence.charAt(sentence.length()-1) == '.') break;
77        }
78
79        // now, we remove the period, and match the regular expression
80
81        String removed_period = sentence.substring(0,
               sentence.length()-1).trim();
82        if (pattern.matcher (removed_period).find()){
83          System.out.println ("Good");
84        } else {
85          System.out.println ("Bad!");
86        }
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
               parse! ";
28     auto v_space = split(test, ' ');
29     auto v_comma = split(test, ',');
30     cout << "|"; for(auto s : v_space) cout << s << "|"; cout << endl;
31     cout << "|"; for(auto s : v_comma) cout << s << "|"; cout << endl;
32     // |Hello,|this|is|a|string|that|we|might,|like,|like|to|parse!|
33     // |Hello| this is a string that we might| like| like to parse! |
34     return 0;
35 }
```

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") != strong::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),
```

```cpp
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;});
```