Contents

1 Flow and matching

1.1 Edmond Karp

```
#define MAX_V 40
#define INF 1000000000

int res[MAX_V][MAX_V], mf, f, s, t;
vi p;
vector<vi> AdjList;

void augment(int v, int minEdge) {
  if (v == s) { f = minEdge; return; }
  else if (p[v] != -1) {
    augment(p[v], min(minEdge, res[p[v]][v]));
    res[p[v]][v] -= f; res[v][p[v]] += f; }
}

int main() {
  int V, k, vertex, weight;

  scanf("%d %d %d", &V, &s, &t);

  memset(res, 0, sizeof res);
  AdjList.assign(V, vi());
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      res[i][vertex] = weight;
      AdjList[i].push_back(vertex);
    }
  }

  mf = 0;
  while (1) {// now a true O(VE^2) Edmonds Karp's algorithm
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)AdjList[u].size(); j++) {
        int v = AdjList[u][j];
        if (res[u][v] > 0 && !vis[v])
          vis[v] = true, q.push(v), p[v] = u;
      }
    }
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }

  printf("%d\n", mf);

  return 0;
}
```

1.2 Max flow (Dinić)

```
// Dinic's blocking flow algorithm
// Running time:
// * general networks: O(|V|^2 |E|)
// * unit capacity networks: O(E min(V^(2/3), E^(1/2)))
// * bipartite matching networks: O(E sqrt(V))

const int INF = 2000000000;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) ↩
        {}
};

struct Dinic {
  int N;
  vector<vector<Edge> > G;
  vector<Edge *> dad;
  vector<int> Q;

  // N = number of vertices
  Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

  // Add an edge to initially empty network. from, to are 0-↩
      based
  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1)↩
        );
  }
```

```
  long long BlockingFlow(int s, int t) {
    fill(dad.begin(), dad.end(), (Edge *) NULL);
    dad[s] = &G[0][0] - 1;

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
      int x = Q[head++];
      for (int i = 0; i < G[x].size(); i++) {
        Edge &e = G[x][i];
        if (!dad[e.to] && e.cap - e.flow > 0) {
          dad[e.to] = &G[x][i];
          Q[tail++] = e.to;
        }
      }
    }
    if (!dad[t]) return 0;

    long long totflow = 0;
    for (int i = 0; i < G[t].size(); i++) {
      Edge *start = &G[G[t][i].to][G[t][i].index];
      int amt = INF;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->
          from]) {
        if (!e) { amt = 0; break; }
        amt = min(amt, e->cap - e->flow);
      }
      if (amt == 0) continue;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->
          from]) {
        e->flow += amt;
        G[e->to][e->index].flow -= amt;
      }
      totflow += amt;
    }
    return totflow;
  }

  // Call this to get the max flow. s, t are 0-based.
  // Note, you can only call this once.
  // To obtain the actual flow values, look at all edges
      with
  // capacity > 0 (zero capacity edges are residual edges).

  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};
```

1.3 Min-cost max-flow (successive shortest paths)

```
/* Min cost max flow (Edmonds-Karp relabelling + fast heap
     Dijkstra)
 * Based on code by Frank Chu and Igor Naverniouk
 * (http://shygypsy.com/tools/mcmf4.cpp)
 *
 * COMPLEXITY:
 *      - Worst case: O(min(m*log(m)*flow, n*m*log(m)*fcost)
     )
 * FIELD TESTING:
 *      - Valladolid 10594: Data Flow
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements in
     Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's
     implementation.
 **/

#define Inf (LLONG_MAX/2)
#define BUBL { \
    t = q[i]; q[i] = q[j]; q[j] = t; \
    t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }
#define Pot(u,v) (d[u] + pi[u] - pi[v])
struct MinCostMaxFlow {
    typedef long long LL;
    int n, qs;
    vector<vector<LL> > cap, cost, fnet;
    vector<vector<int> > adj;
    vector<LL> d, pi;
    vector<int> deg, par, q, inq;

    // n = number of vertices
    MinCostMaxFlow(int n): n(n), qs(0), deg(n+1), par(n+1),
         d(n+1), q(n+1), inq(n+1), pi(n+1), cap(n+1, vector<
         LL>(n+1)), cost(cap), fnet(cap), adj(n+1, vector<
         int>(n+1)) {}

    // call to add a directed edge. vertices are 0-based
    // ALL COSTS MUST BE NON-NEGATIVE
    void AddEdge(int from, int to, LL cap_, LL cost_) {
        cap[from][to] = cap_; cost[from][to] = cost_;
    }

    bool dijkstra( int s, int t ) {
        fill(d.begin(), d.end(), 0x3f3f3f3f3f3f3f3fLL);
        fill(par.begin(), par.end(), -1);
        fill(inq.begin(), inq.end(), -1);
        d[s] = qs = 0;
        inq[q[qs++] = s] = 0;
        par[s] = n;
        while( qs ) {
            int u = q[0]; inq[u] = -1;
            q[0] = q[--qs];
            if( qs ) inq[q[0]] = 0;
            for( int i = 0, j = 2*i + 1, t; j < qs; i = j, j
                 = 2*i + 1 ) {
                if( j + 1 < qs && d[q[j + 1]] < d[q[j]] ) j
                    ++;
                if( d[q[j]] >= d[q[i]] ) break;
                BUBL;
            }
            for( int k = 0, v = adj[u][k]; k < deg[u]; v =
                 adj[u][++k] ) {
                if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][
                    u] )
                    d[v] = Pot(u,v) - cost[v][par[v] = u];

                if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v
                    ) + cost[u][v] )
                    d[v] = Pot(u,v) + cost[par[v] = u][v];
                if( par[v] == u ) {
                    if( inq[v] < 0 ) { inq[q[qs] = v] = qs;
                        qs++; }
                    for( int i = inq[v], j = ( i - 1 )/2, t;
                        d[q[i]] < d[q[j]]; i = j, j = ( i -
                            1 )/2 )
                        BUBL;
                }
            }
        }
        for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i]
            += d[i];
        return par[t] >= 0;
    }

    // Returns: (flow, total cost) between source s and sink
        t
    // Call this once only. fnet[i][j] contains the flow
        from i to j. Careful, fnet[i][j] and fnet[j][i]
        could both be positive.
    pair<LL, LL> mcmf4(int s, int t) {
        for( int i = 0; i < n; i++ )
            for( int j = 0; j < n; j++ )
                if( cap[i][j] || cap[j][i] ) adj[i][deg[i
                    ]++] = j;
        LL flow = 0; LL fcost = 0;
        while( dijkstra( s, t ) ) {
            LL bot = LLONG_MAX;
            for( int v = t, u = par[v]; v != s; u = par[v =
                 u] )
                bot = min(bot, fnet[v][u] ? fnet[v][u] : (
                    cap[u][v] - fnet[u][v] ));
            for( int v = t, u = par[v]; v != s; u = par[v =
                 u] )
                if( fnet[v][u] ) { fnet[v][u] -= bot; fcost
                    -= bot * cost[v][u]; }
                else { fnet[u][v] += bot; fcost += bot *
                    cost[u][v]; }
            flow += bot;
        }
        return make_pair(flow, fcost);
    }
};
```

1.4 Min-cost matching

```
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting
     path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in
     around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right
     node j
//   Lmate[i] = index of right node that left node i pairs
     with
//   Rmate[j] = index of left node that right node j pairs
     with
//
// The values in cost[i][j] may be positive or negative.  To
     perform
// maximization, simply negate the cost[][] matrix.

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate
    ) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j])
        ;
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j]
        - u[i]);
```

```
}

// construct primal solution satisfying complementary slackness
Lmate = VI(n, -1);
Rmate = VI(n, -1);
int mated = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (Rmate[j] != -1) continue;
    if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
      Lmate[i] = j;
      Rmate[j] = i;
      mated++;
      break;
    }
  }
}

VD dist(n);
VI dad(n);
VI seen(n);

// repeat until primal solution is feasible
while (mated < n) {

  // find an unmatched left node
  int s = 0;
  while (Lmate[s] != -1) s++;

  // initialize Dijkstra
  fill(dad.begin(), dad.end(), -1);
  fill(seen.begin(), seen.end(), 0);
  for (int k = 0; k < n; k++)
    dist[k] = cost[s][k] - u[s] - v[k];

  int j = 0;
  while (true) {

    // find closest
    j = -1;
    for (int k = 0; k < n; k++) {
      if (seen[k]) continue;
      if (j == -1 || dist[k] < dist[j]) j = k;
    }
    seen[j] = 1;

    // termination condition
    if (Rmate[j] == -1) break;

    // relax neighbors
    const int i = Rmate[j];
    for (int k = 0; k < n; k++) {
      if (seen[k]) continue;
      const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
      if (dist[k] > new_dist) {
        dist[k] = new_dist;
        dad[k] = j;
      }
    }
  }

  // update dual variables
  for (int k = 0; k < n; k++) {
    if (k == j || !seen[k]) continue;
    const int i = Rmate[k];
    v[k] += dist[k] - dist[j];
    u[i] -= dist[k] - dist[j];
  }
  u[s] += dist[j];

  // augment along path
  while (dad[j] >= 0) {
    const int d = dad[j];
    Rmate[j] = Rmate[d];
    Lmate[Rmate[j]] = j;
    j = d;
  }
  Rmate[j] = s;
  Lmate[s] = j;

  mated++;
}

double value = 0;
for (int i = 0; i < n; i++)
  value += cost[i][Lmate[i]];

return value;
}
```

1.5 Max bipartite matching

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
// For larger input, consider Dinic, which runs in O(E sqrt(V))
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

1.6 Global min cut (Stoer–Wagner)

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm. Runs in O(V^3).
// Note, this is NOT min s-t cut, which is solved by max flow. This finds a global cut in an *undirected* graph.

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

// return value: (min cut value, nodes in half of min cut)
pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
        for (int j = 0; j < N; j++)
          weights[prev][j] += weights[last][j];
        for (int j = 0; j < N; j++)
          weights[j][prev] = weights[prev][j];
        used[last] = true;
        cut.push_back(last);
        if (best_weight == -1 || w[last] < best_weight) {
          best_cut = cut;
          best_weight = w[last];
        }
      } else {
        for (int j = 0; j < N; j++)
          w[j] += weights[last][j];
        added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}
```

1.7 König's Theorem (Text)

In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. To exhibit the vertex cover:

1. Find a maximum matching
2. Change each edge **used** in the matching into a directed edge from **right to left**
3. Change each edge **not used** in the matching into a directed edge from **left to right**
4. Compute the set T of all vertices reachable from unmatched vertices on the left (including themselves)

5. The vertex cover consists of all vertices on the right that are **in** T, and all vertices on the left that are **not in** T

1.8 General Unweighted Maximum Matching (Edmonds' algorithm)

```
// Unweighted general matching.
// Vertices are numbered from 1 to V.
// G is an adjlist.
// G[x][0] contains the number of neighbours of x.
// The neigbours are then stored in G[x][1] .. G[x][G[x↩
    ][0]].
// Mate[x] will contain the matching node for x.
// V and E are the number of edges and vertices.
// Slow Version (2x on random graphs) of Gabow's ↩
    implementation
// of Edmonds' algorithm (O(V^3)).
const int MAXV = 250;
int G[MAXV][MAXV];
int VLabel[MAXV];
int  Queue[MAXV];
int   Mate[MAXV];
int   Save[MAXV];
int   Used[MAXV];
int    Up, Down;
int            V;

void ReMatch(int x, int y)
{
  int m = Mate[x]; Mate[x] = y;
  if (Mate[m] == x)
    {
      if (VLabel[x] <= V)
        {
          Mate[m] = VLabel[x];
          ReMatch(VLabel[x], m);
        }
      else
        {
          int a = 1 + (VLabel[x] - V - 1) / V;
          int b = 1 + (VLabel[x] - V - 1) % V;
          ReMatch(a, b); ReMatch(b, a);
        }
    }
}

void Traverse(int x)
{
  for (int i = 1; i <= V; i++) Save[i] = Mate[i];
  ReMatch(x, x);
  for (int i = 1; i <= V; i++)
    {
      if (Mate[i] != Save[i]) Used[i]++;
      Mate[i] = Save[i];
    }
}

void ReLabel(int x, int y)
{
  for (int i = 1; i <= V; i++) Used[i] = 0;
  Traverse(x); Traverse(y);
  for (int i = 1; i <= V; i++)
    {
      if (Used[i] == 1 && VLabel[i] < 0)
        {
          VLabel[i] = V + x + (y - 1) * V;
          Queue[Up++] = i;
        }
      }
    }
}

// Call this after constructing G
void Solve()
{
  for (int i = 1; i <= V; i++)
    if (Mate[i] == 0)
      {
        for (int j = 1; j <= V; j++) VLabel[j] = -1;
        VLabel[i] = 0; Down = 1; Up = 1; Queue[Up++] = i;
        while (Down != Up)
          {
            int x = Queue[Down++];
            for (int p = 1; p <= G[x][0]; p++)
              {
                int y = G[x][p];
                if (Mate[y] == 0 && i != y)
                  {
                    Mate[y] = x; ReMatch(x, y);
                    Down = Up; break;
                  }
                if (VLabel[y] >= 0)
                  {
                    ReLabel(x, y);
                    continue;
                  }
                if (VLabel[Mate[y]] < 0)
                  {
                    VLabel[Mate[y]] = x;
                    Queue[Up++] = Mate[y];
                  }
              }
          }
      }
}

// Call this after Solve(). Returns number of edges in ↩
    matching (half the number of matched vertices)
int Size()
{
  int Count = 0;
  for (int i = 1; i <= V; i++)
    if (Mate[i] > i) Count++;
  return Count;
}
```

1.9 Minimum Edge Cover (Text)

If a minimum edge cover contains C edges, and a maximum matching contains M edges, then $C + M = |V|$. To obtain the edge cover, start with a maximum matching, and then, for every vertex not matched, just select some edge incident upon it and add it to the edge set.

1.10 Stable Marriage Problem (Gale–Shapley algorithm)

```
// Gale-Shapley algorithm for the stable marriage problem.
// madj[i][j] is the jth highest ranked woman for man i.
// fpref[i][j] is the rank woman i assigns to man j.
// Returns a pair of vectors (mpart, fpart), where mpart[i] ↩
    gives the partner of man i, and fpart is analogous
pair<vector<int>, vector<int> > stable_marriage(vector<↩
    vector<int> >& madj, vector<vector<int> >& fpref) {
    int n = madj.size();
    vector<int> mpart(n, -1), fpart(n, -1);
    vector<int> midx(n);
    queue<int> mfree;
    for (int i = 0; i < n; i++) {
        mfree.push(i);
    }
    while (!mfree.empty()) {
        int m = mfree.front(); mfree.pop();
        int f = madj[m][midx[m]++];
        if (fpart[f] == -1) {
            mpart[m] = f; fpart[f] = m;
        } else if (fpref[f][m] < fpref[f][fpart[f]]) {
            mpart[fpart[f]] = -1; mfree.push(fpart[f]);
            mpart[m] = f; fpart[f] = m;
        } else {
            mfree.push(m);
        }
    }
    return make_pair(mpart, fpart);
}
```

2 Graphs

2.1 Topological Sort

```
vi ts;

void dfs2(int u) {
    dfs_num[u] = VISITED;
    for (int j = 0; j < (int) AdjList[u].size(); j++) {
        pair<int,int> v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED) dfs2(v.first);
    }
    ts.push_back(u); }

// Inside int main()
ts.clear();
memset(dfs_num, UNVISITED, sizeof dfs_num);
for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED) dfs2(i);

for (int i = (int)tx.size() - 1; i >= 0; i--)
    printf(" %d" ts[i]);
```

2.2 Dijkstra

```
vi dist(V, INF); dist[s] = 0;
pq<ii, vector<ii>, greater<ii> > pq;pq.push(ii(0, s));

while (!pq.empty()) {
    ii front = pq.top(); pq.pop();
    int d = front.first, u = front.second;
    for (int j = 0; j < (int) AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (dist[u] + v.second < dist[v.first]) {
```

```
            dist[v.first] = dist[u] + v.second;
            pq.push(ii(dist[v.first], v.first));
}}}
```

2.3 Bellman Ford

```
// Bellman Ford routine
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)  // relax, overall O(VE)
  for (int u = 0; u < V; u++)    // these two loops = O(E)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];      // we can record SP ↩
           spanning here
      dist[v.first] = min(dist[v.first], dist[u] + v.second)↩
           ;// relax
    }

bool hasNegativeCycle = false;
for (int u = 0; u < V; u++)  // one more pass to check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dist[v.first] > dist[u] + v.second) // should be ↩
         false
      hasNegativeCycle = true; // true: negative cycle ↩
           exists!
  }
```

2.4 Floyd Warshall

```
for (int i = 0; i < V; i++) {
  for (int j = 0; j < V; j++)
    AdjMatrix[i][j] = INF;
  AdjMatrix[i][i] = 0;
}

for (int i = 0; i < E; i++) {
  scanf("%d %d %d", &u, &v, &w);
  AdjMatrix[u][v] = w; // directed graph
}

for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AdjMatrix[i][j] = min(AdjMatrix[i][j],
          AdjMatrix[i][k] + AdjMatrix[k][j]);
```

2.5 Prim's

```
vi taken;
priority_queue<ii> pq;

void process(int vtx) {
    taken[vtx] = 1;
    for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
        ii v = AdjList[vtx][j];
        if (!taken[v.first]) pq.push(ii(-v.second, -v.first)↩
             );
}   }

//main()
taken.assign(V, 0);
process(0);
mst_cost = 0;
while (!pq.empty()) {
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;
    if (!taken[u]) mst_cost += w, process(u);
}
```

2.6 Kruskal

```
// main()
vector< pair<int, ii> > EdgeList;
for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    EdgeList.push_back(make_pair(w, ii(u,v))); }
sort(EdgeList.begin(), EdgeList.end());

int mst_cost = 0;
UnionFind UF(V);

for (int i = 0; i < E; i++) {
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.↩
         second)) {
        mst_cost += front.first;
        UF.unionSet(front.second.first, front.second.second)↩
             ;
}   }
```

2.7 Strongly connected components

```
struct SCC {
    int V, group_cnt;
    vector<vector<int> > adj, radj;
    vector<int> group_num, vis;
    stack<int> stk;

    // V = number of vertices
    SCC(int V): V(V), group_cnt(0), group_num(V), vis(V), ↩
         adj(V), radj(V) {}

    // Call this to add an edge (0-based)
    void add_edge(int v1, int v2) {
        adj[v1].push_back(v2);
        radj[v2].push_back(v1);
    }

    void fill_forward(int x) {
        vis[x] = true;
        for (int i = 0; i < adj[x].size(); i++) {
            if (!vis[adj[x][i]]) {
                fill_forward(adj[x][i]);
            }
        }
        stk.push(x);
    }

    void fill_backward(int x) {
        vis[x] = false;
        group_num[x] = group_cnt;
        for (int i = 0; i < radj[x].size(); i++) {
            if (vis[radj[x][i]]) {
                fill_backward(radj[x][i]);
            }
        }
    }

    // Returns number of strongly connected components.
    // After this is called, group_num contains component ↩
         assignments (0-based)
    int get_scc() {
        for (int i = 0; i < V; i++) {
            if (!vis[i]) fill_forward(i);
        }
        group_cnt = 0;
        while (!stk.empty()) {
            if (vis[stk.top()]) {
                fill_backward(stk.top());
                group_cnt++;
            }
            stk.pop();
        }
        return group_cnt;
    }
};
```

2.8 Bridges

```
// Finds bridges and cut vertices
// Receives:
// N: number of vertices
// l: adjacency list
// Gives:
// vis, seen, par (used to find cut vertices)
// ap - 1 if it is a cut vertex, 0 otherwise
// brid - vector of pairs containing the bridges
typedef pair<int, int> PII;

int N;
vector <int> l[MAX];
vector <PII> brid;
int vis[MAX], seen[MAX], par[MAX], ap[MAX];
int cnt, root;

void dfs(int x){
  if(vis[x] != -1)
    return;
  vis[x] = seen[x] = cnt++;

  int adj = 0;
  for(int i = 0; i < (int)l[x].size(); i++){
    int v = l[x][i];
    if(par[x] == v)
      continue;
    if(vis[v] == -1){
      adj++;
      par[v] = x;
      dfs(v);
      seen[x] = min(seen[x], seen[v]);
      if(seen[v] >= vis[x] && x != root)
        ap[x] = 1;
      if(seen[v] == vis[v])
        brid.push_back(make_pair(v, x));
    }
    else{
      seen[x] = min(seen[x], vis[v]);
      seen[v] = min(seen[x], seen[v]);
    }
```

```
  }
  if(x == root) ap[x] = (adj>1);
}

void bridges(){
  brid.clear();
  for(int i = 0; i < N; i++){
    vis[i] = seen[i] = par[i] = -1;
    ap[i] = 0;
  }
  cnt = 0;
  for(int i = 0; i < N; i++)
    if(vis[i] == -1){
      root = i;
      dfs(i);
    }
}
```

2.9 Eulerian path

```
// Eulerian path/circuit in an undirected graph. TODO: Does this handle self-edges?
// NOTE(Brian): This looks like it could theoretically degrade to quadratic time in, say, a graph where we keep going back and forth between two vertices; in this case a lot of time could be wasted searching for an unused edge.
struct EulerianPath {
    int n;
    vector<vector<int> > adj;
    vector<pair<int, int> > edges;
    vector<int> valid;
    vector<int> circuit;

    EulerianPath(int n): n(n), adj(n) {}

    // Call this to construct the graph.
    // Edges are zero-based and undirected (only add each edge once!)
    void add_edge(int x, int y) {
        adj[x].push_back(edges.size());
        adj[y].push_back(edges.size());
        edges.push_back(make_pair(x, y));
        valid.push_back(1);
    }

    void find_path(int x){
      for(int i = 0; i < adj[x].size(); i++){
        int e = adj[x][i];
        if(!valid[e]) continue;
        int v = edges[e].first;
        if(v == x) v = edges[e].second;
        valid[e] = 0;
        find_path(v);
      }
      circuit.push_back(x);
    }

    // Call this to find the path/circuit (autodetects)
    // Returns the path/circuit itself in "circuit" variable
    // Initial node is repeated at end if it's a circuit.
    void find_euler_path() {
      circuit.clear();
      //supposes graph is connected and has correct degree
      for(int i = 0; i < n; i++)
        if(adj[i].size()%2){
          find_path(i);
          return;
        }
      find_path(0);
    }
};
```

2.10 Lowest Common Ancestor (Pseudocode)

Tarjan's offline algorithm (requires $O(N)$ disjoint set operations).

```
function TarjanOLCA(u)
    MakeSet(u);
    u.ancestor := u;
    for each v in u.children do
        TarjanOLCA(v);
        Union(u,v);
        Find(u).ancestor := u;
    u.colour := black;
    for each v such that {u,v} in P do
        if v.colour == black
            print "Tarjan's Least Common Ancestor of " + u +
                " and " + v + " is " + Find(v).ancestor + ".";
```

This function is called on the root of the tree. The set P of pairs of nodes to query must be specified in advance. Each node is initially white, and is colored black after it and all its children have been visited. The lowest common ancestor of the pair u,v is available as Find(v).ancestor immediately (and only immediately) after u is colored black, provided v is already black. Otherwise, it will be available later as Find(u).ancestor, immediately after v is colored black.

3 Data Structures

3.1 UFDS

```
int find(int i) { return (p[i] == i) ? i : (p[i] = find(p[i])); }

int join(int i, int j) {
    int a = find(i), b = find(j);
    if (a != b) {
        if (rank[a] > rank[b]) p[b] = a;
        else { p[a] = b;
               if (rank[a] == rank[b]) rank[b]++; }
    }
}
```

3.2 Suffix arrays

```
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};

int main() {

  // bobocel is the 0'th suffix
  //  obocel is the 5'th suffix
  //   bocel is the 1'st suffix
  SuffixArray suffix("bobocel");
  vector<int> v = suffix.GetSuffixArray();

  // Expected output: 0 5 1 6 2 3 4
  //                  2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```

3.3 Fenwick Tree (short hand)

```
class FenwickTree {
    private: vi ft;
    public: FenwickTree(int n) { ft.assign(n+1, 0); }
    int rsq(int b) { // Returns RSQ(1, b)
        int sum=0; for (;b; b-=LSOne(b)) sum += ft[b];
        return sum; // LSOne(S) (S & (-S)) }
    int rsq(int a, int b) { rsq(b)-(a==1 ? 0:rsq(a-1)) }
    void adjust(int k, int v) {
        for (;k<(int)ft.size(); k+=LSOne(k)) ft[k] += v; }
}
```

3.4 Binary Indexed Tree (BIT)

```
// Binary indexed tree supporting binary search.
struct BIT {
    int n;
    vector<int> bit;
    // BIT can be thought of as having entries f[1], ..., f[n]
    // which are 0-initialized
    BIT(int n):n(n), bit(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    int read(int idx) {
        idx--;
        int res = 0;
        while (idx > 0) {
            res += bit[idx];
            idx -= idx & -idx;
        }
        return res;
    }
    // returns f[idx1] + ... + f[idx2-1]
    // precondition idx1 <= idx2 <= n+1
    int read2(int idx1, int idx2) {
        return read(idx2) - read(idx1);
    }
    // adds val to f[idx]
    // precondition 1 <= idx <= n (there is no element 0!)
    void update(int idx, int val) {
        while (idx <= n) {
            bit[idx] += val;
            idx += idx & -idx;
        }
    }
    // returns smallest positive idx such that read(idx) >= target
    int lower_bound(int target) {
        if (target <= 0) return 1;
        int pwr = 1; while (2*pwr <= n) pwr*=2;
        int idx = 0; int tot = 0;
        for (; pwr; pwr >>= 1) {
            if (idx+pwr > n) continue;
            if (tot + bit[idx+pwr] < target) {
                tot += bit[idx+=pwr];
            }
        }
        return idx+2;
    }
    // returns smallest positive idx such that read(idx) > target
    int upper_bound(int target) {
        if (target < 0) return 1;
        int pwr = 1; while (2*pwr <= n) pwr*=2;
        int idx = 0; int tot = 0;
        for (; pwr; pwr >>= 1) {
            if (idx+pwr > n) continue;
            if (tot + bit[idx+pwr] <= target) {
                tot += bit[idx+=pwr];
            }
        }
        return idx+2;
    }
};
```

3.5 BIT with range updates

```
// BIT with range updates, inspired by Petr Mitrichev
struct BIT {
    int n;
    vector<int> slope;
    vector<int> intercept;
    // BIT can be thought of as having entries f[1], ..., f[n]
    // which are 0-initialized
    BIT(int n): n(n), slope(n+1), intercept(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    int query(int idx) {
        int m = 0, b = 0;
        for (int i = idx-1; i > 0; i -= i&-i) {
            m += slope[i];
            b += intercept[i];
        }
        return m*idx + b;
    }
    // adds amt to f[i] for i in [idx1, idx2)
    // precondition 1 <= idx1 <= idx2 <= n+1 (you can't update element 0)
    void update(int idx1, int idx2, int amt) {
        for (int i = idx1; i <= n; i += i&-i) {
            slope[i] += amt;
            intercept[i] -= idx1*amt;
        }
        for (int i = idx2; i <= n; i += i&-i) {
            slope[i] -= amt;
            intercept[i] += idx2*amt;
        }
    }
};
```

3.6 Segment Tree with Lazy Propagation

```
// This is set up for range minimum queries, but can be easily adapted for computing other quantities.
// To enable lazy propagation and range updates, uncomment the following line.
// #define LAZY
struct Segtree {
    int n;
    vector<int> data;
#ifdef LAZY
#define NOLAZY 2e9
#define GET(node) (lazy[node] == NOLAZY ? data[node] : lazy[node])
    vector<int> lazy;
#else
#define GET(node) data[node]
#endif
    void build_rec(int node, int* begin, int* end) {
        if (end == begin+1) {
            if (data.size() <= node) data.resize(node+1);
            data[node] = *begin;
        } else {
            int* mid = begin + (end-begin+1)/2;
            build_rec(2*node+1, begin, mid);
            build_rec(2*node+2, mid, end);
            data[node] = min(data[2*node+1], data[2*node+2]);
        }
    }
#ifndef LAZY
    void update_rec(int node, int begin, int end, int pos, int val) {
        if (end == begin+1) {
            data[node] = val;
        } else {
            int mid = begin + (end-begin+1)/2;
            if (pos < mid) {
                update_rec(2*node+1, begin, mid, pos, val);
            } else {
                update_rec(2*node+2, mid, end, pos, val);
            }
            data[node] = min(data[2*node+1], data[2*node+2]);
        }
    }
#else
    void update_range_rec(int node, int tbegin, int tend, int abegin, int aend, int val) {
        if (tbegin >= abegin && tend <= aend) {
            lazy[node] = val;
        } else {
            int mid = tbegin + (tend - tbegin + 1)/2;
            if (lazy[node] != NOLAZY) {
                lazy[2*node+1] = lazy[2*node+2] = lazy[node]; lazy[node] = NOLAZY;
            }
            if (mid > abegin && tbegin < aend)
                update_range_rec(2*node+1, tbegin, mid, abegin, aend, val);
            if (tend > abegin && mid < aend)
                update_range_rec(2*node+2, mid, tend, abegin, aend, val);
            data[node] = min(GET(2*node+1), GET(2*node+2));
        }
    }
#endif
    int query_rec(int node, int tbegin, int tend, int abegin, int aend) {
        if (tbegin >= abegin && tend <= aend) {
            return GET(node);
        } else {
#ifdef LAZY
            if (lazy[node] != NOLAZY) {
                data[node] = lazy[2*node+1] = lazy[2*node+2] = lazy[node]; lazy[node] = NOLAZY;
            }
#endif
            int mid = tbegin + (tend - tbegin + 1)/2;
            int res = INT_MAX;
            if (mid > abegin && tbegin < aend)
                res = min(res, query_rec(2*node+1, tbegin, mid, abegin, aend));
            if (tend > abegin && mid < aend)
                res = min(res, query_rec(2*node+2, mid, tend, abegin, aend));
            return res;
        }
    }
```

```
    // Create a segtree which stores the range [begin, end) in its bottommost level.
    Segtree(int* begin, int* end): n(end - begin) {
        build_rec(0, begin, end);
#ifdef LAZY
        lazy.assign(data.size(), NOLAZY);
#endif
    }

#ifndef LAZY
    // Call this to update a value (indices are 0-based). If lazy propagation is enabled, use update_range(pos, pos+1, val) instaed.
    void update(int pos, int val) {
        update_rec(0, 0, n, pos, val);
    }
#else
    // Call this to update range [begin, end), if lazy propagation is enabled. Indices are 0-based.
    void update_range(int begin, int end, int val) {
        update_range_rec(0, 0, n, begin, end, val);
    }
#endif
    // Returns minimum in range [begin, end). Indices are 0-based.
    int query(int begin, int end) {
        return query_rec(0, 0, n, begin, end);
    }
};
```

3.7 Size-Balanced Tree

```
// Size-Balanced Tree, taken from http://blog.tomtung.com/2007/06/size-balanced-tree-in-cpp
// TODO: How does this behave with duplicate keys?
struct SBTNode{
    SBTNode *ch[2],*p;
    long key;
    unsigned long size;
    SBTNode(long _key,unsigned long _size);
}NIL=SBTNode(0,0);
// To make an empty tree: SBTree t = &NIL;
typedef SBTNode *SBTree;
SBTNode::SBTNode(long _key,unsigned long _size=1){
    ch[0]=ch[1]=p=&NIL;
    size=_size;
    key=_key;
}

// Returns node if key is found, &NIL otherwise
SBTNode *SBT_Search(SBTree T,long key);

// Example invocation: SBT_Insert(t, new SBTNode(42));
void SBT_Insert(SBTree &T, SBTNode* x);

// Careful, if no matching key is found, this will still delete a node from the tree.
// The deleted node is not freed, and a pointer to it is returned.
SBTNode *SBT_Delete(SBTree &T, long key);

// Returns first node whose key is strictly less than key, or &NIL if no such node
SBTNode *SBT_Pred(SBTree T, long key);

// Returns first node whose key is strictly greater than key, or &NIL if no such node
SBTNode *SBT_Succ(SBTree T,long key);

// Returns node with ith smallest value (1 <= i <= T->size)
SBTNode *SBT_Select(SBTree T, unsigned long i);

// Returns k such that key is the kth smallest key in the tree (1 <= k <= T->size), or 0 if key not found.
unsigned long SBT_Rank(SBTree T, long key);

inline void SBT_Rotate(SBTree &x,bool flag){
    SBTNode *y=x->ch[!flag];
    // assert(x!=&NIL&&y!=&NIL);
    y->p=x->p;
    x->p=y;
    if(y->ch[flag]!=&NIL) y->ch[flag]->p=x;
    x->ch[!flag]=y->ch[flag];
    y->ch[flag]=x;
    y->size=x->size;
    x->size=x->ch[0]->size+x->ch[1]->size+1;
    x=y;
}
void SBT_Maintain(SBTree &T,bool flag){
    if(T->ch[flag]->ch[flag]->size>T->ch[!flag]->size)
        SBT_Rotate(T,!flag);
    else if(T->ch[flag]->ch[!flag]->size>T->ch[!flag]->size){
        SBT_Rotate(T->ch[flag],flag);
        SBT_Rotate(T,!flag);
    }
    else return;
    SBT_Maintain(T->ch[0],0),SBT_Maintain(T->ch[1],1);
    SBT_Maintain(T,0),SBT_Maintain(T,1);
}

SBTNode *SBT_Search(SBTree T,long key){
    return T==&NIL||T->key==key?T:SBT_Search(T->ch[key>T->key],key);
}

void SBT_Insert(SBTree &T, SBTNode* x){
    if(T==&NIL)    T=x;
    else{
        T->size++;
        x->p=T;
        SBT_Insert(T->ch[x->key>T->key],x);
        SBT_Maintain(T,x->key>T->key);
    }
}

SBTNode *SBT_Delete(SBTree &T, long key){
    if(T==&NIL)    return &NIL;
    T->size--;
    if(T->key==key||T->ch[key>T->key]==&NIL){
        SBTNode *toDel;
        if(T->ch[0]==&NIL||T->ch[1]==&NIL){
            toDel=T;
            T=T->ch[T->ch[1]!=&NIL];
            if(T!=&NIL)    T->p=toDel->p;
        }else{
            toDel=SBT_Delete(T->ch[1],key-1);
            T->key=toDel->key;
        }
        return toDel;
    }
    else return SBT_Delete(T->ch[key>T->key],key);
}

SBTNode *SBT_Pred(SBTree T, long key){
    if(T==&NIL)    return &NIL;
    if(key<=T->key)    return SBT_Pred(T->ch[0],key);
    else{
        SBTNode *pred=SBT_Pred(T->ch[1],key);
        return (pred!=&NIL?pred:T);
    }
}

SBTNode *SBT_Succ(SBTree T,long key){
    if(T==&NIL)    return &NIL;
    if(key>=T->key)    return SBT_Succ(T->ch[1],key);
    else{
        SBTNode *succ= SBT_Succ(T->ch[0],key);
        return(succ!=&NIL?succ:T);
    }
}

SBTNode *SBT_Select(SBTree T, unsigned long i){
    unsigned long r = T->ch[0]->size+1;
    if(i==r)    return T;
    else return SBT_Select(T->ch[i>r],i>r?i-r:i);
}

unsigned long SBT_Rank(SBTree T, long key){
    if(T==&NIL)    return 0;
    if(T->key==key)    return T->ch[0]->size+1;
    else if(key<T->key)    return SBT_Rank(T->ch[0],key);
    else{
        unsigned long r=SBT_Rank(T->ch[1],key);
        return r==0?0:r+T->ch[0]->size+1;
    }
}
```

4 Geometry

4.1 Heron's formula

$S = \frac{A+B+C}{2} \qquad A = \sqrt{S(S-A)(S-B)(S-C)}$

4.2 Miscellaneous Geometry

```
// 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
// if the projection doesn't lie on the segment, returns closest vertex
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)));
}

// 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);
}

// determine if c and d are on same side of line passing through a and b
bool OnSameSide(PT a, PT b, PT c, PT d) {
  return cross(c-a, c-b) * cross(d-a, d-b) > 0;
}

// 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;
  }
  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) {
  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;
}
```

4.3 3D Geometry

```
#define LINE 0
#define SEGMENT 1
#define RAY 2

struct point{
    double x, y, z;
    point(){};
    point(double _x, double _y, double _z){ x=_x; y=_y; z=_z; }
    point operator+ (point p) { return point(x+p.x, y+p.y, z+p.z); }
    point operator- (point p) { return point(x-p.x, y-p.y, z-p.z); }
    point operator* (double c) { return point(x*c, y*c, z*c); }
};

double dot(point a, point b){
    return a.x*b.x + a.y*b.y + a.z*b.z;
}

point cross(point a, point b) {
```

```
    return point(a.y*b.z-a.z*b.y,
                 a.z*b.x-a.x*b.z,
                 a.x*b.y-a.y*b.x);
}

double distSq(point a, point b){
    return dot(a-b, a-b);
}

// compute a, b, c, d such that all points lie on ax + by + cz = d. TODO: test this
double planeFromPts(point p1, point p2, point p3, double& a, double& b, double& c, double& d) {
    point normal = cross(p2-p1, p3-p1);
    a = normal.x; b = normal.y; c = normal.z;
    d = -a*p1.x-b*p1.y-c*p1.z;
}

// project point onto plane. TODO: test this
point ptPlaneProj(point p, double a, double b, double c, double d) {
    double l = (a*p.x+b*p.y+c*p.z+d)/(a*a+b*b+c*c);
    return point(p.x-a*l, p.y-b*l, p.z-c*l);
}

// distance from point p to plane aX + bY + cZ + d = 0
double ptPlaneDist(point p, double a, double b, double c, double d){
    return fabs(a*p.x + b*p.y + c*p.z + d) / sqrt(a*a + b*b + c*c);
}

// distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0
double planePlaneDist(double a, double b, double c, double d1, double d2){
    return fabs(d1 - d2) / sqrt(a*a + b*b + c*c);
}

// square distance between point and line, ray or segment
double ptLineDistSq(point s1, point s2, point p, int type){
    double pd2 = distSq(s1, s2);
    point r;
    if(pd2 == 0)
    r = s1;
    else{
    double u = dot(p-s1, s2-s1) / pd2;
    r = s1 + (s2 - s1)*u;
    if(type != LINE && u < 0.0)
        r = s1;
    if(type == SEGMENT && u > 1.0)
        r = s2;
    }
    return distSq(r, p);
}

// Distance between lines ab and cd. TODO: Test this
double lineLineDistance(point a, point b, point c, point d) {
    point v1 = b-a;
    point v2 = d-c;
    point cr = cross(v1, v2);
    if (dot(cr, cr) < EPS) {
        point proj = v1*(dot(v1, c-a)/dot(v1, v1));
        return sqrt(dot(c-a-proj, c-a-proj));
    } else {
        point n = cr/sqrt(dot(cr, cr));
        point p = dot(n, c - a);
        return sqrt(dot(p, p));
    }
}

// Distance between line segments ab and cd (translated from Java)
double segmentSegmentDistance(point a, point b, point c, point d) {
    point u = b - a, v = d - c, w = a - c;
    double a = dot(u, u), b = dot(u, v), c = dot(v, v), d = dot(u, w), e = dot(v, w);
    double D = a*c-b*b;
    double sc, sN, sD = D;
    double tc, tN, tD = D;

    // compute the line parameters of the two closest points
    if (D < EPS) { // the lines are almost parallel
        sN = 0.0;        // force using point P0 on segment S1
        sD = 1.0;        // to prevent possible division by 0.0 later
        tN = e;
        tD = c;
    } else {                // get the closest points on the infinite lines
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0.0) {       // sc < 0 => the s=0 edge is visible
            sN = 0.0;
            tN = e;
            tD = c;
        }
        else if (sN > sD) {  // sc > 1 => the s=1 edge is visible
            sN = sD;
            tN = e + b;
            tD = c;
        }
    }

    if (tN < 0.0) {           // tc < 0 => the t=0 edge is visible
        tN = 0.0;
        // recompute sc for this edge
        if (-d < 0.0)
            sN = 0.0;
        else if (-d > a)
            sN = sD;
        else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {      // tc > 1 => the t=1 edge is visible
        tN = tD;
        // recompute sc for this edge
        if ((-d + b) < 0.0)
            sN = 0;
        else if ((-d + b) > a)
            sN = sD;
        else {
            sN = (-d + b);
            sD = a;
        }
    }
    // finally do the division to get sc and tc
    sc = (abs(sN) < EPS ? 0.0 : sN / sD);
    tc = (abs(tN) < EPS ? 0.0 : tN / tD);

    // get the difference of the two closest points
    point dP = w + (sc * u) - (tc * v);  // = S1(sc) - S2(tc)
    return sqrt(dot(dP, dP));   // return the closest distance
}

double signedTetrahedronVol(point A, point B, point C, point D) {
    double A11 = A.x - B.x;
    double A12 = A.x - C.x;
    double A13 = A.x - D.x;
    double A21 = A.y - B.y;
    double A22 = A.y - C.y;
    double A23 = A.y - D.y;
    double A31 = A.z - B.z;
    double A32 = A.z - C.z;
    double A33 = A.z - D.z;
    double det =
        A11*A22*A33 + A12*A23*A31 +
        A13*A21*A32 - A11*A23*A32 -
        A12*A21*A33 - A13*A22*A31;
    return det / 6;
}

// Parameter is a vector of vectors of points - each interior vector
// represents the 3 points that make up 1 face, in any order.
// Note: The polyhedron must be convex, with all faces given as triangles.
double polyhedronVol(vector<vector<point> > poly) {
    int i,j;
    point cent(0,0,0);
    for (i=0; i<poly.size(); i++)
        for (j=0; j<3; j++)
            cent=cent+poly[i][j];
    cent=cent*(1.0/(poly.size()*3));
    double v=0;
    for (i=0; i<poly.size(); i++)
        v+=fabs(signedTetrahedronVol(cent,poly[i][0],poly[i][1],poly[i][2]));
    return v;
}
```

4.4 Convex hull

```
// O(N log N) Monotone Chains algorithm for 2d convex hull.
// Gives the hull in counterclockwise order from the leftmost point, which is repeated at the end. Minimizes the number of points on the hull when collinear points exist.
long long cross(pair<int, int> A, pair<int, int> B, pair<int, int> C) {
    return (B.first - A.first)*(C.second - A.second)
         - (B.second - A.second)*(C.first - A.first);
}
// The hull is returned in param "hull"
void convex_hull(vector<pair<int, int> > pts, vector<pair<int, int> >& hull) {
    hull.clear(); sort(pts.begin(), pts.end());
    for (int i = 0; i < pts.size(); i++) {
        while (hull.size() >= 2 && cross(hull[hull.size()-2], hull.back(), pts[i]) <= 0) {
            hull.pop_back();
        }
        hull.push_back(pts[i]);
    }
    int s = hull.size();
    for (int i = pts.size()-2; i >= 0; i--) {
        while (hull.size() >= s+1 && cross(hull[hull.size()-2], hull.back(), pts[i]) <= 0) {
            hull.pop_back();
        }
        hull.push_back(pts[i]);
    }
}
```

4.5 Slow Delaunay triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m triples of indices
//                     corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
        int n = x.size();
        vector<T> z(n);
        vector<triple> ret;

        for (int i = 0; i < n; i++)
            z[i] = x[i] * x[i] + y[i] * y[i];

        for (int i = 0; i < n-2; i++) {
            for (int j = i+1; j < n; j++) {
                for (int k = i+1; k < n; k++) {
                    if (j == k) continue;
                    double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
                    double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
                    double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
                    bool flag = zn < 0;
                    for (int m = 0; flag && m < n; m++)
                        flag = flag && ((x[m]-x[i])*xn +
                                        (y[m]-y[i])*yn +
                                        (z[m]-z[i])*zn <= 0);
                    if (flag) ret.push_back(triple(i, j, k));
                }
            }
        }
        return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

4.6 Minimum Enclosing Disk (Welzl's Algorithm)

```
// Minimum enclosing circle, Welzl's algorithm
// Expected linear time.
// If there are any duplicate points in the input, be sure to remove them first.
struct point {
    double x;
    double y;
};
struct circle {
    double x;
    double y;
    double r;
    circle() {}
    circle(double x, double y, double r): x(x), y(y), r(r) {}
};
circle b_md(vector<point> R) {
    if (R.size() == 0) {
        return circle(0, 0, -1);
    } else if (R.size() == 1) {
        return circle(R[0].x, R[0].y, 0);
    } else if (R.size() == 2) {
        return circle((R[0].x+R[1].x)/2.0,
                      (R[0].y+R[1].y)/2.0,
                      hypot(R[0].x-R[1].x, R[0].y-R[1].y)/2.0);
    } else {
        double D = (R[0].x - R[2].x)*(R[1].y - R[2].y) - (R[1].x - R[2].x)*(R[0].y - R[2].y);
        double p0 = (((R[0].x - R[2].x)*(R[0].x + R[2].x) + (R[0].y - R[2].y)*(R[0].y + R[2].y)) / 2 * (R[1].y - R[2].y) - ((R[1].x - R[2].x)*(R[1].x + R[2].x) + (R[1].y - R[2].y)*(R[1].y + R[2].y)) / 2 * (R[0].y - R[2].y))/D;
        double p1 = (((R[1].x - R[2].x)*(R[1].x + R[2].x) + (R[1].y - R[2].y)*(R[1].y + R[2].y)) / 2 * (R[0].x - R[2].x) - ((R[0].x - R[2].x)*(R[0].x + R[2].x) + (R[0].y - R[2].y)*(R[0].y + R[2].y)) / 2 * (R[1].x - R[2].x))/D;
        return circle(p0, p1, hypot(R[0].x - p0, R[0].y - p1));
    }
}
circle b_minidisk(vector<point>& P, int i, vector<point> R) {
    if (i == P.size() || R.size() == 3) {
        return b_md(R);
    } else {
        circle D = b_minidisk(P, i+1, R);
        if (hypot(P[i].x-D.x, P[i].y-D.y) > D.r) {
            R.push_back(P[i]);
            D = b_minidisk(P, i+1, R);
        }
        return D;
    }
}

// Call this function.
circle minidisk(vector<point> P) {
    random_shuffle(P.begin(), P.end());
    return b_minidisk(P, 0, vector<point>());
}
```

4.7 Pick's Theorem (Text)

For a polygon with all vertices on lattice points, $A = i + b/2 - 1$, where A is the area, i is the number of lattice points strictly within the polygon, and b is the number of lattice points on the boundary of the polygon. (Note, there is no generalization to higher dimensions)

5 Math Algorithms

5.1 Sieve of Eratosthenes

```
// This is the famous "Yarin sieve", for use when memory is tight.
#define MAXSIEVE 100000000 // All prime numbers up to this
#define MAXSIEVEHALF (MAXSIEVE/2)
#define MAXSQRT 5000 // sqrt(MAXSIEVE)/2
char a[MAXSIEVE/16+2];
#define isprime(n) (a[(n)>>4]&(1<<(((n)>>1)&7))) // Works when n is odd

int i,j;
memset(a,255,sizeof(a));
a[0]=0xFE;
for(i=1;i<MAXSQRT;i++)
if (a[i>>3]&(1<<(i&7)))
for(j=i+i+i+1;j<MAXSIEVEHALF;j+=i+i+1)
a[j>>3]&=~(1<<(j&7));
```

5.2 Primes

```
ll _sieve_size;
bitset<10000010> bs; // #include <bitset>
vi primes;

void sieve(ll upperbound) {
  _sieve_size = upperbound + 1;
  bs.set();
  bs[0] = bs[1] = 0;
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i);
} }

bool isPrime(ll N) {
  if (N <= _sieve_size) return bs[N];
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true;
} // note: only work for N <= (last prime in vi "primes")^2

vi primeFactors(ll N) {// remember: vi is vector of integers, ll is long long
  vi factors;// vi 'primes' is optional
  ll PF_idx = 0, PF = primes[PF_idx];
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }
    PF = primes[++PF_idx];
```

```
  }
  if (N != 1) factors.push_back(N);
  return factors;
}

ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans++;
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans += N;
  return ans;
}
// 1, 2, 5, 10, 25, 50, 6 divisors
ll numOfDivisors(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0; // count the power
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;
  return ans;
}
// N=50 : 1 + 2 + 5 + 10 + 25 + 50 = 93
ll sumOfDivisors(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1)↩
        ;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1)↩
      ;
  return ans;
}
//  20 integers < 50 are relatively prime with 50
ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans -= ans / PF;
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans -= ans / N;r
  return ans;
}
```

5.3 Modular arithmetic and linear Diophantine solver

```
// This is a collection of useful code for solving problems ↩
    that
// involve modular linear equations.  Note that all of the
// algorithms described here work on nonnegative integers.

typedef vector<int> VI;
typedef pair<int,int> PII;

// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
  return a/gcd(a,b)*b;
}

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on ↩
    failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}

// Chinese remainder theorem (special case): find z such ↩
    that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x↩
    ,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[i]).  Return (z,M).  On
// failure, M = -1.  Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.first, ret.second, x↩
        [i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y↩
    =-1
void linear_diophantine(int a, int b, int c, int &x, int &y)↩
    {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

5.4 Gaussian elimination for square matrices of full rank; finds inverses and determinants

```
// Gauss-Jordan elimination with full pivoting.
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//           A MUST BE INVERTIBLE!
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
```

```
      if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { return 0; }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}
```

5.5 Reduced row echelon form (RREF), matrix rank

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
  int n = a.size();
  int m = a[0].size();
  int r = 0;
  for (int c = 0; c < m; c++) {
    int j = r;
    for (int i = r+1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);

    T s = 1.0 / a[r][c];
    for (int j = 0; j < m; j++) a[r][j] *= s;
    for (int i = 0; i < n; i++) if (i != r) {
      T t = a[i][c];
      for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}
```

5.6 Solving linear systems (Text)

To solve a general system of linear equations, put it into matrix form and compute the reduced row echelon form. For example,

$$2x + y = 5$$
$$3x + 2y = 6$$

corresponds to the matrix

$$\left[\begin{array}{cc|c} 2 & 1 & 5 \\ 3 & 2 & 6 \end{array}\right]$$

with RREF

$$\left[\begin{array}{cc|c} 1 & 0 & 4 \\ 0 & 1 & -3 \end{array}\right]$$

After row reduction, if any row has a 1 in the rightmost column and 0 everywhere else, then the system is inconsistent and has no solution. Otherwise, to find a solution, set the variable corresponding to the leftmost 1 in each column equal to the corresponding value in the rightmost column, and set all other variables to 0. Ignore rows consisting entirely of 0. The solution is unique iff the rank of the matrix equals the number of variables.

5.7 Fast Fourier transform (FFT)

```
struct cpx
{
  cpx(){}
  cpx(double aa):a(aa){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const
  {
    return a * a + b * b;
  }
  cpx bar(void) const
  {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b)
{
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b)
{
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b)
{
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta)
{
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST BE A POWER OF 2}
// dir:    either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j * k / size)
void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
  if(size < 1) return;
  if(size == 1)
  {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++)
  {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
  }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass dir = 1 as the argument).
//   2. Get H by element-wise multiplying F and G.
//   3. Get h by taking the inverse FFT (use dir = -1 as the argument)
//      and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.
// To compute an *acyclic* convolution, pad f and g to the right with zeroes.
```

5.8 Simplex algorithm

```
// Two-phase simplex algorithm for solving linear programs of the form
```

```
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
        }
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
         D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1;
         D[i][n+1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j];
          }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    DOUBLE inv = 1.0 / D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r)
      for (int j = 0; j < n+2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] * inv;
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] *= inv
         ;
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] *= -
         inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][
             s] && N[j] < N[s]) s = j;
      }
      if (s < 0 || D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r
             ][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[
                 i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r
         = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return -
           numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i
               ][s] && N[j] < N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity
         ();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i
         ][n+1];
    return D[m][n+1];
  }
};
```

5.9 Fast factorization (Pollard rho) and primality testing (Rabin–Miller)

```
typedef long long unsigned int llui;
typedef long long int lli;
typedef long double float64;

llui mul_mod(llui a, llui b, llui m){
   llui y = (llui)((float64)a*(float64)b/m+(float64)1/2);
   y = y * m;
   llui x = a * b;
   llui r = x - y;
   if ( (lli)r < 0 ){
      r = r + m; y = y - 1;
   }
   return r;
}

llui C,a,b;
llui gcd(){
   llui c;
   if(a>b){
      c = a; a = b; b = c;
   }
   while(1){
      if(a == 1LL) return 1LL;
      if(a == 0 || a == b) return b;
      c = a; a = b%a;
      b = c;
   }
}

llui f(llui a, llui b){
   llui tmp;
   tmp = mul_mod(a,a,b);
   tmp+=C; tmp%=b;
   return tmp;
}

llui pollard(llui n){
   if(!(n&1)) return 2;
   C=0;
   llui iteracoes = 0;
   while(iteracoes <= 1000){
      llui x,y,d;
      x = y = 2; d = 1;
      while(d == 1){
         x = f(x,n);
         y = f(f(y,n),n);
         llui m = (x>y)?(x-y):(y-x);
         a = m; b = n; d = gcd();
      }
      if(d != n)
         return d;
      iteracoes++; C = rand();
   }
   return 1;
}

llui pot(llui a, llui b, llui c){
   if(b == 0) return 1;
   if(b == 1) return a%c;
   llui resp = pot(a,b>>1,c);
   resp = mul_mod(resp,resp,c);
   if(b&1)
      resp = mul_mod(resp,a,c);
   return resp;
}

// Rabin-Miller primality testing algorithm
bool isPrime(llui n){
   llui d = n-1;
   llui s = 0;
   if(n <=3 || n == 5) return true;
   if(!(n&1)) return false;
   while(!(d&1)){ s++; d>>=1; }
   for(llui i = 0;i<32;i++){
      llui a = rand();
      a <<=32;
      a+=rand();
      a%=(n-3); a+=2;
      llui x = pot(a,d,n);
      if(x == 1 || x == n-1) continue;
      for(llui j = 1;j<= s-1;j++){
         x = mul_mod(x,x,n);
         if(x == 1) return false;
         if(x == n-1)break;
      }
      if(x != n-1) return false;
   }
   return true;
}
map<llui,int> factors;
// Precondition: factors is an empty map, n is a positive integer
// Postcondition: factors[p] is the exponent of p in prime factorization of n
void fact(llui n){
   if(!isPrime(n)){
      llui fac = pollard(n);
      fact(n/fac); fact(fac);
   }else{
      map<llui,int>::iterator it;
      it = factors.find(n);
      if(it != factors.end()){
         (*it).second++;
      }else{
         factors[n] = 1;
      }
   }
}
```

5.10 Euler's Totient

```
// This code took less than 0.5s to calculate with MAX = 10^7
#define MAX 10000000

int phi[MAX];
bool pr[MAX];

void totient(){
  for(int i = 0; i < MAX; i++){
    phi[i] = i;
    pr[i] = true;
  }
  for(int i = 2; i < MAX; i++)
    if(pr[i]){
      for(int j = i; j < MAX; j+=i){
        pr[j] = false;
        phi[j] = phi[j] - (phi[j] / i);
      }
      pr[i] = true;
    }
}
```

6 Number Theory Reference

6.1 Polynomial Coefficients (Text)

$(x_1+x_2+...+x_k)^n = \sum_{c_1+c_2+...+c_k=n} \frac{n!}{c_1!c_2!...c_k!} x_1^{c_1} x_2^{c_2} ... x_k^{c_k}$

6.2 Möbius Function (Text)

$$\mu(n) = \begin{cases} 0 & n \text{ not squarefree} \\ 1 & n \text{ squarefree w/ even no. of prime factors} \\ 1 & n \text{ squarefree w/ odd no. of prime factors} \end{cases}$$

Note that $\mu(a)\mu(b) = \mu(ab)$ for a, b relatively prime Also $\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$

Möbius Inversion If $g(n) = \sum_{d|n} f(d)$ for all $n \geq 1$, then $f(n) = \sum_{d|n} \mu(d) g(n/d)$ for all $n \geq 1$.

6.3 Burnside's Lemma (Text)

The number of orbits of a set X under the group action G equals the average number of elements of X fixed by the elements of G.

Here's an example. Consider a square of $2n$ times $2n$ cells. How many ways are there to color it into X colors, up to rotations and/or reflections? Here, the group has only 8 elements (rotations by 0, 90, 180 and 270 degrees, reflections over two diagonals, over a vertical line and over a horizontal line). Every coloring stays itself after rotating by 0 degrees, so that rotation has X^{4n^2} fixed points. Rotation by 180 degrees and reflections over a horizonal/vertical line split all cells in pairs that must be of the same color for a coloring to be unaffected by such rotation/reflection, thus there exist X^{2n^2} such colorings for each of them. Rotations by 90 and 270 degrees split cells in groups of four, thus yielding X^{n^2} fixed colorings. Reflections over diagonals split cells into $2n$ groups of 1 (the diagonal itself) and $2n^2 - n$ groups of 2 (all remaining cells), thus yielding $X^{2n^2-n+2n} = X^{2n^2+n}$ unaffected colorings. So, the answer is $(X^{4n^2} + 3X^{2n^2} + 2X^{n^2} + 2X^{2n^2+n})/8$.

7 Miscellaneous

7.1 Knuth–Morris–Pratt (KMP)

```
/*
Searches for the string w in the string s (of length k). Returns the
0-based index of the first match (k if no match is found). Algorithm
runs in O(k) time.
*/

typedef vector<int> VI;

void buildTable(string& w, VI& t)
{
  t = VI(w.length());
  int i = 2, j = 0;
  t[0] = -1; t[1] = 0;

  while(i < w.length())
  {
    if(w[i-1] == w[j]) { t[i] = j+1; i++; j++; }
    else if(j > 0) j = t[j];
    else { t[i] = 0; i++; }
  }
}

int KMP(string& s, string& w)
{
  int m = 0, i = 0;
  VI t;

  buildTable(w, t);
  while(m+i < s.length())
  {
    if(w[i] == s[m+i])
    {
      i++;
      if(i == w.length()) return m;
    }
    else
    {
      m += i-t[i];
      if(i > 0) i = t[i];
    }
  }
  return s.length();
}
```

7.2 2-SAT

```
// 2-SAT solver based on Kosaraju's algorithm.
// Variables are 0-based. Positive variables are stored in vertices 2n, corresponding negative variables in 2n+1
// TODO: This is quite slow (3x-4x slower than Gabow's algorithm)
struct TwoSat {
    int n;
    vector<vector<int> > adj, radj, scc;
    vector<int> sid, vis, val;
    stack<int> stk;
    int scnt;

    // n: number of variables, including negations
    TwoSat(int n): n(n), adj(n), radj(n), sid(n), vis(n), val(n, -1) {}

    // adds an implication
    void impl(int x, int y) { adj[x].push_back(y); radj[y].push_back(x); }
    // adds a disjunction
    void vee(int x, int y) { impl(x^1, y); impl(y^1, x); }
    // forces variables to be equal
    void eq(int x, int y) { impl(x, y); impl(y, x); impl(x^1, y^1); impl(y^1, x^1); }
    // forces variable to be true
    void tru(int x) { impl(x^1, x); }

    void dfs1(int x) {
        if (vis[x]++) return;
        for (int i = 0; i < adj[x].size(); i++) {
            dfs1(adj[x][i]);
        }
        stk.push(x);
    }

    void dfs2(int x) {
        if (!vis[x]) return; vis[x] = 0;
        sid[x] = scnt; scc.back().push_back(x);
        for (int i = 0; i < radj[x].size(); i++) {
            dfs2(radj[x][i]);
        }
    }

    // returns true if satisfiable, false otherwise
    // on completion, val[x] is the assigned value of variable x
    // note, val[x] = 0 implies val[x^1] = 1
    bool two_sat() {
        scnt = 0;
        for (int i = 0; i < n; i++) {
            dfs1(i);
        }
        while (!stk.empty()) {
            int v = stk.top(); stk.pop();
            if (vis[v]) {
                scc.push_back(vector<int>());
                dfs2(v);
                scnt++;
```

```
            }
        }
        for (int i = 0; i < n; i += 2) {
            if (sid[i] == sid[i+1]) return false;
        }
        vector<int> must(scnt);
        for (int i = 0; i < scnt; i++) {
            for (int j = 0; j < scc[i].size(); j++) {
                val[scc[i][j]] = must[i];
                must[sid[scc[i][j]^1]] = !must[i];
            }
        }
        return true;
    }
};
```

7.3 Shunting Yard (Pseudocode)

```
// Add '(' to start of expression, and ')' to end.
O = empty vector of tokens (values or operators)
S = empty stack of tokens (brackets or operators)
for each token:
  if token == value:
    O.push(token)
  else if token == '(':
    S.push(token)
  else if token == ')':
    while S.top() != '(':
      O.push(S.top())
      S.pop()
    S.pop()
  else:
    // Note: If token is a right-associative operator (^), this should be <=.
    // priority('(') < priority('+') < priority('*').
    while priority(S.top()) < priority(token):
      O.push(S.top())
      S.pop()
    S.push(token)
// Finally, evaluate O as a postfix expression.
```

7.4 Convex hull trick

```
// "Convex hull trick": data structure that maintains a set of lines y = mx + b and allows querying the minimum value of mx_0 + b over all lines for some given x_0. Very useful in optimizing DP algorithms for partitioning problems.
// Tested against USACO MAR08 acquire. TODO: Test against IOI '02 Batch.
struct ConvexHullTrick {
    typedef long long LL;
    vector<LL> M;
    vector<LL> B;
    vector<double> left;
    ConvexHullTrick() {}
    bool bad(LL m1, LL b1, LL m2, LL b2, LL m3, LL b3) {
            // Careful, this may overflow
            return (b3-b1)*(m1-m2) < (b2-b1)*(m1-m3);
    }
    // Add a new line to the structure, y = mx + b.
    // Lines must be added in decreasing order of slope.
    void add(LL m, LL b) {
            while (M.size() >= 2 && bad(M[M.size()-2], B[B.size()-2], M.back(), B.back(), m, b)) {
                M.pop_back(); B.pop_back(); left.pop_back();
            }
            if (M.size() && M.back() == m) {
                    if (B.back() > b) {
                            M.pop_back(); B.pop_back(); left.pop_back();
                    } else {
                            return;
                    }
            }
            if (M.size() == 0) {
                    left.push_back(-numeric_limits<double>::infinity());
            } else {
                    left.push_back((double)(b - B.back())/(M.back() - m));
            }
            M.push_back(m);
            B.push_back(b);
    }
    // Get the minimum value of mx + b among all lines in the structure.
    // There must be at least one line.
    LL query(LL x) {
            int i = upper_bound(left.begin(), left.end(), x) - left.begin();
            return M[i-1]*x + B[i-1];
        }
};
```

7.5 Binary search

```
// Binary search. This is included because binary search can be tricky.
// n is size of array A, c is value we're searching for. Semantics follow those of std::lower_bound and std::upper_bound
int lower_bound(int A[], int n, int c) {
    int l = 0;
    int r = n;
    while (l < r) {
        int m = (r-l)/2+l; //prevents integer overflow
        if (A[m] < c) l = m+1; else r = m;
    }
    return l;


int upper_bound(int A[], int n, int c) {
    int l = 0;
    int r = n;
    while (l < r) {
        int m = (r-l)/2+l;
        if (A[m] <= c) l = m+1; else r = m;
    }
    return l;
}
```

7.6 All nearest smaller values

```
// Linear time all nearest smaller values, standard stack-based algorithm.
// ansv_left stores indices of nearest smaller values to the left in res. -1 means no smaller value was found.
// ansv_right likewise looks to the right. v.size() means no smaller value was found.
void ansv_left(vector<int>& v, vector<int>& res) {
    stack<pair<int, int> > stk; stk.push(make_pair(INT_MIN, v.size()));
    for (int i = v.size()-1; i >= 0; i--) {
        while (stk.top().first > v[i]) {
            res[stk.top().second] = i; stk.pop();
        }
        stk.push(make_pair(v[i], i));
    }
    while (stk.top().second < v.size()) {
        res[stk.top().second] = -1; stk.pop();
    }
}

void ansv_right(vector<int>& v, vector<int>& res) {
    stack<pair<int, int> > stk; stk.push(make_pair(INT_MIN, -1));
    for (int i = 0; i < v.size(); i++) {
        while (stk.top().first > v[i]) {
            res[stk.top().second] = i; stk.pop();
        }
        stk.push(make_pair(v[i], i));
    }
    while (stk.top().second > -1) {
        res[stk.top().second] = v.size(); stk.pop();
    }
}
```

7.7 Longest palindromic substring

```
// Manacher's algorithm: finds maximal palindrome lengths centered around each
// position in a string (including positions between characters) and returns
// them in left-to-right order of centres. Linear time.
// Ex: "opposes" -> [0, 1, 0, 1, 4, 1, 0, 1, 0, 1, 0, 3, 0, 1, 0]
vector<int> fastLongestPalindromes(string str) {
    int i=0,j,d,s,e,lLen,palLen=0;
    vector<int> res;
    while (i < str.length()) {
        if (i > palLen && str[i-palLen-1] == str[i]) {
            palLen += 2; i++; continue;
        }
        res.push_back(palLen);
        s = res.size()-2;
        e = s-palLen;
        bool b = true;
        for (j=s; j>e; j--) {
            d = j-e-1;
            if (res[j] == d) { palLen = d; b = false; break;
                }
            res.push_back(min(d, res[j]));
        }
        if (b) { palLen = 1; i++; }
    }
    res.push_back(palLen);
    lLen = res.size();
    s = lLen-2;
    e = s-(2*str.length()+1-lLen);
    for (i=s; i>e; i--) { d = i-e-1; res.push_back(min(d, res[i])); }
    return res;
```

```
}
```

7.8 .vimrc

```
set number
set wrap
set linebreak
set nolist
set mouse=a
set hlsearch
set tabstop=4
set shiftwidth=4
set softtabstop=4
set expandtab
set autoindent
```