Sudoers Codebook

Suyash Agrawal, Saket Dingliwal, Utkarsh N. Singh

Contents

1 Flow and matching

1.1 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;
  }
};
```

1.2 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);
    }
};
```

1.3 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;
}
```

1.4 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;
}
```

1.5 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);
}
```

1.6 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

1.7 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;
}
```

1.8 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.

1.9 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 Geometry

2.1 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);
}
```

```cpp
// 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;
}
```

```cpp
// 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;
}
```

2.2 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;
}
```

2.3 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]);
    }
```

```
}

long long int area_hull(const vector<pair<int, int>> &↩
    hull){
    //returns 2*Area of the convex hull
    int sizehull = hull.size(), j=hull.size()-1;
    long long int area=0;
    for(int i=0; i<sizehull; i++)
    {
        area+=((hull[j].first*hull[i].second)-(hull[i].↩
            first*hull[j].second));
        j=i;
    }
    area=abs(area);
    return area;
}
```

2.4 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)

3 Math Algorithms

3.1 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;
  }
}
```

3.2 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;
      }
   }
}
```

3.3 Euler's Totient

```
// Euler s Totient function phi(n) is count of numbers in
// {1, 2, 3,    , n} whose GCD with n is 1.
// 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;
    }
```

```
}

int fi(int n) {
  int result = n;
  for(int i=2;i*i <= n;i++) {
    if (n % i == 0) result -= result / i;
    while (n % i == 0) n /= i;
  }
  if (n > 1) result -= result / n;
  return result;
}
```

4 Graphs

4.1 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;
    }
};
```

4.2 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);
    }
}
```

5 Data Structures

5.1 LCA

```
int dp[MAXN][MAXLOGN];
int depth[MAXN]; //depth of nodes (root is 0)
int parent[MAXN]; //immediate parent of node (for root ↩
   is -1)
void preprocess_lca(int N)
{
    //Initialize DP. DP[i][j] stores the parent of node ↩
       i at height 2^i from node.
    for(int i = 0 ; i < N ; i++)
        for(int j = 0 ; (1<<j) < N; j++)
            dp[i][j] = -1;
    //At height 2^0 = 1, DP[node][0] = parent[node]
    for(int i = 0; i < N ; i ++)
        dp[i][0] = parent[i];
    //Now start computing
    for(int j = 1; (1<<j) < N; j++)
        for(int i = 0 ; i < N ; i++)
            if(dp[i][j-1] != -1)
                dp[i][j] = dp[dp[i][j-1]][j-1];
}

int lca(int u, int v)
{
    u--, v--;
    if(depth[u] < depth[v])
        swap(u, v);
    int log = 0;
    //First, bring u to same level as that of v
    while((1 << log) <= depth[u]) //for "<", log will ↩
       exceed unless 1<<log == depth[u], and needs to ↩
       be decreased eventually
        log++;
    log--;
    for(int i = log; i>=0; i--)
        if(depth[u] - (1<<i) >= depth[v])
            u = dp[u][i];
    //now ensured same level
    if(u == v)
        return u;
    //now start going up
    for(int i = log; i>=0; i--)
```

```
        if (dp[u][i] != -1 && dp[u][i] != dp[v][i])
            u = dp[u][i], v = dp[v][i];
    return parent[u];
}
```

5.2 Segment Tree (Lazy)

```
int arr[100001], segtree[4*100001], lazy[4*100001];

void build(int node, int start, int end)
{
    if(start == end)
    {
        segtree[node] = arr[start];
        lazy[node] = 0;
    }
    else
    {
        int mid = (start+end)/2;
        build(2*node, start, mid);
        build(2*node+1, mid+1, end);
        segtree[node] = min(segtree[2*node], segtree[2*↩
           node+1]);
        lazy[node] = 0;
    }
}

void update(int node, int start, int end, int X)
{
    if(start > end)
        return;
    if(lazy[node])
    {
        segtree[node] -= lazy[node];
        if(start != end)
        {
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(segtree[node] > X)
    {
        segtree[node] -= 1;
        if(start != end)
        {
            lazy[2*node] += 1;
            lazy[2*node+1] += 1;
        }
    }
    else
    {
        if(start == end)
            return;
        int mid = (start+end)/2;
        update(2*node, start, mid, X);
        update(2*node+1, mid+1, end, X);
        segtree[node] = min(segtree[2*node], segtree[2*↩
           node+1]);
    }
}


int query(int node, int start, int end, int idx)
{
    if(idx < start || idx > end || start > end)
        return 0;
    if(lazy[node])
    {
        segtree[node] -= lazy[node];
        if(start != end)
        {
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(start == end)
        return segtree[node];
    else
    {
        int mid = (start+end)/2;
        if(start<=idx && idx<=mid)
            return query(2*node, start, mid, idx);
        else
            return query(2*node+1, mid+1, end, idx);
    }
}
```

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 Expansions

$$\text{coeff of } x^i \text{ in } (1-x)^{-n} = \binom{n+r-1}{r}$$
$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$
$$\binom{m}{n} = \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod p$$

where,

$$m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0$$
$$n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$$

6.3 Facts

- **Wilsons Theorem**: A natural number $p > 1$ is a prime number if and only if $(p-1)! = -1 \pmod p$
- Sum of values of totient functions of all divisors of n is equal to n.

6.4 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 $n \geq 1$.

6.5 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 String Algorithms

7.1 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) {
    if (L==1) return;
    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;
}
```

7.2 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();
}
```

7.3 Z Algorithm

```
// string s as input outputs in z vector the length of
// longest common prefix of substring starting at i
// and s in o(n) time
void Z(string &s,vector<int> &z){
  int n = s.length();
  z.resize(n);
  int L = 0, R = 0;
  for (int i = 1; i < n; i++) {
    if (i > R) {
      L = R = i;
      while (R < n && s[R-L] == s[R]) R++;
      z[i] = R-L; R--;
    } else {
      int k = i-L;
      if (z[k] < R-i+1) z[i] = z[k];
      else {
        L = i;
        while (R < n && s[R-L] == s[R]) R++;
        z[i] = R-L; R--;
      }
    }
  }
}
```

7.4 Longest palindromic substring (Manacherś)

```
// 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;
}
```

8 Miscellaneous

8.1 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;
    }
};
```

8.2 Grundy Number

Grundy Number is equal to 0 for a game that is lost immediately by the first player, and is equal to Mex of the nimbers of all possible next positions for any other game.

8.3 Sprague-Grundy Theorem

Suppose there is a composite game (more than one sub-game) made up of N sub-games and two players, A and B. Then if both A and B play optimally, then the player starting first is guaranteed to win if the XOR of the grundy numbers of position in each sub-games at the beginning of the game is non-zero. Otherwise, if the XOR evaluates to zero, then player A will lose definitely, no matter what.

8.4 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];
      }
};
```

8.5 Counting Inversion

```
// l = left index of the array, r = right index of the ↩
   array,
// arr = main array for which to compute inversions
// tmp_vec = temporary vector to store intermediate ↩
   result (should be of size N)
// NOTE: arr is modified in the process ! (sorted)
int count_inv(int l, int r,vector<int>& arr,vector<int>&↩
    tmp_vec){
    if (l==r) return 0;
    int mid = l + (r-l)/2;
    int inv_l = count_inv(l,mid,arr,tmp_vec);
    int inv_r = count_inv(mid+1,r,arr,tmp_vec);
    int p1 = l, p2 = mid+1;
    int extra_inv = 0;
    int idx = l;
```

```
    while(p1<=mid && p2 <= r)
    {
        if(arr[p1] <= arr[p2])
            tmp_vec[idx++]=arr[p1++];
        else
        {
            extra_inv += (mid-p1+1);
            tmp_vec[idx++]=arr[p2++];
        }
    }
    while(p1<=mid)
        tmp_vec[idx++]=arr[p1++];
    while(p2<=r)
        tmp_vec[idx++]=arr[p2++];
    for(int i=l;i<=r;i++)
        arr[i]=tmp_vec[i];
    return (extra_inv+inv_r+inv_l);
}
```

8.6 Template

```
#include <bits/stdc++.h>
using namespace std;

typedef pair<int,int> PII;
typedef long long int LL;
typedef pair<int,int> PII;
typedef pair<LL,LL> PLL;
#define MK make_pair
#define PB push_back
#define SZ(a) ((int)(a.size()))
#define MOD(a,m) ( ( ( (a) % (m) ) + (m) ) % (m) )

#define what_is(x) cerr << #x << " is " << x << endl;

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    // freopen("in","r",stdin);
    return 0;
}
```

8.7 Primes

```
2350490027,2125898167,1628175011,1749241873,1593209441
1524872353,1040332871,2911165193,1387346491,2776808933
```