Listings

Listing 1: **GEOMETRY**

Listing 2: 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  ); }
bool operator <(const PT &p)   const { return (x < p.x ? true : (x == p.x && y < p.y)); }
bool operator ==(const PT &p)  const { return fabs(x-p.x) < EPS && fabs(y-p.y) < EPS; }
bool operator !=(const PT &p)  const { return !((*this) == p); }
};
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) {
  return os << "(" << p.x << "," << p.y << ")";
}

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

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

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

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

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

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

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

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

//Returns of list of intersection points between segments s1, and s2
//If they do not intersect, the result is an empty vector
//If they intersect at exactly 1 point, the result contains that point
//If they overlap for non-0 distance, the left and right points of that intersection
//  are returned
vector<PT> intersect(PT a, PT b, PT c, PT d) {
    vector<PT> res;
    if(fabs(cross(b-a, c-b)) < EPS && fabs(cross(b-a, d-b)) < EPS &&
        fabs(cross(d-c, a-d)) < EPS && fabs(cross(d-c, b-d)) < EPS) {
        PT min_s1 = min(a, b), max_s1 = max(a, b);
        PT min_s2 = min(c, d), max_s2 = max(c, d);
        if(min_s1 < min_s2) {
            if(max_s1 < min_s2) return res;//return false;
        }
        else if(min_s2 < min_s1 && max_s2 < min_s1) return res;//return false;
        PT start = max(min_s1, min_s2), end = min(max_s1, max_s2);
        if(start==end) res.push_back(start);//overlap is one point
        else {
            res.push_back(min(start, end));
            res.push_back(max(start, end));
        }
        return res;//return true; //remove overlap code block
    }
    double x1 = (b.x-a.x), y1 = (b.y-a.y), x2 = (d.x - c.x), y2 = (d.y - c.y);
    double u1 = (-y1*(a.x - c.x) + x1 * (a.y - c.y))/(-x2 * y1 + x1 * y2);
    double u2 = (x2 * (a.y - c.y) - y2 * (a.x - c.x)) / (-x2 * y1 + x1 * y2);
    if(u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1)
        res.push_back(PT((a.x + u2*x1),(a.y+u2*y1))); //return true;
    return res;//return false;
}
```

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

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

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

Listing 3: Convex Hull

```
ll dist(pair<int, int> a, pair<int, int> b) {
    int dx1 = b.first - a.first;
    int dy1 = b.second - a.second;
    return 1LL * dx1 * dx1 + 1LL * dy1 * dy1;
}

ll ccw(pair<int, int> a, pair<int, int> b, pair<int, int> c) {
    int dx1 = b.first - a.first;
    int dy1 = b.second - a.second;
    int dx2 = c.first - a.first;
    int dy2 = c.second - a.second;
    return 1LL * dx1 * dy2 - 1LL * dy1 * dx2;
}

vector<pair<int, int>> getPoly(vector<pair<int, int> > &poly) {
    int n = poly.size();
```

```
    pair<int, int> least = poly[0];
    int pos = 0;
    for(int i = 0; i < n; i++) {
        if(poly[i] < least) {
            least = poly[i];
            pos = i;
        }
    }
    swap(poly[0], poly[pos]);
    sort(poly.begin() + 1, poly.end(), [&](const pair<int, int> &a, const pair<int, int>
     ↪ &b){
        auto x = ccw(poly[0], a, b);
        if(x != 0) return x > 0;
        return dist(poly[0], a) < dist(poly[0], b);
    });
    vector<pair<int, int> > hull;
    for(auto &i : poly) {
        while(hull.size() >= 2 && ccw(hull[hull.size()-2], hull.back(), i) <= 0){
            hull.pop_back();
        }
        hull.push_back(i);
    }
    return hull;
}
```

Listing 4: Pick's Theorem

```
struct Point{
    long long x, y;
    Point(){}
    Point(long long x, long long y) : x(x), y(y){}
};

/// twice the area of polygon
long long double_area(Point poly[], int n){
    long long res = 0;
    for (int i = 0, j = n - 1; i < n; j = i++){
        res += ((poly[j].x + poly[i].x) * (poly[j].y - poly[i].y));
    }
    return abs(res);
}

/// number of lattice points strictly on polygon border
long long on_border(Point poly[], int n){
    long long res = 0;
    for (int i = 0, j = n - 1; i < n; j = i++){
        res += __gcd(abs(poly[i].x - poly[j].x), abs(poly[i].y - poly[j].y));
    }
    return res;
}

/// number of lattice points strictly inside polygon
long long interior(Point poly[], int n){
    long long res = 2 + double_area(poly, n) - on_border(poly, n);
    return res / 2;
}
```

Listing 5: Closest Pair of Points

```
//returns pair of indexes into sorted array of points
pair<int, int> closestPair(vector<pair<ll, ll>> &pts) {
    int n = pts.size();
    sort(pts.begin(), pts.end());
    set<pair<pair<ll, ll>, int>> s;
    ll best_dist = 1e18;
    int j = 0;
    pair<int, int> pos;
    for (int i = 0; i < n; ++i) {
        ll d = ceil(sqrt(best_dist));
        while (pts[i].first - pts[j].first > best_dist) {
            s.erase({{pts[j].second, pts[j].first}, j});
            j += 1;
        }
        auto it1 = s.lower_bound({{pts[i].second - d, pts[i].first}, -1});
        auto it2 = s.upper_bound({{pts[i].second + d, pts[i].first}, i});
        for (auto it = it1; it != it2; ++it) {
            ll dx = pts[i].first - it->first.second;
            ll dy = pts[i].second - it->first.first;
            ll dist = dx * dx + dy * dy;
            if(best_dist > dist) {
                best_dist = dist;
                pos.first = it->second;
                pos.second = i;
            }
        }
        s.insert({{pts[i].second, pts[i].first}, i});
    }
    return pos;
}
```

Listing 6: **GRAPHS**

Listing 7: Bipartite Matching

```
// Implementation of Hopcroft-Karp algorithm
struct BipartiteMatcher {
    vector<vector<int>> G;
    vector<int> L, R, Viz;
    vector<bool> visitedA, visitedB;
    BipartiteMatcher(int n, int m) :
        G(n), L(n, -1), R(m, -1), Viz(n), visitedA(n), visitedB(m) {}

    void AddEdge(int a, int b) {
        G[a].push_back(b);
    }

    bool Match(int node) {
        if(Viz[node])
            return false;
        Viz[node] = true;
        for(auto vec : G[node]) {
            if(R[vec] == -1 || Match(R[vec])) {
                L[node] = vec;
                R[vec] = node;
                return true;
            }
        }
```

```
        return false;
    }
    int Solve() {
        bool ok = true;
        while(ok) {
            ok = false;
            fill(Viz.begin(), Viz.end(), 0);
            for(int i = 0; i < L.size(); ++i)
                if(L[i] == -1)
                    ok |= Match(i);
        }

        int ret = 0;
        for(int i = 0; i < L.size(); ++i)
            ret += (L[i] != -1);
        return ret;
    }
    void dfs(int node) {
        visitedA[node] = true;
        for(auto to : G[node]) {
            if(!visitedB[to] && R[to] != -1) {
                visitedB[to] = true;
                dfs(R[to]);
            }
        }
    }
    //returns all the nodes in the min vertex cover
    //O(time to find max matching) + O(n)
    //first vector: all nodes in the cover on the left side
    //second vector: all nodes in the cover on the right side
    pair<vector<int>, vector<int>> getMinVertexCover() {
        Solve();
        for(int i = 0; i < L.size(); ++i) {
            if(L[i] == -1) {
                dfs(i);
            }
        }
        pair<vector<int>, vector<int>> cover;
        for(int i = 0; i < L.size(); ++i) {
            if(!visitedA[i]) cover.first.push_back(i);
        }
        for(int i = 0; i < R.size(); ++i) {
            if(visitedB[i]) cover.second.push_back(i);
        }
        return cover;
    }
};
```

Listing 8: Bridges

```
//Doesn't handle multiple edges
const int Max = 3e5+2;
vector<int> adj[Max], bridgeTree[Max];
bool visited[Max];
int timeIn[Max], currTime = 0, minTime[Max];
vector<pair<int, int> > bridges;

vector<int> p(Max,-1);
int find(int x) {return p[x] < 0 ? x : p[x] = find(p[x]);}
void merge(int x, int y) {
    if((x=find(x)) == (y=find(y))) return;
    if(p[y] < p[x]) swap(x,y);
    p[x] += p[y];
    p[y] = x;
}

void dfs(int node, int prev) {
    visited[node] = true;
    timeIn[node] = minTime[node] = ++currTime;
    for(int to : adj[node]) {
        if(to != prev) {
            minTime[node] = min(minTime[node], timeIn[to]);
        }
        if(visited[to]) continue;
        dfs(to, node);
        minTime[node] = min(minTime[node], minTime[to]);
        if(minTime[to] > timeIn[node]) {
            bridges.push_back({node, to});
        } else {
            merge(node, to);
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int n, m;
    cin >> n >> m;
    for(int i = 1; i <= n; ++i) {
        minTime[i] = timeIn[i] = Max;
        visited[i] = false;
    }
    int u, v;
    for(int i = 0; i < m; ++i) {
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    currTime = 0;
    dfs(1,1);
    for(auto &p : bridges) {
        u = find(p.first);
        v = find(p.second);
        bridgeTree[u].push_back(v);
        bridgeTree[v].push_back(u);
    }
    return 0;
}
```

Listing 9: Centroid Decomposition

```
const int Max = 2e5+2;
vector<int> adj[Max];
int sizes[Max], parent[Max];
bool removed[Max];

void dfs2(int node, int par) {
    sizes[node] = 1;
```

```
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            dfs2(to, node);
            sizes[node] += sizes[to];
        }
    }
}

int findCentroid(int node) {
    dfs2(node, node);
    bool found = true;
    int sizeCap = sizes[node]/2;
    int par = node;
    while(found) {
        found = false;
        for(int to : adj[node]) {
            if(to != par && !removed[to] && sizes[to] > sizeCap) {
                found = true;
                par = node;
                node = to;
                break;
            }
        }
    }
    return node;
}

void dfs1(int node, int par) {
    removed[node] = true;
    parent[node] = par;
    for(int to : adj[node]) {
        if(!removed[to]) {
            dfs1(findCentroid(to), node);
        }
    }
}
//dfs1(findCentroid(1), 0);
```

Listing 10: Count Paths of Each Length

```
const int Max = 1e6+10;
int n, sizes[Max];
vector<int> adj[Max], cntPathLength[Max];
ll prod[Max], cntTotalPathLengths[Max] = {0}, cntTotalPathLengthsNaive[Max] = {0};
bool removed[Max];

void dfs2(int node, int par, int root, int currDist) {
    if(cntPathLength[root].size() <= currDist) {
        cntPathLength[root].push_back(0);
    }
    cntPathLength[root][currDist]++;
    sizes[node] = 1;
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            dfs2(to, node, root, currDist+1);
            sizes[node] += sizes[to];
        }
    }
}
int findCentroid(int node) {
    dfs2(node, node, node, 1);
    bool found = true;
    int sizeCap = sizes[node]/2;
    int par = node;
    while(found) {
        found = false;
        for(int to : adj[node]) {
            if(to != par && !removed[to] && sizes[to] > sizeCap) {
                found = true;
                par = node;
                node = to;
                break;
            }
        }
    }
    return node;
}

void dfs1(int node, int par) {
    removed[node] = true;
    int maxLength = 1;
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            cntPathLength[to].clear();
            cntPathLength[to].push_back(0);
            dfs2(to, to, to, 1);
            maxLength = max(maxLength, (int)cntPathLength[to].size());
        }
    }
    vector<int> temp(maxLength, 0);
    temp[0]++;
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            vector<int> prod;
            multiply(temp, cntPathLength[to], prod);
            for(int i = 0; i < prod.size(); ++i) {
                cntTotalPathLengths[i] += prod[i];
            }
            for(int i = 0; i < cntPathLength[to].size(); ++i) {
                temp[i] += cntPathLength[to][i];
            }
        }
    }

    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            dfs1(findCentroid(to), node);
        }
    }
}
```

Listing 11: Cut Vertices

```
const int Max = 1e5+2;
vector<int> adj[Max];
bool visited[Max], cutNode[Max];
int timeIn[Max], currTime = 0, minTime[Max];

void dfs(int node, int prev) {
```

```
    visited[node] = true;
    timeIn[node] = minTime[node] = ++currTime;
    int numChildren = 0;
    for(int to : adj[node]) {
        if(to != prev) {
            minTime[node] = min(minTime[node], timeIn[to]);
        }
        if(visited[to]) continue;
        numChildren++;
        if(node == prev && numChildren > 1) {
            cutNode[node] = true;
        }
        dfs(to, node);
        minTime[node] = min(minTime[node], minTime[to]);
        if(node != prev && minTime[to] >= timeIn[node]) {
            cutNode[node] = true;
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int n, m;
    cin >> n >> m;
    for(int i = 0; i < Max; ++i) {
        minTime[i] = timeIn[i] = Max;
        visited[i] = cutNode[i] = false;
    }
    int u, v;
    for(int i = 0; i < m; ++i) {
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    currTime = 0;
    for(int i = 0; i < n; ++i) {
        if(!visited[i]) {
            dfs(i,i);
        }
    }
    for(int i = 0; i < n; ++i) {
        if(cutNode[i]) {

        }
    }
    return 0;
}
```

Listing 12: Floyd Warshall

```
for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            if (w[i][j] > w[i][k] + w[k][j]){
                w[i][j] = w[i][k] + w[k][j];
                prev[i][j] = k;
            }
        }
```

```
    }
}
```

Listing 13: DSU on Tree

```
const int Max = 1e5+3;
int color[Max], Time = 1, timeIn[Max], timeOut[Max], ver[Max], size[Max], cnt[Max],
    ↪ heavyChild[Max], Depth[Max] = {0}, answer[Max];
vector<int> adj[Max];

void dfs(int node, int prev) {
    timeIn[node] = Time;
    ver[Time] = node;
    Time++;
    size[node] = 1;
    int largest = heavyChild[node] = -1;
    Depth[node] = 1 + Depth[prev];
    for(int to : adj[node]) {
        if(to == prev) continue;
        dfs(to, node);
        size[node] += size[to];
        if(size[to] > largest) {
            largest = size[to];
            heavyChild[node] = to;
        }
    }
    timeOut[node] = Time;
}

void dfs1(int node, int prev, bool keep = true) {
    for(int to : adj[node]) {
        if(to == prev || to == heavyChild[node]) continue;
        dfs1(to, node, false);
    }
    if(heavyChild[node] != -1) {
        dfs1(heavyChild[node], node, true);
    }
    cnt[color[node]]++;
    for(int to : adj[node]) {
        if(to == prev || to == heavyChild[node]) continue;
        for(int i = timeIn[to]; i < timeOut[to]; ++i) {
            cnt[color[ver[i]]]++;
        }
    }
    if(!keep) {
        for(int i = timeIn[node]; i < timeOut[node]; ++i) {
            cnt[color[ver[i]]]--;
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int n;
    cin >> n;
    dfs(1, 1);
    dfs1(1, 1);
    for(int i = 1; i <= n; ++i) {
```

```
        cout << answer[i] << ' ';
    }
    cout << '\n';
    return 0;
}
```

Listing 14: Lowest Common Ancestor (LCA)

```
const int Max = 1e5+3, Log = 20+1;
vector<int> adj[Max];
int memo[Max][Log];
int depth[Max];

void dfs(int node, int prev, int Depth = 0) {
    depth[node] = Depth;
    memo[node][0] = prev;
    for(int i = 1; i < Log; ++i) {
        memo[node][i] = memo[memo[node][i-1]][i-1];
    }
    for(int to : adj[node]) {
        if(to == prev) continue;
        dfs(to, node, Depth+1);
    }
}

int LCA(int x, int y) {
    if(depth[x] < depth[y]) swap(x,y);
    int diff = depth[x] - depth[y];
    for(int k = Log-1; k >= 0; --k) {
        if(diff&(1<<k)) {
            x = memo[x][k];
        }
    }
    for(int k = Log-1; k >= 0; --k) {
        if(memo[x][k] != memo[y][k]) {
            x = memo[x][k];
            y = memo[y][k];
        }
    }
    if(x != y) x = memo[x][0];
    return x;
}
//dfs(1,1);
```

Listing 15: Strongly Connected Components (SCC)

```
int n,m;
vector<vector<int> > adj, adjInv;
vector<int> scc;
int sccID;
vector<bool> visited;

void dfs1(int curr, stack<int> &seen) {
    visited[curr] = true;
    for(int x : adj[curr]) {
        if(!visited[x]) {
            dfs1(x, seen);
        }
    }
    seen.push(curr);
}

void dfs2(int curr) {
    visited[curr] = true;
    scc[curr] = sccID;
    for(int x : adjInv[curr]) {
        if(!visited[x]) {
            dfs2(x);
        }
    }
}

void calcSCC() {
    visited.resize(n+1,false);
    stack<int> seen;
    for(int i = 1; i <= n; ++i) {
        if(!visited[i]) {
            dfs1(i, seen);
        }
    }
    visited.clear();
    visited.resize(n+1,false);
    sccID = 0;
    while(!seen.empty()) {
        while(!seen.empty() && visited[seen.top()]) seen.pop();
        if(!seen.empty()) {
            dfs2(seen.top());
            sccID++;
        }
    }
}
```

Listing 16: **MAX FLOW**

Listing 17: Dinic's Max Flow

```
struct maxflow {
    struct edge {
        ll a, b, cap, flow;
    };

    ll n, s, t;
    vector<ll> d, ptr, q;
    vector<edge> e;
    vector<vector<ll>> g;

    maxflow(int _n, int _s, int _t) : n(_n), s(_s), t(_t) {
        d.resize(n);
        ptr.resize(n);
        q.resize(n);
        g.resize(n);
    }

    void addedge(ll a, ll b, ll cap) {
        edge e1 = { a, b, cap, 0 };
        edge e2 = { b, a, 0, 0 };
        g[a].push_back((ll) e.size());
```

```
        e.push_back(e1);
        g[b].push_back((ll) e.size());
        e.push_back(e2);
    }

    bool bfs() {
        ll qh=0, qt=0;
        q[qt++] = s;
        d.assign(d.size(), -1);
        d[s] = 0;
        while(qh < qt && d[t] == -1) {
            ll v = q[qh++];
            for(size_t i=0; i<g[v].size(); ++i) {
                ll id = g[v][i],
                    to = e[id].b;
                if(d[to] == -1 && e[id].flow < e[id].cap) {
                    q[qt++] = to;
                    d[to] = d[v] + 1;
                }
            }
        }
        return d[t] != -1;
    }

    ll dfs(ll v, ll flow) {
        if(!flow) return 0;
        if(v == t) return flow;
        for(; ptr[v]<(ll)g[v].size(); ++ptr[v]) {
            ll id = g[v][ptr[v]];
            ll to = e[id].b;
            if(d[to] != d[v] + 1)  continue;
            ll pushed = dfs(to, min (flow, e[id].cap - e[id].flow));
            if(pushed) {
                e[id].flow += pushed;
                e[id^1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }

    ll getflow() {
        ll flow = 0;
        for(;;) {
            if(!bfs())  break;
            ptr.assign(ptr.size(), 0);
            while(ll pushed = dfs(s,inf)) {
                flow += pushed;
            }
        }
        return flow;
    }
};
```

Listing 18: Hungarian Algorithm

```
// this is one-indexed
// jobs X workers cost matrix
// cost[i][j] is cost of job i done by worker j
// #jobs must be <= #workers
// Default finds min cost; to find max cost set all costs[i][j] to -costs[i][j]
ll HungarianMatch(const vector<vector<ll>>& a) {
    ll n = a.size()-1;
    ll m = a[0].size()-1;
    vector<ll> u(n+1), v(m+1), p(m+1), way(m+1);
    for(ll i = 1; i <= n; ++i) {
        p[0] = i;
        ll j0 = 0;
        vector<ll> minv(m+1, inf);
        vector<char> used(m+1, false);
        do {
            used[j0] = true;
            ll i0 = p[j0], delta = inf, j1;
            for(ll j = 1; j <= m; ++j)
                if(!used[j]) {
                    ll cur = a[i0][j] - u[i0] - v[j];
                    if(cur < minv[j])
                        minv[j] = cur, way[j] = j0;
                    if(minv[j] < delta)
                        delta = minv[j], j1 = j;
                }
            for(ll j = 0; j <= m; ++j)
                if(used[j])
                    u[p[j]] += delta, v[j] -= delta;
                else
                    minv[j] -= delta;
            j0 = j1;
        } while(p[j0] != 0);
        do {
            ll j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while(j0);
    }

    /*
    // For each N, it contains the M it selected
    vector<ll> ans(n+1);
    for(ll j = 1; j <= m ; ++j)
        ans[p[j]] = j;
    */

    return -v[0];
}
```

Listing 19: Min Cost Max Flow

```
struct mincostmaxflow {
    struct edge {
        ll a, b, cap, cost, flow;
        size_t back;
    };

    vector<edge> e;
    vector<vector<ll>> g;
    ll n, s, t;
    ll k = inf; // The maximum amount of flow allowed

    mincostmaxflow(int _n, int _s, int _t) : n(_n), s(_s), t(_t) {
        g.resize(n);
```

```
    }

    void addedge(ll a, ll b, ll cap, ll cost) {
        edge e1 = {a,b,cap,cost,0,g[b].size()};
        edge e2 = {b,a,0,-cost,0,g[a].size()};
        g[a].push_back((ll) e.size());
        e.push_back(e1);
        g[b].push_back((ll) e.size());
        e.push_back(e2);
    }

    // Returns {flow,cost}
    pair<ll,ll> getflow() {
        ll flow = 0, cost = 0;
        while(flow < k) {
            vector<ll> id(n, 0);
            vector<ll> d(n, inf);
            vector<ll> q(n);
            vector<ll> p(n);
            vector<size_t> p_edge(n);
            ll qh=0, qt=0;
            q[qt++] = s;
            d[s] = 0;
            while(qh != qt) {
                ll v = q[qh++];
                id[v] = 2;
                if(qh == n) qh = 0;
                for(size_t i=0; i<g[v].size(); ++i) {
                    edge& r = e[g[v][i]];
                    if(r.flow < r.cap && d[v] + r.cost < d[r.b]) {
                        d[r.b] = d[v] + r.cost;
                        if(id[r.b] == 0) {
                            q[qt++] = r.b;
                            if(qt == n) qt = 0;
                        }
                        else if(id[r.b] == 2) {
                            if(--qh == -1) qh = n-1;
                            q[qh] = r.b;
                        }
                        id[r.b] = 1;
                        p[r.b] = v;
                        p_edge[r.b] = i;
                    }
                }
            }
            if(d[t] == inf) break;
            ll addflow = k - flow;
            for(ll v=t; v!=s; v=p[v]) {
                ll pv = p[v]; size_t pr = p_edge[v];
                addflow = min(addflow, e[g[pv][pr]].cap - e[g[pv][pr]].flow);
            }
            for(ll v=t; v!=s; v=p[v]) {
                ll pv = p[v]; size_t pr = p_edge[v], r = e[g[pv][pr]].back;
                e[g[pv][pr]].flow += addflow;
                e[g[v][r]].flow -= addflow;
                cost += e[g[pv][pr]].cost * addflow;
            }
            flow += addflow;
        }
        return {flow,cost};
    }
};
```

Listing 20: STRINGS

Listing 21: KMP String Matching

```
struct KMP_Match {
    vector<int> T;
    string pat;
    KMP_Match() {};
    KMP_Match(string pattern) : pat(pattern) {this->buildTable(pat);};
    void buildTable(string pattern) {
        pat = pattern;
        T.clear();
        T.resize(pat.length()+1);
        int i = 0, j = -1;
        T[i] = j;
        while(i < pat.size()) {
            while(j >= 0 && pat[i] != pat[j]) j = T[j];
            i++, j++;
            T[i] = j;
        }
    }
    vector<int> find(string txt, bool all = true) {
        int m = 0, i = 0;
        vector<int> matches;
        while(m + i < txt.length()) {
            if(pat[i] == txt[m+i]) {
                if(i == pat.length()-1) {
                    matches.push_back(m);
                    if(!all) return matches;
                    m = m + i - T[i];
                    i = T[i];
                }
                i++;
            } else {
                if(T[i] != -1) {
                    m = m + i - T[i];
                    i = T[i];
                } else {
                    i = 0;
                    m++;
                }
            }
        }
        return matches;
    }
};

int fail[1000005];

// Checks if two arrays are rotationally equivalent
bool KMPints(vector<ll> a, vector<ll> b){
    for(int i = 0; i < a.size(); i++) {
        b.push_back(b[i]);
    }
    int p = 0;
    for(int i = 1; i < a.size(); i++) {
```

```
        while(p && a[i] != a[p]) p = fail[p];
        if(a[i] == a[p]) p++;
        fail[i + 1] = p;
    }
    p = 0;
    for(auto &i : b) {
        while(p && i != a[p]) p = fail[p];
        if(i == a[p]) p++;
        if(p == a.size()) return true;
    }
    return false;
}
```

Listing 22: Manachers (Palindromes)

```
// Returns the longest palindrome
string cntPals(string &s) {
    string T = "";
    for(int i = 0; i < s.size(); ++i) {
        T += "#" + s.substr(i,1);
    }
    T += '#';
    vector<int> P(T.size(),0);
    int center = 0, boundary = 0, maxLen = 0, resCenter = 0, cnt = 0;
    for(int i = 1; i < T.size()-1; ++i) {
        int iMirror = 2 * center - i;
        P[i] = (boundary > i ? min(boundary - i, P[iMirror]) : 0);
        while(i-1-P[i] >= 0 && i+1+P[i] <= (int)T.size()-1 && T[i+1+P[i]] == T[i-1-P[i]])
            P[i]++;
        if(i + P[i] > boundary) {
            center = i;
            boundary = i+P[i];
        }
        if(P[i] > maxLen) {
            maxLen = P[i];
            resCenter = i;
        }
        cnt += (P[i]+1)/2;
    }
    return s.substr((resCenter - maxLen)/2, maxLen);
    //return cnt;//number of palindromes
}
```

Listing 23: Suffix Array

```
const int MAXN = 1 << 21;
string s;
int N, gap;
int sa[MAXN], pos[MAXN], lcp[MAXN], tmp[MAXN];

bool sufCmp(int i, int j) {
    if(pos[i] != pos[j]) return pos[i] < pos[j];
    i += gap;
    j += gap;
    return (i < N && j < N) ? pos[i] < pos[j] : i > j;
}

void buildSA() {
    N = s.length();
    for(int i = 0; i < N; ++i) {
        sa[i] = i;
        pos[i] = s[i];
    }
    for(gap = 1;; gap *= 2) {
        sort(sa, sa + N, sufCmp);
        for(int i = 0; i < N-1; ++i)
            tmp[i+1] = tmp[i] + sufCmp(sa[i], sa[i+1]);
        for(int i = 0; i < N; ++i) pos[sa[i]] = tmp[i];
        if(tmp[N-1] == N-1) break;
    }
}

void buildLCP() {
    N = s.size();
    for(int i = 0, k = 0; i < N; ++i) {
        if(pos[i] != 0 ) {
            for(int j = sa[pos[i]-1]; s[i+k] == s[j+k];) k++;
            lcp[pos[i]] = k;
            if(k) k--;
        }
    }
}
```

Listing 24: Trie

```
const int K = 26;//character size

struct node {
    int next[K];
    bool leaf = 0;
    char pch;
    int p = -1;
    int id;
    node(int p = -1,char ch = '#'):p(p),pch(ch) {
        fill(next,next+K,-1);
    }
};

vector<node> t(1);//adj list

void add_string(string s, int id) {
    int c = 0;
    for(char ch: s){
        int v = ch-'a';
        if(t[c].next[v] == -1) {
            t[c].next[v] = t.size();
            t.emplace_back(c,ch);
        }
        c = t[c].next[v];
    }
    t[c].leaf = 1;
    t[c].id = id;
}

void remove_string(string s) {
    int c = 0;
    for(char ch: s){
        int v = ch-'a';
        if(t[c].next[v] == -1) {
```

```
            return;
        }
        c = t[c].next[v];
    }
    t[c].leaf = 0;
}

int find_string(string s) {
    int c = 0;
    for(char ch: s){
        int v = ch-'a';
        if(t[c].next[v] == -1) {
            return -1;
        }
        c = t[c].next[v];
    }
    if(!t[c].leaf) return -1;
    return t[c].id;
}
```

Listing 25: Z Algorithm

```
string s;
//z[i] is the length of the longest substring
//starting from s[i] which is also a prefix of s
int z[100010];//change size here
void zAlg() {
    int n = s.size();
    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--;
            }
        }
    }
}
```

Listing 26: **DATA STRUCTURES**

Listing 27: Longest Increasing Subsequence (LIS)

```
// Returns an array with the indexes of the LIS
template <class T>
vector<int> LIS(vector<T>& v) {
    if(v.size() == 0) return {};
    vector<int> p(v.size(), -1);
    vector<int> t(v.size(), 0);
    int lis = 1;
    for(int i = 1; i < v.size(); i++) {
        if(v[i] <= v[t[0]]) {
            t[0] = i;
        }
        else if(v[i] > v[t[lis - 1]]) {
            p[i] = t[lis - 1];
            t[lis++] = i;
        }
        else {
            int l = -1;
            int r = lis - 1;
            while(r - l > 1) {
                int m = l + (r - l) / 2;
                if(v[t[m]] >= v[i]) r = m;
                else l = m;
            }
            p[i] = t[r - 1];
            t[r] = i;
        }
    }

    vector<int> ans;
    for(int i = t[lis - 1]; i >= 0; i = p[i]) {
        ans.push_back(i);
    }
    reverse(ans.begin(), ans.end());

    return ans;
}

vector<int> LIS(vector<int> &arr) {//longest non-decreasing sequence
    vector<int> longest(arr.size(),0);
    multiset<int> seq;
    for(int i = 0; i < arr.size(); ++i) {
        seq.insert(arr[i]);
        auto it = seq.upper_bound(arr[i]);
        if(it != seq.end()) seq.erase(it);
        longest[i] = seq.size();
    }
    return longest;
}

vector<int> LSIS(vector<int> &arr) {//longest strictly increasing sequence
    vector<int> longest(arr.size(),0);
    multiset<int> seq;
    for(int i = 0; i < arr.size(); ++i) {
        seq.insert(arr[i]);
        auto it = seq.lower_bound(arr[i]);
        it++;
        if(it != seq.end()) seq.erase(it);
        longest[i] = seq.size();
    }
    return longest;
}
```

Listing 28: Count Rectangles

```
//given a 2D boolean matrix, calculate cnt[i][j]
//cnt[i][j] = the number of times an (i * j) rectangle appears in the matrix
```

```
//such that all cells in the rectangle are false
//O(R*C)
vector<vector<int>> getNumRectangles(vector<vector<bool>> &grid) {
    vector<vector<int>> cnt;
    const int rows = grid.size();
    if(rows == 0) return cnt;
    const int cols = grid[0].size();
    if(cols == 0) return cnt;
    cnt.resize(rows+1, vector<int>(cols+1, 0));
    vector<vector<int>> arr(rows+2, vector<int>(cols+1, 0));
    for(int i = 1; i <= rows; ++i) {
        for(int j = 1; j <= cols; ++j) {
            arr[i][j] = 1 + arr[i][j-1];
            if(grid[i-1][j-1]) arr[i][j] = 0;
        }
    }
    for(int j = 1; j <= cols; ++j) {
        arr[rows+1][j] = 0;
        stack<pair<int, int> > st;
        st.push({0,0});
        for(int i = 1; i <= rows+1; ++i) {
            pair<int, int> curr = {i, arr[i][j]};
            while(arr[i][j] < st.top().second) {
                curr = st.top();
                st.pop();
                cnt[i-curr.first][curr.second]++;
                cnt[i-curr.first][max(arr[i][j], st.top().second)]--;
            }
            st.push({curr.first, arr[i][j]});
        }
    }
    for(int j = 1; j <= cols; ++j) {
        for(int i = rows-1; i >= 1; --i) {
            cnt[i][j] += cnt[i+1][j];
        }
        for(int i = rows-1; i >= 1; --i) {
            cnt[i][j] += cnt[i+1][j];
        }
    }
    for(int i = 1; i <= rows; ++i) {
        for(int j = cols-1; j >= 1; --j) {
            cnt[i][j] += cnt[i][j+1];
        }
    }
    return cnt;
}
```

Listing 29: Disjoint Set

```
vector<int> p(1000001,-1);//change size here if needed
int find(int x) {return p[x] < 0 ? x : p[x] = find(p[x]);}
void merge(int x, int y) {
    if((x=find(x)) == (y=find(y))) return;
    if(p[y] < p[x]) swap(x,y);
    p[x] += p[y];
    p[y] = x;
}
```

Listing 30: Indexed Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class TI>
using indexed_set = tree<TI, null_type, less<TI>, rb_tree_tag,
    ↪ tree_order_statistics_node_update>;
//order_of_key
//find_by_order
```

Listing 31: Largest Range Where Element Is Max/Min

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

int main() {
    int n;
    cin >> n;
    vector<int> arr(n), lefts(n), rights(n);
    for(int &x : arr) cin >> x;
    stack<int> positions;
    for(int i = 0; i < n; ++i) {
        while(!positions.empty() && arr[positions.top()] >= arr[i]) positions.pop();
        if(positions.empty()) {
            lefts[i] = -1;
        } else {
            lefts[i] = positions.top();
        }
        positions.push(i);
    }
    while(!positions.empty()) positions.pop();
    for(int i = n-1; i >= 0; --i) {
        while(!positions.empty() && arr[positions.top()] >= arr[i]) positions.pop();
        if(positions.empty()) {
            rights[i] = n;
        } else {
            rights[i] = positions.top();
        }
        positions.push(i);
    }
    return 0;
}
```

Listing 32: Online Convex Hull

```
const ll is_query = -(1LL<<62);
struct Line {
    ll m, b;
    mutable function<const Line*()> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line* s = succ();
        if (!s) return 0;
        ll x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : public multiset<Line> { // will maintain upper hull for maximum
    bool bad(iterator y) {
        auto z = next(y);
```

```
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (x->b - y->b) * (long double)(z->m - y->m) >= (y->b - z->b) * (long
            ↪ double)(y->m - x->m);
    }
    void insert_line(ll m, ll b) {
        auto y = insert({ m, b });
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll getVal(ll x) {
        auto l = *lower_bound((Line) { x, is_query });
        return l.m * x + l.b;
    }
};
```

Listing 33: Queue Max

```
struct queueMax {
    stack<pair<ll, ll>> s1, s2;
    int size() {
        return s1.size() + s2.size();
    }
    bool isEmpty() {
        return (size() == 0);
    }
    void clear() {
        while(!s1.empty()) {
            s1.pop();
        }
        while(!s2.empty()) {
            s2.pop();
        }
    }
    ll getMax() {
        if(isEmpty()) {
            return -1e18;
        }
        if(!s1.empty() && !s2.empty()) {
            return max(s1.top().second, s2.top().second);
        }
        if(!s1.empty()) {
            return s1.top().second;
        }
        return s2.top().second;
    }
    void push(ll val) {
        if(s2.empty()) {
            s2.push({val, val});
        } else {
            s2.push({val, max(val, s2.top().second)});
        }
    }
    void pop() {
        if(s1.empty()) {
            while(!s2.empty()) {
                if(s1.empty()) {
                    s1.push({s2.top().first, s2.top().first});
                } else {
                    s1.push({s2.top().first, max(s2.top().first, s1.top().second)});
                }
                s2.pop();
            }
        }
        s1.pop();
    }
};
```

Listing 34: Two Sat

```
struct twosat {
    int n;
    vector<vector<int> > adj, adjInv;
    vector<int> scc;
    int sccID;
    vector<bool> visited, assignment;

    twosat(int nodes) {
        n = 2*nodes;
        adj.resize(n);
        adjInv.resize(n);
        scc.resize(n);
        assignment.resize(n/2);
    }

    //X AND Y = (X OR X) AND (Y OR Y)
    //X NAND Y = (!X OR !Y)
    //X NOR Y = (!X OR !X) AND (!Y OR !Y)
    //X XOR Y = (X OR Y) AND (!X OR !Y)
    //X XNOR Y = (!Y OR X) AND (!X OR Y)
    void add(int i, bool statusI, int j, bool statusJ) {
        const int from1 = i+(!statusI)*(n/2);
        const int to1 = j+statusJ*(n/2);
        adj[from1].push_back(to1);
        adjInv[to1].push_back(from1);
        const int from2 = j+(!statusJ)*(n/2);
        const int to2 = i+statusI*(n/2);
        adj[from2].push_back(to2);
        adjInv[to2].push_back(from2);
    }

    void dfs1(int curr, stack<int> &seen) {
        visited[curr] = true;
        for(int x : adj[curr]) {
            if(!visited[x]) {
                dfs1(x, seen);
            }
        }
        seen.push(curr);
    }

    void dfs2(int curr) {
        visited[curr] = true;
        scc[curr] = sccID;
```

```
        for(int x : adjInv[curr]) {
            if(!visited[x]) {
                dfs2(x);
            }
        }
    }

    bool solve() {
        visited.resize(n+1,false);
        stack<int> seen;
        for(int i = 0; i < n; ++i) {
            if(!visited[i]) {
                dfs1(i, seen);
            }
        }
        visited.clear();
        visited.resize(n+1,false);
        sccID = 0;
        while(!seen.empty()) {
            while(!seen.empty() && visited[seen.top()]) seen.pop();
            if(!seen.empty()) {
                dfs2(seen.top());
                sccID++;
            }
        }
        for(int i = 0; i < n/2; ++i) {
            if(scc[i] == scc[i+n/2]) {
                return false;
            }
            assignment[i] = scc[i] < scc[i+n/2];
        }
        return true;
    }
};
```

Listing 35: **RANGE DATA STRUCTURES**

Listing 36: Mo's Algorithm

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

const int Max = 1e6+2;
int block, answer[Max], answerToQuery;

struct query {
    int l, r, index;
};

bool cmp(query x, query y) {
    if(x.l/block == y.l/block) return x.r < y.r;
    return x.l < y.l;
}
void add(int pos) {
}
void remove(int pos) {
}
int main() {
    int q;
    cin >> q;
    vector<query> queries(q);
    for(int i = 0; i < q; ++i) {
        cin >> queries[i].l >> queries[i].r;
        queries[i].index = i;
        answer[i] = 0;
    }
    sort(queries.begin(), queries.end(), cmp);
    int left = 0, right = 0;//store inclusive ranges, start at [0,0]
    add(0);
    answerToQuery = 0;
    for(auto &q : queries) {
        while(left > q.l) {
            left--;
            add(left);
        }
        while(right < q.r) {
            right++;
            add(right);
        }
        while(left < q.l) {
            remove(left);
            left++;
        }
        while(right > q.r) {
            remove(right);
            right--;
        }
        answer[q.index] = answerToQuery;
    }
    for(int i = 0; i < q; ++i) cout << answer[i] << '\n';
    return 0;
}
```

Listing 37: Fenwick Tree

```
struct fenwickTree {
    vector<ll> bit;
    int n;
    fenwickTree() {
        n = (int)1e5+3;
        bit.assign(n,0);
    }
    ll sum(int r) {
        ll ret = 0;
        for(; r >= 0; r = (r&(r+1))-1)
            ret += bit[r];
        return ret;
    }
    void add(int idx, ll d) {
        for(; idx < n; idx = idx | (idx+1))
            bit[idx] += d;
    }
    ll sum(int l, int r) {
        return sum(r) - sum(l-1);
    }
}ft;
```

```
#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
  while(x <= N) {
    tree[x] += v;
    x += (x & -x);
  }
}

// get cumulative sum up to and including x
int get(int x) {
  int res = 0;
  while(x) {
    res += tree[x];
    x -= (x & -x);
  }
  return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
  int idx = 0, mask = N;
  while(mask && idx < N) {
    int t = idx + mask;
    if(x >= tree[t]) {
      idx = t;
      x -= tree[t];
    }
    mask >>= 1;
  }
  return idx;
}
```

Listing 38: Fenwick Tree 2D

```
struct fenwickTree2D {
    vector<vector<ll>> bit;
    int n, m;
    //2D matrix, with n rows, and m columns
    fenwickTree2D(int _n, int _m) {
        n = _n;
        m = _m;
        bit.resize(n+1, vector<ll>(m+1,0));
    }
    //returns sum of rows [0..i], and [0..j] inclusive
    ll sum(int i, int j) {
        ll ret = 0;
        for(; i >= 0; i = (i&(i+1))-1)
            for(int jj = j; jj >= 0; jj = (jj&(jj+1))-1)
                ret += bit[i][jj];
        return ret;
    }
    //adds value to location i,j
    void add(int i, int j, ll d) {
        for(; i <= n; i = i | (i+1))
            for(int jj = j; jj <= m; jj = jj | (jj+1))
                bit[i][jj] += d;
    }
};
```

Listing 39: Search Buckets

```
// Author: Neal Wu
// search_buckets provides two operations on an array:
// 1) set array[i] = x
// 2) count how many i in [start, end) satisfy array[i] < value
// Both operations take sqrt(N log N) time. Amazingly, because of the cache efficiency
    ↪ this is faster than the
// (log N)^2 algorithm until N = 2-5 million.
template<typename T>
struct search_buckets {
    // values are just the values in order. buckets are sorted in segments of
        ↪ BUCKET_SIZE (last segment may be smaller)
    int N, BUCKET_SIZE;
    vector<T> values, buckets;

    search_buckets(const vector<T> &initial = {}) {
        init(initial);
    }

    int get_bucket_end(int bucket_start) const {
        return min(bucket_start + BUCKET_SIZE, N);
    }

    void init(const vector<T> &initial) {
        values = buckets = initial;
        N = values.size();
        BUCKET_SIZE = 3 * sqrt(N * log(N + 1)) + 1;
        for (int start = 0; start < N; start += BUCKET_SIZE)
            sort(buckets.begin() + start, buckets.begin() + get_bucket_end(start));
    }

    int bucket_less_than(int bucket_start, T value) const {
        auto begin = buckets.begin() + bucket_start;
        auto end = buckets.begin() + get_bucket_end(bucket_start);
        return lower_bound(begin, end, value) - begin;
    }

    int less_than(int start, int end, T value) const {
        int count = 0;
        int bucket_start = start - start % BUCKET_SIZE;
        int bucket_end = min(get_bucket_end(bucket_start), end);
        if (start - bucket_start < bucket_end - start) {
            while (start > bucket_start)
                count -= values[--start] < value;
        } else {
            while (start < bucket_end)
                count += values[start++] < value;
        }
        if (start == end) return count;
        bucket_start = end - end % BUCKET_SIZE;
        bucket_end = get_bucket_end(bucket_start);
        if (end - bucket_start < bucket_end - end) {
            while (end > bucket_start)
                count += values[--end] < value;
```

```
        } else {
            while (end < bucket_end)
                count -= values[end++] < value;
        }
        while (start < end && get_bucket_end(start) <= end) {
            count += bucket_less_than(start, value);
            start = get_bucket_end(start);
        }
        return count;
    }

    int prefix_less_than(int n, T value) const {
        return less_than(0, n, value);
    }

    void modify(int index, T value) {
        int bucket_start = index - index % BUCKET_SIZE;
        int old_pos = bucket_start + bucket_less_than(bucket_start, values[index]);
        int new_pos = bucket_start + bucket_less_than(bucket_start, value);
        if (old_pos < new_pos) {
            copy(buckets.begin() + old_pos + 1, buckets.begin() + new_pos,
              ↪ buckets.begin() + old_pos);
            new_pos--;
        } else {
            copy_backward(buckets.begin() + new_pos, buckets.begin() + old_pos,
              ↪ buckets.begin() + old_pos + 1);
        }
        buckets[new_pos] = value;
        values[index] = value;
    }
};
```

Listing 40: Seg Tree

```
struct SegmentTree {
    vector<ll> treeSum, treeMax, lazy;
    ll n, root, size;
    SegmentTree(int currSize) : n(currSize), root(1) {
        ll x = (ll)(ceil(log2(currSize)));
        size = 2*(ll)pow(2, x);
        treeSum.resize(size, 0);
        treeMax.resize(size, 0);
        lazy.resize(size, 0);
    }
    SegmentTree(const vector<ll> &arr) : n(arr.size()), root(1) {
        ll x = (ll)(ceil(log2(n)));
        size = 2*(ll)pow(2, x);
        treeSum.resize(size);
        treeMax.resize(size);
        lazy.resize(size, 0);
        build(arr, root, 0, n-1);
    }
    void build(const vector<ll> &arr, int node, int start, int end) {
        if(start == end) treeMax[node] = treeSum[node] = arr[start];
        else {
            ll mid = (start+end)/2;
            build(arr, 2*node, start, mid);
            build(arr, 2*node+1, mid+1, end);
            treeSum[node] = treeSum[2*node] + treeSum[2*node+1];
            treeMax[node] = max(treeMax[2*node], treeMax[2*node+1]);
        }
    }
    void pendingUpdate(int node, int start, int end) {
        if(lazy[node]) {
            treeSum[node] += (end-start+1) * lazy[node];
            treeMax[node] += lazy[node];
            if(start != end) {
                lazy[2*node] += lazy[node];
                lazy[2*node+1] += lazy[node];
            }
            lazy[node] = 0;
        }
    }
    void updateRange(int l, int r, ll diff) {updateRange(root, 0, n-1, l, r, diff);}
    void updateRange(int node, int start, int end, int l, int r, ll diff) {
        pendingUpdate(node, start, end);
        if(start > end || start > r || end < l) return;
        if(start >= l && end <= r) {
            treeSum[node] += (end-start+1) * diff;
            treeMax[node] += diff;
            if(start != end) {
                lazy[2*node] += diff;
                lazy[2*node+1] += diff;
            }
            return;
        }
        ll mid = (start + end) / 2;
        updateRange(2*node, start, mid, l, r, diff);
        updateRange(2*node+1, mid+1, end, l, r, diff);
        treeSum[node] = treeSum[2*node] + treeSum[2*node+1];
        treeMax[node] = max(treeMax[2*node], treeMax[2*node+1]);
    }
    ll querySum(int l, int r) {return querySum(root, 0, n-1, l, r);}
    ll querySum(int node, int start, int end, int l, int r) {
        if(r < start || end < l) return 0;
        pendingUpdate(node, start, end);
        if(l <= start && end <= r) return treeSum[node];
        ll mid = (start+end)/2;
        return querySum(2*node, start, mid, l, r) + querySum(2*node+1, mid+1, end, l, r);
    }
    ll queryMax(int l, int r) {return queryMax(root, 0, n-1, l, r);}
    ll queryMax(int node, int start, int end, int l, int r) {
        if(r < start || end < l) return -1e18;
        pendingUpdate(node, start, end);
        if(l <= start && end <= r) return treeMax[node];
        ll mid = (start+end)/2;
        return max(queryMax(2*node, start, mid, l, r), queryMax(2*node+1, mid+1, end, l,
          ↪ r));
    }
};
```

Listing 41: Sparse Table

```
struct sparseTable {
    vector<vector<ll> > memo;
    vector<int> logTwo;
    int maxPow;
    sparseTable(const vector<ll> &arr) {
        int n = arr.size();
        logTwo.resize(n+1,0);
```

```
        for(int i = 2; i <= n; ++i) logTwo[i] = 1 + logTwo[i/2];
        maxPow = logTwo[n]+1;
        memo.resize(maxPow, vector<ll>(n));
        for(int j = 0; j < maxPow; ++j) {
            for(int i = 0; i < n; ++i) {
                if(i+(1<<j)-1<n) {
                    memo[j][i] = (j?min(memo[j-1][i], memo[j-1][i+(1<<(j-1))]):arr[i]);
                } else break;
            }
        }
    }
    ll query(int l, int r) {
        int j = logTwo[r-l+1];
        return min(memo[j][l], memo[j][r-(1<<j)+1]);
    }
};
```

Listing 42: Sparse Table 2D

```
//***zero based indexing
int table[11][1010][11][1010],logTwo[1010];
struct sparseTable2D {
    int n, m;
    sparseTable2D(const vector<vector<int>> &Matrix) {
        n = Matrix.size();
        m = Matrix[0].size();
        logTwo[0] = -1;
        for(int i = 1; i <= max(n,m); ++i) {
            logTwo[i] = 1 + logTwo[i/2];
        }
        for(int ir=0; ir<n; ++ir) {
            for(int ic=0; ic<m; ++ic)
                table[0][ir][0][ic] = Matrix[ir][ic];
            for(int jc=1; jc<=logTwo[m]; ++jc)
                for(int ic=0; ic+(1<<(jc-1))<m; ++ic)
                    table[0][ir][jc][ic] = min(table[0][ir][jc-1][ic],
                        ↪ table[0][ir][jc-1][ic+(1<<(jc-1))]);
        }
        for(int jr=1; jr<=logTwo[n]; ++jr)
            for(int ir=0; ir<n; ++ir)
                for(int jc=0; jc<=logTwo[m]; ++jc)
                    for(int ic=0; ic<m; ++ic)
                        table[jr][ir][jc][ic] = min(table[jr-1][ir][jc][ic],
                            ↪ table[jr-1][ir+(1<<(jr-1))][jc][ic]);
    }
    int queryMin(int x1, int y1, int x2, int y2){
        int kx = logTwo[x2-x1+1];
        int ky = logTwo[y2-y1+1];
        int min_R1 = min(table[kx][x1][ky][y1], table[kx][x1][ky][y2+1-(1<<ky)]);
        int min_R2 = min(table[kx][x2+1-(1<<kx)][ky][y1],
            ↪ table[kx][x2+1-(1<<kx)][ky][y2+1-(1<<ky)]);
        return min(min_R1, min_R2);
    }
};
```

Listing 43: **MATH**

Listing 44: Chinese Remainder Theorem

```
inline ll LCM(ll a, ll b) {
    return a / __gcd(a, b) * b;
}

inline ll normalize(ll x, ll mod) {
    x %= mod;
    if (x < 0) x += mod;
    return x;
}

struct gcd_type { ll x, y, d; };

gcd_type ex_gcd(ll a, ll b) {
    if (b == 0) return {1, 0, a};
    gcd_type pom = ex_gcd(b, a % b);
    return {pom.y, pom.x - a / b * pom.y, pom.d};
}

// Chinese remainder theorem: find z such that
// z % mods[i] = vals[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (mods[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
pair<ll, ll> crt(const vector<int> &vals, const vector<int> &mods) {
    for(int i = 0; i < vals.size(); i++) {
        normalize(vals[i], mods[i]);
    }
    ll ans = vals[0];
    ll Lcm = mods[0];
    for(int i = 1; i < vals.size(); i++) {
        auto pom = ex_gcd(Lcm, mods[i]);
        int x1 = pom.x;
        int d = pom.d;
        if((vals[i] - ans) % d != 0) {
            return {0,-1};
        }
        ans = normalize(ans + x1 * (vals[i] - ans) / d % (mods[i] / d) * Lcm, Lcm *
            ↪ mods[i] / d);
        Lcm = LCM(Lcm, mods[i]); // you can save time by replacing above lcm * n[i] /d
            ↪ by lcm = lcm * n[i] / d
    }
    return {ans, Lcm};
}
```

Listing 45: Combinatorics, FastPow, ModInv

```
//returns x such that x*a == 1 (mod b)
//x only exists if gcd(a, b) == 1
ll modInverse(ll a, ll b){
    return 1<a ? b - modInverse(b%a,a)*b/a : 1;
}

//returns a^pw % mod
ll fastPow(ll a, ll pw, ll mod) {
    ll res = 1;
    a %= mod;
    while(pw > 0) {
        if(pw & 1) res = (res*a)%mod;
        a = (a*a)%mod;
```

```
        pw >>= 1;
    }
    return res;
}

struct NchooseK {
    int range, mod;
    vector<ll> fact,ifact;

    NchooseK(int currMod) {
        this->mod = currMod;
        range = (int)1e6+3;
        //range = currMod-1;//TODO: uncomment this to use chooseLucas
        fact.resize(range+1);
        ifact.resize(range+1);
        calcFacts();
    }

    void calcFacts() {
        fact[0] = 1;
        for(ll i = 1; i <= range; ++i) {
            fact[i] = (1LL*fact[i-1]*i)%this->mod;
        }
        ifact[range] = fastPow(fact[range], this->mod-2, this->mod);
        for(int i = range-1; i >= 0; --i) {
            ifact[i] = (1LL*ifact[i+1]*(i+1))%this->mod;
        }
    }

    //helper function for calcChoose
    ll modFact(ll n, ll &e) const {
        if(n <= 1) return 1;
        ll res = modFact(n/this->mod, e);
        e += n/this->mod;
        if ((n/this->mod)%2 == 1) return
            ↪ res*(fact[n%this->mod]*(this->mod-1)%this->mod)%this->mod;
        return res*fact[n%this->mod]%this->mod;
    }

    //n choose k with n >= mod
    //***use prime moduli***
    ll calcChoose(ll n, ll k) const {
        ll e1 = 0, e2 = 0, e3 = 0;
        ll a1 = modFact(n, e1);
        ll a2 = modFact(k, e2);
        ll a3 = modFact(n-k, e3);
        if (e1-e2-e3 > 0) return 0;
        return (a1*fastPow (a2*a3%this->mod, this->mod-2, this->mod)%this->mod);
    }

    //classic n choose k
    //***use prime moduli***
    ll choose(int n, int k) const {
        if(k < 0 || k > n || n < 0) return 0;
        return ((1LL*fact[n]*ifact[k])%this->mod * 1LL*ifact[n-k])%this->mod;
    }

    //lucas theorem to calculate n choose k in O(log(k))
    //***use prime moduli***
    //***can only use with: prime moduli < 1e6***
    ll chooseLucas(ll n, ll k) const {
        if(k < 0 || k > n || n < 0) return 0;
        if(k == 0) return 1;
        int ni = n % this->mod;
        int ki = k % this->mod;
        return (this->chooseLucas(n/this->mod, k/this->mod) * this->choose(ni,ki)) %
            ↪ this->mod;
    }

    //bars and stars problem: given n objects, each with an endless supply,
    //this returns the number of ways to choose k of them.
    ll multiChoose(ll n, ll k) const {
        return chooseLucas(n+k-1, n-1);
    }
};
```

Listing 46: Commonly Used Primes

```
10^3 + {-9,-3,9,13}
10^6 + {-17,3,33}
10^9 + {7,9,21,33,87}
```

Listing 47: Count Primes

```
const int MAX =1000005;
bool prime[MAX];
int prec[MAX];
vector<int> P;

ll rec(ll N, int K) {
    if (N <= 1 || K < 0) return 0;
    if (N <= P[K]) return N-1;
    if (N < MAX && ll(P[K])*P[K] > N) return N-1 - prec[N] + prec[P[K]];

    const int LIM = 250;
    static int memo[LIM*LIM][LIM];

    bool ok = N < LIM*LIM;
    if (ok && memo[N][K]) return memo[N][K];

    ll ret = N/P[K] - rec(N/P[K], K-1) + rec(N, K-1);

    if (ok) memo[N][K] = ret;
    return ret;
}

// init_count_primes();
// count_primes(x);
// Time complexity: Around O(N ^ 0.75)
// Constants to configure:
// - MAX is the maximum value of sqrt(N) + 2
// increase MAX to increase time efficiency
ll count_primes(ll N) {
    if (N < MAX) return prec[N];
    int K = prec[(int)sqrt(N) + 1];
    return N-1 - rec(N, K) + prec[P[K]];
}

void init_count_primes() {
    prime[2] = true;
```

```
    for (int i = 3; i < MAX; i += 2) prime[i] = true;

    for (int i = 3; i*i < MAX; i += 2)
        if (prime[i])
            for (int j = i*i; j < MAX; j += i+i)
                prime[j] = false;
    for(int i = 0; i < MAX; ++i) if (prime[i]) P.push_back(i);
    for(int i = 1; i < MAX; ++i) prec[i] = prec[i-1] + prime[i];
}
```

Listing 48: Discrete Log

```
ll expo(ll x, ll n, ll m){
    ll res = 1;
    while (n){
        if (n & 1) res = (res * x) % m;
        x = (x * x) % m;
        n >>= 1;
    }
    return (res % m);
}

ll extended_gcd(ll a, ll b, ll& x, ll& y){
    /// Bezout's identity, ax + by = gcd(a,b)
    if (!b){
        y = 0, x = 1;
        return a;
    }
    ll g = extended_gcd(b, a % b, y, x);
    y -= ((a / b) * x);
    return g;
}

ll inverse_modulo(ll a, ll m){
    /// inverse exists if and only if a and m are co-prime
    ll x, y, inv;
    extended_gcd(a, m, x, y);
    inv = (x + m) % m;
    return inv;
}

/***
    * returns smallest x such that (g^x) % p = h, -1 if none exists
    * p must be a PRIME
    * function returns x, the discrete log of h with respect to g modulo p
***/
ll discrete_log(ll g, ll h, ll p){
    if (h >= p) return -1;
    if ((g % p) == 0){
        if (h == 1) return 0; /// return -1 if strictly positive lleger solution is
            ↪ required
        else return -1;
    }
    unordered_map <ll, ll> mp;
    ll i, q, r, m = ceil(sqrt(p)); /// may be change to sqrtl() ?
    ll d = 1, inv = expo(inverse_modulo(g, p), m, p);
    for (i = 0; i <= m; i++){
        if (!mp[d]) mp[d] = i + 1;
        d *= g;
        if (d >= p) d %= p;
    }
    d = h;
    for (q = 0; q <= m; q++){
        r = mp[d];
        if (r) return ((m * q) + (--r));
        d *= inv;
        if (d >= p) d %= p;
    }
    return -1;
}
```

Listing 49: Fast Fourier Transform (FFT)

```
/* emaxx implementation */
/* Multiplication with arbitrary modulos
 *    use ntt if mod is prime and can be written as 2**k * c + 1
 *    if not, use Chinese Reminder Theorem
 *    or transform A(x) = A1(x) + A2(x)*c decompose into A(x)/c and A(x)%c
 *                 B(x) = B1(x) + B2(x)*c
 *        where c ~= sqrt(mod)
 *        A * B = A1*B1 + c*(A1*B2 + A2*B1) * c**2(A2*B2)
 *        with all values < sqrt(mod) subpolynomials have coefficientes < mod * N after
     ↪ fft multiply decreasing changes of rounding error
 * */

const double PI=acos(-1);
typedef complex<double> base;
inline void fft (vector<base> & a, bool invert) {
    int n=(int) a.size();
    for (int i=1, j=0; i<n; ++i) {
        int bit=n>>1;
        for (;j>=bit;bit>>=1)
            j-=bit;
        j+=bit;
        if(i<j)
            swap(a[i],a[j]);
    }
    for (int len=2; len<=n; len<<=1) {
        double ang = 2*PI/len * (invert ? -1 : 1);
        base wlen(cos(ang), sin(ang));
        for (int i=0; i<n; i+=len) {
            base w(1);
            for (int j=0; j<len/2; ++j) {
                base u=a[i+j], v=a[i+j+len/2]*w;
                a[i+j]=u+v;
                a[i+j+len/2]=u-v;
                w*=wlen;
            }
        }
    }
    if (invert)
        for(int i=0;i<n;++i)
            a[i]/=n;
}

// a, b => coefs to multiply,  res => resulting coefs
// a[0], b[0], res[0] = coef x^0
inline void multiply (const vector<int> & a, const vector<int> & b, vector<int> & res)
     ↪ {//change res to ll if needed
    if(a.size() * b.size() <= 256) {
```

```
        res.resize(a.size() + b.size(), 0);
        for(int i = 0; i < (int)a.size(); i++)
            for(int j = 0; j < (int)b.size(); j++)
                res[i + j] += 1LL * a[i] * b[j];
        return;
    }
    vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
    size_t n=1;
    while (n<max(a.size(),b.size())) n<<=1;
    n<<=1;
    fa.resize(n),fb.resize(n);
    fft (fa,false);  fft(fb,false);
    for (size_t i=0; i<n; ++i)
        fa[i]*=fb[i];
    fft (fa, true);
    res.resize (n);
    for(size_t i=0; i<n; ++i)
        res[i]=(int)(fa[i].real()>0 ? fa[i].real()+0.5 : fa[i].real()-0.5);
}
```

Listing 50: Fibonacci Sequence

```
unordered_map<ll,ll> table;
int fib(int n) {//**O(log(n))**
    if(n<2) return 1;
    if(table.find(n) != table.end()) return table[n];
    table[n] = (fib((n+1) / 2)*fib(n/2) + fib((n-1) / 2)*fib((n-2) / 2)) % mod;
    return table[n];
}
```

Listing 51: Fractions

```
struct Fraction {
    int a, b;
    Fraction() {
        a = 0;
        b = 1;
    }
    Fraction(int _a, int _b) {
        a = _a;
        b = _b;
        norm();
    }
    Fraction(int x) {
        a = x;
        b = 1;
    }
    Fraction operator + (const Fraction& other) const {
        return Fraction(a * other.b + b * other.a, b * other.b);
    }
    Fraction operator - (const Fraction& other) const {
        return Fraction(a * other.b - b * other.a, b * other.b);
    }
    Fraction operator * (const Fraction& other) const {
        return Fraction(a * other.a, b * other.b);
    }
    Fraction operator / (const Fraction& other) const {
        assert(other.a != 0);
        return Fraction(a * other.b, b * other.a);
    }
    bool operator < (const Fraction& other) const {
        return a*other.b < b*other.a;
    }
    bool operator <= (const int& other) const {
        return (*this < other || *this == other);
    }
    bool operator > (const Fraction& other) const {
        return a*other.b > b*other.a;
    }
    bool operator == (const Fraction& other) const {
        return a*other.b == b*other.a;
    }
    bool operator != (const Fraction& other) const {
        return !(*this == other);
    }
    bool operator == (int& other) const {
        return a == b*other;
    }
    bool operator != (int& other) const {
        return !(*this == other);
    }
    void norm() {
        if (b < 0) {
            a = -a;
            b = -b;
        }
        if (a == 0) b = 1;
        else {
            int g = __gcd(llabs(a), llabs(b));
            a /= g;
            b /= g;
        }
    }
};
istream& operator >> (istream& cin, Fraction& p) {
    cin >> p.a;
    p.b = 1;
    return cin;
}
ostream& operator << (ostream& cout, Fraction& p) {
    cout << p.a << '/' << p.b;
    return cout;
}
Fraction abs(Fraction f) {
    f.a = abs(f.a);
    f.b = abs(f.b);
    return f;
}
```

Listing 52: Gaussian Elimination

```
int Rank;
Fraction Det;
void gauss(vector<vector<Fraction>> &a) {
    int n = a.size();
    int m = a[0].size();//possible RTE here
    Det = 1.0, Rank = 0;
    vector<int> where(max(n, m)+1, -1);
    for(int col = 0, row = 0; col < m && row < n; ++col) {
```

```
        int sel = row;
        for(int i = row+1; i < n; ++i)
            if(abs(a[i][col]) > abs(a[sel][col])) sel = i;
        if(abs(a[sel][col]) == 0) { Det = 0.0; continue; }
        for(int j = 0; j <= m; ++j) swap(a[sel][j], a[row][j]);
        if(row != sel) Det = Fraction(0,1)-Det;
        Det = Det * a[row][col];
        where[col] = row;
        Fraction s = (Fraction(1,1) / a[row][col]);
        for(int j = 0; j <= m; ++j) a[row][j] = a[row][j] * s;
        for(int i = 0; i < n; ++i) if (i != row && abs(a[i][col]) != 0) {
            Fraction t = a[i][col];
            for(int j = 0; j <= m; ++j) a[i][j] = a[i][j] - (a[row][j] * t);
        }
        ++row, ++Rank;
    }
}
```

Listing 53: Matrix Multiplication

```
const ll mod = 1e9+7;
vector<vector<ll> > mult(vector<vector<ll> > a, vector<vector<ll> > b) {
    if(a.size() == 0) return {};
    if(a[0].size() == 0) return {};
    if(b.size() == 0) return {};
    if(b[0].size() == 0) return {};
    if(a[0].size() != b.size()) return {};
    int resultRow = a.size(), resultCol = b[0].size(), n = a[0].size();
    vector<vector<ll> > product(resultRow, vector<ll>(resultCol));
    for(int i = 0; i < resultRow; ++i) {
        for(int j = 0; j < resultCol; ++j) {
            product[i][j] = 0;
            for(int k = 0; k < n; ++k) {
                product[i][j] += a[i][k] * b[k][j];
                product[i][j] %= mod;
            }
        }
    }
    return product;
}

vector<vector<ll> > power(vector<vector<ll> > matrix, ll b) {
    if (b <= 1) return matrix;
    vector<vector<ll> > temp = power(matrix, b/2);
    if (b % 2 == 0) return mult(temp, temp);
    return mult(mult(temp, temp), matrix);
}
```

Listing 54: Pollard Rho

```
ll mulMod(ll x, ll y, ll p) {
    if (y == 0) return 0;
    if (x < 1000111000111000111ll / y) return x * y % p;
    ll mid = mulMod((x+x)%p, y>>1ll, p);
    if (y & 1) return (mid + x) % p;
    else return mid;
}
ll powMod(ll x, ll k, ll m) {
    if (k == 0) return 1;
    if ((k & 1)) return mulMod(x,powMod(x, k-1, m), m);
    else return powMod(mulMod(x,x,m), k/2, m);
}
bool suspect(ll a, ll s, ll d, ll n) {
    ll x = powMod(a, d, n);
    if (x == 1) return true;
    for (int r = 0; r < s; ++r) {
        if (x == n - 1) return true;
        x = mulMod(x, x, n);
    }
    return false;
}
// {2,7,61,-1}                 is for n < 4759123141 (= 2^32)
// {2,3,5,7,11,13,17,19,23,-1} is for n < 10^15 (at least)
bool isPrime(ll n) {
    if (n <= 1 || (n > 2 && n % 2 == 0)) return false;
    ll test[] = {2,3,5,7,11,13,17,19,23,-1};
    ll d = n - 1, s = 0;
    while (d % 2 == 0) ++s, d /= 2;
    for (int i = 0; test[i] < n && test[i] != -1; ++i)
        if (!suspect(test[i], s, d, n)) return false;
    return true;
}
// Killer prime: 5555555557ll (fail when not used mulMod)

ll pollard_rho(ll n, ll seed) { // always call factorize
  ll x, y;
  x = y = rand() % (n - 1) + 1;
  int head = 1, tail = 2;
  while (true) {
    x = mulMod(x, x, n);
    x = (x + seed) % n;
    if (x == y) return n;
    ll d = __gcd(max(x - y, y - x), n);
    if (1 < d && d < n) return d;
    if (++head == tail) y = x, tail <<= 1;
  }
}

void factorize(ll n, vector<ll> &divisor) { // pollard-rho factorization
  if (n == 1) return;
  if (isPrime(n)) divisor.push_back(n);
  else {
    ll d = n;
    while (d >= n) d = pollard_rho(n, rand() % (n - 1) + 1);
    factorize(n / d, divisor);
    factorize(d, divisor);
  }
}
```

Listing 55: Mobius Prime Sieve

```
const int Max = 1e6;
int seive[Max], mob[Max];

//returns 0 iff there exists a prime p s.t. n%(p^2)=0
//returns -1 iff n has an odd number of distinct prime factors
//returns 1 iff n has an even number of distinct prime factors
int mobius(int n) {
    int &temp = mob[n];
```

```
    if(temp != -2) return temp;
    int factors = 0, counter = 1, prev = 0;
    while(n > 1) {
        if(prev == seive[n]) counter++;
        else counter = 1;
        if(counter == 2) {
            return temp = 0;
        }
        prev = seive[n];
        n /= seive[n];
        factors++;
    }
    if(counter == 2) {
        return temp = 0;
    }
    return temp = (factors%2==0?1:-1);
}

void doSeive() {
    for (int i = 0; i < Max; ++i) {
        mob[i] = -2;
        seive[i] = i;
    }
    seive[0] = seive[1] = -1;
    for (int i = 2; i*i < Max; ++i) {
        if (seive[i] == i) {
            for (int j = 2 * i; j < Max; j += i) seive[j] = min(seive[j], i);
        }
    }
}
```

Listing 56: Rabin-Miller Primality Test

```
ll power(ll x, ll y, ll p) {
    ll res = 1;
    x = x % p;
    while (y > 0) {
        if (y & 1) res = (res*x) % p;
        y = y>>1;
        x = (x*x) % p;
    }
    return res;
}

bool miillerTest(ll d, ll n) {
    ll a = 2 + rand() % (n - 4);
    ll x = power(a, d, n);
    if (x == 1  || x == n-1) return true;
    while (d != n-1) {
        x = (x * x) % n;
        d *= 2;
        if (x == 1) return false;
        if (x == n-1) return true;
    }
    return false;
}

// It returns false if n is composite and returns true if n
// is probably prime.  k is an input parameter that determines
// accuracy level. Higher value of k indicates more accuracy.
bool isPrime(ll n, ll k) {
    if (n <= 1 || n == 4)  return false;
    if (n <= 3) return true;
    ll d = n - 1;
    while (d % 2 == 0) d /= 2;
    for (ll i = 0; i < k; i++)
         if (!miillerTest(d, n))
              return false;
    return true;
}
```

Listing 57: Sum Floor Arithmetic Series

```
//computes:
//[p/q] + [2p/q] + [3p/q] + ... + [np/q]
//(p, q, n are natural numbers)
//[x] = floor(x)

ll cnt(ll p, ll q, ll n) {
    ll t = __gcd(p, q);
    p = p/t;
    q = q/t;
    ll s = 0;
    ll z = 1;
    while((q > 0) && (n > 0)) {
        //(point A)
        t = p/q;
        s += z*t*n*(n+1)/2;
        p -= q*t;
        //(point B)
        t = n/q;
        s += z*p*t*(n+1) - z*t*(p*q*t + p + q - 1)/2;
        n -= q*t;
        //(point C)
        t = n*p/q;
        s += z*t*n;
        n = t;
        swap(p,q);
        z = -z;
    }
    return s;
}
```

Listing 58: Sum of Kth Powers

```
#define MAX 1000010
#define MOD 1000000007

//Faulhaber'the sum of the k-th powers of the first n positive integers
//1^k + 2^k + 3^k + 4^k + ... + n^k
//O(k*log(k))

//Usage: lgr::lagrange(n, k)

namespace lgr{
    short factor[MAX];
    int P[MAX], S[MAX], ar[MAX], inv[MAX];

    inline int expo(int a, int b){
```

```
        int res = 1;
        while (b){
            if (b & 1) res = (long long)res * a % MOD;
            a = (long long)a * a % MOD;
            b >>= 1;
        }
        return res;
    }

    int lagrange(long long n, int k){
        if (!k) return (n % MOD);
        int i, j, x, res = 0;
        if (!inv[0]){
            for (i = 2, x = 1; i < MAX; i++) x = (long long)x * i % MOD;
            inv[MAX - 1] = expo(x, MOD - 2);
            for (i = MAX - 2; i >= 0; i--) inv[i] = ((long long)inv[i + 1] * (i + 1)) %
                ↪ MOD;
        }
        k++;
        for (i = 0; i <= k; i++) factor[i] = 0;
        for (i = 4; i <= k; i += 2) factor[i] = 2;
        for (i = 3; (i * i) <= k; i += 2){
            if (!factor[i]){
                for (j = (i * i), x = i << 1; j <= k; j += x){
                    factor[j] = i;
                }
            }
        }
        for (ar[1] = 1, ar[0] = 0, i = 2; i <= k; i++){
            if (!factor[i]) ar[i] = expo(i, k - 1);
            else ar[i] = ((long long)ar[factor[i]] * ar[i / factor[i]]) % MOD;
        }
        for (i = 1; i <= k; i++){
            ar[i] += ar[i - 1];
            if (ar[i] >= MOD) ar[i] -= MOD;
        }
        if (n <= k) return ar[n];
        P[0] = 1, S[k] = 1;
        for (i = 1; i <= k; i++) P[i] = ((long long)P[i - 1] * ((n - i + 1) % MOD)) %
            ↪ MOD;
        for (i = k - 1; i >= 0; i--) S[i] = ((long long)S[i + 1] * ((n - i - 1) % MOD))
            ↪ % MOD;
        for (i = 0; i <= k; i++){
            x = (long long)ar[i] * P[i] % MOD * S[i] % MOD * inv[k - i] % MOD * inv[i] %
                ↪ MOD;
            if ((k - i) & 1){
                res -= x;
                if (res < 0) res += MOD;
            } else{
                res += x;
                if (res >= MOD) res -= MOD;
            }
        }
        return (res % MOD);
    }
}
```

Listing 59: Euler's Totient

```
//  Euler's totient function counts the positive integers
//  up to a given integer n that are relatively prime to n.
ll phi(ll n) {
    ll tempN = n;
    ll result = n;
    for(ll i = 2; i*i <= tempN; i++) {
        if(n % i == 0) {
            while(n % i == 0) {
                n /= i;
            }
            result -= result / i;
        }
    }
    if(n > 1) result -= result / n;
    return result;
}
```

Listing 60: **MISCELLANEOUS**

Listing 61: Random Numbers

```
//Source: http://codeforces.com/blog/entry/61587
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
inline int getRand(int l, int r) {
    uniform_int_distribution<int> uid(l, r);
    return uid(rng);
}
```

Listing 62: Josephus Problem

```

/// Josephus problem, n people numbered from 1 to n stand in a circle.
/// Counting starts from 1 and every k'th people dies
/// Returns the position of the m'th killed people
/// For example if n = 10 and k = 3, then the people killed are 3, 6, 9, 2, 7, 1, 8, 5,
    ↪ 10, 4 respectively

/// O(n)
int josephus(int n, int k, int m){
    int i;
    for (m = n - m, i = m + 1; i <= n; i++){
        m += k;
        if (m >= i) m %= i;
    }
    return m + 1;
}

/// O(k log(n))
long long josephus2(long long n, long long k, long long m){ /// hash = 583016
    m = n - m;
    if (k <= 1) return n - m;

    long long i = m;
    while (i < n){
        long long r = (i - m + k - 2) / (k - 1);
        if ((i + r) > n) r = n - i;
        else if (!r) r = 1;
        i += r;
```

```
        m = (m + (r * k)) % i;
    }
    return m + 1;
}
```

Listing 63: Farey Sequence

```
//Farey sequence, grows n^2/3 with n
def farey_function(n, descending=False):
    """Print the nth Farey sequence, either ascending or descending."""
    a, b, c, d = 0, 1, 1, n
    if descending:
        a, c = 1, n-1
    print "%d/%d" % (a,b)
    while (c <= n and not descending) or (a > 0 and descending):
        k = int((n + b) / d)
        a, b, c, d = c, d, (k*c-a), (k*d-b)
print "%d/%d" % (a,b)
```

Listing 64: Python Test Cases

```
# Random numbers and permutations
# List of N random integers from interval [a,b]:
print( [ randint(a, b) for i in range(N) ] )

# List of N random real numbers from interval [a,b]:
print( [ uniform(a, b) for i in range(N) ] )

# Random permutation of numbers 1 to N:
seq = list(range(1, N+1))
shuffle(seq)
print(seq)

# Random matrices
# Matrix of size NxN with random integers from interval [a,b]:
for r in range(N):
    print( [ randint(a, b) for c in range(N) ] )

# With zeros on diagonal:
for r in range(N):
    print( [ randint(a, b) if r != c else 0 for c in range(N) ] )

# Symmetric matrix:
matrix = [ [0] * N for r in range(N) ]
for r in range(N):
    for c in range(r+1):
        matrix[r][c] = matrix[c][r] = randint(a, b)
print(matrix)

# If you want to output matrix with spaces between the numbers:
for row in matrix:
    print( ' '.join( map(str, row) ) )

# Random tree on N vertices:
# Output is the list of edges. Vertices are numbered from 0 - (N-1).
print( [ (randint(0, i), i+1) for i in range(N-1) ] )

# As Michal šForiek pointed out, the above code will generate rather shallow trees, with
    ↪ expected depth only O(log(N)).
# To generate the deep trees use:
alpha = 3  # affects the depth of the tree. Smaller value generates deeper trees.
# If alpha == 0, then the code will generate a path, i.e. the deepest possible tree.
print( [ (randint(max(0, i-alpha), i), i+1) for i in range(N-1) ] )

# Random graph on N vertices (may be unconnected):
# Output is the list of edges. Vertices are numbered from 0 - (N-1).
print( [ (i,j) for i in range(N) for j in range(i) if randint(0,1) ] )

# Random connected graph on N vertices:
# Simply union the edges of random tree and random graph:
print(set((randint(0, i), i+1) for i in range(N-1)) | set((i,j) for i in range(N) for j
    ↪ in range(i) if randint(0,1)))

# Random string of length N:
# Letters of your choice:
print( ''.join( choice('ABCabc123') for i in range(N) ) )

# Upper-case / lower-case letters:
from string import *
print( ''.join( choice(ascii_uppercase) for i in range(N) ) )  # upper-case
print( ''.join( choice(ascii_lowercase) for i in range(N) ) )  # lower-case

# matching a regex
import re
letters_re = re.compile(r'[A-Za-z0-9]')  # regex of letters we are interested in
print( ''.join(chr(c) for c in range(256) if letters_re.match(chr(c))) )
```

Listing 65: The only thing that matters

```
inoremap kj <esc>
set ts=4 sw=4 sts=0
set expandtab
syntax on
```