**ACM – SCL**

**By Jonathan Shi (DHU)**

**Jan 5, 2020**

Contents

[1 Computational Geometry 1](#_Toc29123093)

[1.1 Basic Definitions 1](#_Toc29123094)

[1.2 Points, Lines & Segments 1](#_Toc29123095)

[1.3 Polygons 2](#_Toc29123096)

[1.4 Convex Hulls & Rotating Calipers 2](#_Toc29123097)

[1.5 Simulating Annealing 3](#_Toc29123098)

[1.6 Closest Pair of Points 3](#_Toc29123099)

[1.7 Half Plane Intersection 4](#_Toc29123100)

[2 Graph Theory 5](#_Toc29123101)

[2.1 Maximum Flow 5](#_Toc29123102)

[2.2 Min Cost Max Flow 5](#_Toc29123103)

[2.3 Bipartite Matching 6](#_Toc29123104)

[2.3.1 Hungarian Algorithm 6](#_Toc29123105)

[2.3.2 KM Algorithm (BFS) 6](#_Toc29123106)

[2.3.3 KM Algorithm (DFS) 7](#_Toc29123107)

[2.4 Strongly Connected Components 8](#_Toc29123108)

[2.4.1 Kosaraju 8](#_Toc29123109)

[2.4.2 Tarjan 8](#_Toc29123110)

[2.5 Biconnected Components, Cut Vertices & Bridges 9](#_Toc29123111)

[2.6 Lowest Common Ancestors 9](#_Toc29123112)

[2.6.1 LCA 9](#_Toc29123113)

[2.6.2 RMQ 10](#_Toc29123114)

[2.6.3 Tarjan 10](#_Toc29123115)

[3 Number Theory 10](#_Toc29123116)

[3.1 Basics 10](#_Toc29123117)

[3.2 Modular Linear Equations 11](#_Toc29123118)

[3.3 Sieve 11](#_Toc29123119)

[3.4 Lucas 12](#_Toc29123120)

[3.5 Discrete Logarithm 13](#_Toc29123121)

[3.6 Primitive Root 13](#_Toc29123122)

[3.7 N-th Root 14](#_Toc29123123)

[3.8 Miller-Rabin & Pollard-Rho 14](#_Toc29123124)

[3.9 Fast Fourier Transform 15](#_Toc29123125)

[3.9.1 FFT 15](#_Toc29123126)

[3.9.2 NTT 16](#_Toc29123127)

[3.9.3 FWT 17](#_Toc29123128)

[4 Matrix 17](#_Toc29123129)

[4.1 Basic Definitions 17](#_Toc29123130)

[4.2 Gaussian Elimination 18](#_Toc29123131)

[5 String 18](#_Toc29123132)

[5.1 KMP 18](#_Toc29123133)

[5.1.1 KMP 18](#_Toc29123134)

[5.1.2 Ext. KMP 19](#_Toc29123135)

[5.2 Manacher 19](#_Toc29123136)

[5.3 Suffix Array 20](#_Toc29123137)

[5.3.1 Solution 20](#_Toc29123138)

[5.3.2 Solution 20](#_Toc29123139)

[5.4 Automaton 21](#_Toc29123140)

[5.4.1 Palindrome Automaton 21](#_Toc29123141)

[5.4.2 Suffix Automaton 22](#_Toc29123142)

[5.4.3 Trie & Aho-Corasick 25](#_Toc29123143)

[6 Tree 25](#_Toc29123144)

[6.1 Binary Indexed Tree 25](#_Toc29123145)

[6.2 Segment Tree 26](#_Toc29123146)

[6.3 Persistent Segment Tree 27](#_Toc29123147)

[6.4 Treap 28](#_Toc29123148)

[6.5 Heavy-Light Decomposition 29](#_Toc29123149)

[6.6 Centroid Decomposition of Tree 30](#_Toc29123150)

[6.7 DSU on Tree 30](#_Toc29123151)

[6.8 Auxiliary Tree 31](#_Toc29123152)

[7 Divide & Conquer 32](#_Toc29123153)

[7.1 CDQ 32](#_Toc29123154)

[7.2 FFT 32](#_Toc29123155)

[7.3 Heuristics 32](#_Toc29123156)

[8 Misc 33](#_Toc29123157)

[8.1 Game 33](#_Toc29123158)

[8.2 Disjoint Sets 34](#_Toc29123159)

[8.3 Sparse Table 34](#_Toc29123160)

[8.4 Hash 34](#_Toc29123161)

[8.5 DLX 34](#_Toc29123162)

[8.6 Counting DP Template 35](#_Toc29123163)

[8.7 Mo’s Algorithm 36](#_Toc29123164)

[8.7.1 Mo’s Algorithm on Sequence 36](#_Toc29123165)

[8.7.2 Mo’s Algorithm on Tree 36](#_Toc29123166)

[8.8 C++ Code Template 37](#_Toc29123167)

[8.9 Java Code Template 37](#_Toc29123168)

[8.10 LIS 38](#_Toc29123169)

[9 References 39](#_Toc29123170)

[9.1 Lucas’s Theorem 39](#_Toc29123171)

[9.2 Facts about primes 39](#_Toc29123172)

[9.3 Möbius Inversion Formula 39](#_Toc29123173)

[9.4 Binomial Transform 39](#_Toc29123174)

[9.5 Wilson’s theorem 40](#_Toc29123175)

[9.6 Euler’s theorem 40](#_Toc29123176)

[9.7 Fermat’s little theorem 40](#_Toc29123177)

[9.8 Fermat’s Last Theorem 40](#_Toc29123178)

[9.9 Catalan number 40](#_Toc29123179)

[9.10 Stirling numbers 40](#_Toc29123180)

[9.10.1 Stirling numbers of the first kind 40](#_Toc29123181)

[9.10.2 Stirling numbers of the second kind 40](#_Toc29123182)

[9.11 Bell number 41](#_Toc29123183)

[9.12 Burnside’s lemma 41](#_Toc29123184)

[9.13 Pólya enumeration theorem 41](#_Toc29123185)

[9.14 Pick's theorem 41](#_Toc29123186)

[9.15 Solving Recurrences by Generating Functions 41](#_Toc29123187)

[9.16 杜教筛 42](#_Toc29123188)

[9.17 类欧几里得 42](#_Toc29123189)

[9.18 博弈论 43](#_Toc29123190)

[9.19 公式 43](#_Toc29123191)

[9.19.1 求和公式 43](#_Toc29123192)

[9.19.2 三角形的面积 44](#_Toc29123193)

[9.19.3 三角函数 44](#_Toc29123194)

# Computational Geometry

## Basic Definitions

struct point { double x, y;

point(double \_x = 0, double \_y = 0) : x(\_x), y(\_y) { }

bool operator==(const point & p) const

{ return fabs(p.x - x) < eps && fabs(p.y - y) < eps; }

bool operator!=(const point & p) const

{ return !(\*this == p); }

point operator-(const point & p) const

{ return point(x - p.x, y - p.y); }

point operator\*(double d) const

{ return point(x \* d, y \* d); }

point operator/(double d) const

{ return point(x / d, y / d); }

};

point operator\*(double d, const point & p) { return p \* d; }

inline double square(double d) { return d \* d; }

double dist(const point & p1, const point & p2)

{ return sqrt(square(p1.x - p2.x) + square(p1.y - p2.y)); }

double dot(const point & p1, const point & p2)

{ return p1.x \* p2.x + p1.y \* p2.y; }

double cross(const point & p1, const point & p2)

{ return p1.x \* p2.y - p2.x \* p1.y; }

struct vec { point from, to;

vec(const point & p1, const point & p2) : from(p1), to(p2) { }

vec(double x = 0, double y = 0) : from(0, 0), to(x, y) { }

vec(const point & p) : from(0, 0), to(p) { }

};

double dot(const vec & v1, const vec & v2)

{ return dot(v1.to - v1.from, v2.to - v2. from); }

double cross(const vec & v1, const vec & v2)

{ return cross(v1.to - v1.from, v2.to - v2.from); }

double cross(const point & o, const point & p1, const point & p2)

{ return cross(vec(o, p1), vec(o, p2)); }

## Points, Lines & Segments

bool is\_point\_on\_line(const vec & v, const point & p)

{ return fabs(cross(vec(v.from, p), v)) < eps; }

bool is\_point\_on\_segment(const vec & v, const point & p) {

return is\_point\_on\_line(v, p) &&

p.x - min(v.from.x, v.to.x) > -eps &&

p.x - max(v.from.x, v.to.x) < eps &&

p.y - min(v.from.y, v.to.y) > -eps &&

p.y - max(v.from.y, v.to.y) < eps;

}

bool are\_lines\_parallel(const vec & v1, const vec & v2)

{ return fabs(cross(v1, v2)) < eps; }

bool are\_lines\_equal(const vec & v1, const vec & v2)

{ return is\_point\_on\_line(v1, v2.from) && is\_point\_on\_line(v1, v2.to); }

bool are\_line\_segment\_cross(const vec & line, const vec & seg)

{ return cross(line, vec(line.from, seg.from)) \* cross(line, vec(line.from, seg.to)) < eps; }

bool are\_segments\_cross(const vec & v1, const vec & v2) {

return max(v1.from.x, v1.to.x) - min(v2.from.x, v2.to.x) > -eps &&

max(v2.from.x, v2.to.x) - min(v1.from.x, v1.to.x) > -eps &&

max(v1.from.y, v1.to.y) - min(v2.from.y, v2.to.y) > -eps &&

max(v2.from.y, v2.to.y) - min(v1.from.y, v1.to.y) > -eps &&

(cross(vec(v2.from, v1.from), v2) \* cross(v2, vec(v2.from, v1.to)) >= eps && cross(vec(v1.from, v2.from), v1) \* cross(v1, vec(v1.from, v2.to)) >= eps || is\_point\_on\_segment(v1, v2.from) || is\_point\_on\_segment(v1, v2.to) || is\_point\_on\_segment(v2, v1.from) || is\_point\_on\_segment(v2, v1.to));

}

bool line\_intersection(const vec & v1, const vec & v2, point & p) {

double D = cross(v1, v2), C1 = cross(vec(v1.from), v1),

C2 = cross(vec(v2.from), v2);

if (fabs(D) < eps) return false;

p = (C2 \* (v1.to - v1.from) - C1 \* (v2.to - v2.from)) / D;

return true;

}

double dist\_between\_point\_and\_seg(const point & p, const vec & seg) {

if (dot(seg, vec(seg.from, p)) > eps && dot(seg, vec(seg.to, p)) < -eps) return fabs(cross(seg, vec(seg.from, p)) / dist(seg.from, seg.to));

else return min(dist(seg.from, p), dist(seg.to, p));

}

## Polygons

double area(point p[], int n) {

double r = 0.0;

for (int i = 0; i < n; i++) r += cross(p[i], p[(i + 1) % n]);

return fabs(r / 2.0);

}

point gravity(point p[], int n) {

point pt, s; double tp, area = 0, tpx = 0, tpy = 0; pt = p[0];

for (int i = 1; i <= n; i++) {

s = p[i == n ? 0 : i]; tp = cross(pt, s);

area += tp / 2.0; tpx += (pt.x + s.x) \* tp;

tpy += (pt.y + s.y) \* tp; pt = s;

}

s.x = tpx / (6 \* area); s.y = tpy / (6 \* area);

return s;

}

int dcmp(double x) { if (x < -eps) return -1; else return x > eps; }

int is\_point\_in\_polygon(const point & pt, point p[], int n) {

int k, d1, d2, wn = 0; p[n] = p[0];

for (int i = 0; i < n; i++) {

if (is\_point\_on\_segment(vec(p[i], p[i + 1]), pt)) return 2;

k = dcmp(cross(p[i], p[i + 1], pt));

d1 = dcmp(p[i + 0].y - pt.y); d2 = dcmp(p[i + 1].y - pt.y);

if (k > 0 && d1 <= 0 && d2 > 0) wn++;

if (k < 0 && d2 <= 0 && d1 > 0) wn--;

}

return wn != 0;

}

## Convex Hulls & Rotating Calipers

vector<point> convex\_hull(vector<point> & p) {

sort(p.begin(), p.end(), [] (const point & p1, const point & p2) {

if (p1.x != p2.x) return p1.x < p2.x; return p1.y < p2.y;

}); int n = p.size(); int k = 0; vector<point> r(n << 1);

for (int i = 0; i < n; i++) {

while (k > 1 && cross(vec(r[k - 2], r[k - 1]), vec(r[k - 1], p[i])) < eps) k--; r[k++] = p[i]; }

for (int i = n - 2, t = k; i >= 0; i--) {

while (k > t && cross(vec(r[k - 2], r[k - 1]), vec(r[k - 1], p[i])) < eps) k--; r[k++] = p[i]; }

r.resize(k - 1);

return r;

}

double rotating\_calipers(point ch[], int n) {

int q = 1; double ans = 0; ch[n] = ch[0];

for (int p = 0; p < n; p++) {

while (cross(ch[p + 1], ch[q + 1], ch[p]) > cross(ch[p + 1], ch[q], ch[p]) + eps) q = (q + 1) % n;

ans = max(ans, max(dist(ch[p], ch[q]), dist(ch[p + 1], ch[q + 1])));

}

return ans;

}

## Simulating Annealing

const double INF = 1e99;

const double delta = 0.98;

const double T = 100;

int dx[4] = { 0, 0, -1, 1 };

int dy[4] = { -1, 1, 0, 0 };

double get\_dist\_sum(point p, point pt[], int n) {

double ans = 0; while (n--) ans += dist(p, pt[n]);

return ans;

}

double fermat\_point(point p[], int n) {

point s = p[0]; double t = T; double ans = INF;

while (t > eps) { bool flag = true;

while (flag) { flag = false;

for (int i = 0; i < 4; i++) {

point z(s.x + dx[i] \* t, s.y + dy[i] \* t);

double tp = get\_dist\_sum(z, p, n);

if (ans > tp) { ans = tp; s = z; flag = true; }

}

}

t \*= delta;

}

return ans;

}

## Closest Pair of Points

double closest\_pair(vector<point>::iterator l, vector<point>::iterator r) { if (r - l <= 1) return INF;

int m = (r - l) >> 1; double x = (l + m)->x;

double d = min(closest\_pair(l, l + m), closest\_pair(l + m, r));

inplace\_merge(l, l + m, r, [] (const point & p1, const point & p2) {

return p1.y - p2.y < -eps;

}); vector<point> v;

for (vector<point>::iterator i = l; i != r; i++) {

if (fabs(i->x - x) - d > -eps) continue;

for (vector<point>::reverse\_iterator j = v.rbegin(); j != v.rend(); j++) {

double dx = i->x - j->x; double dy = i->y - j->y;

if (dy - d > -eps) break;

d = min(d, sqrt(square(dx) + square(dy)));

}

v.push\_back(\*i);

}

return d;

}

double find\_closest\_pair(vector<point> & v) {

sort(v.begin(), v.end(), [] (const point & p1, const point & p2) {

return p1.x - p2.x < -eps;

});

return closest\_pair(v.begin(), v.end());

}

## Half Plane Intersection

bool on\_left(const vec & v, const point & p)

{ return cross(v, vec(v.from, p)) > eps; }

bool on\_left(const vec & v1, const vec & v2)

{ return cross(v2, vec(v2.from, v1.to)) > eps; }

vector<point> half\_plane\_intersection(vector<vec> & v) {

sort(v.begin(), v.end(), [] (const vec & v1, const vec & v2) {

point p1 = v1.to - v1.from, p2 = v2.to - v2.from;

double d = atan2(p1.y, p1.x) - atan2(p2.y, p2.x);

if (fabs(d) < eps) return on\_left(v1, v2);

else return d < -eps;

}); point p; deque<vec> q; deque<point> r; q.push\_back(v[0]);

for (int i = 1; i < v.size(); i++) {

if (are\_lines\_parallel(q.back(), v[i])) continue;

while (r.size() > 0 && !on\_left(v[i], r.back())) {

q.pop\_back(); r.pop\_back();

}

while (r.size() > 0 && !on\_left(v[i], r.front())) {

q.pop\_front(); r.pop\_front();

}

line\_intersection(q.back(), v[i], p);

q.push\_back(v[i]); r.push\_back(p);

}

while (r.size() > 0 && !on\_left(q.front(), r.back())) {

q.pop\_back(); r.pop\_back();

}

line\_intersection(q.front(), q.back(), p); r.push\_back(p);

return vector<point>(r.begin(), r.end());

}

point rotate(const point & p, double d)

{ return point(p.x \* cos(d) - p.y \* sin(d), p.x \* sin(d) + p.y \* cos(d)); }

/\* vector<point> pt(4);

pt[0] = point(-INF, -INF);

pt[1] = point(INF, -INF);

pt[2] = point(INF, INF);

pt[3] = point(-INF, INF); \*/

bool outside(const vec & v, const point & p)

{ return cross(v, vec(v.from, p)) < -eps; }

void cut(vector<point> & pt, const vec & v) {

int n = pt.size(); if (n == 0) return;

pt.push\_back(pt[0]); vector<point> tp; point p;

for (int i = 0; i < n; i++) {

if (!outside(v, pt[i])) tp.push\_back(pt[i]);

else {

if (i == 0 && !outside(v, pt[n - 1])) {

line\_intersection(v, vec(pt[i], pt[n - 1]), p);

tp.push\_back(p);

} if (i != 0 && !outside(v, pt[i - 1])) {

line\_intersection(v, vec(pt[i], pt[i - 1]), p);

tp.push\_back(p);

} if (!outside(v, pt[i + 1])) {

line\_intersection(v, vec(pt[i], pt[i + 1]), p);

tp.push\_back(p);

}

}

}

pt.clear();

for (int i = 0; i < tp.size(); i++)

if (pt.size() == 0 || pt[pt.size() - 1] != tp[i])

pt.push\_back(tp[i]);

}

# Graph Theory

## Maximum Flow

bool isap\_bfs(int s, int t) {

memset(depth, -1, sizeof(depth));

memset(group, 0, sizeof(group));

group[depth[t] = 0]++;

queue<int> q; q.push(t);

while (!q.empty()) {

int u = q.front(); q.pop();

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (~depth[v]) continue;

group[depth[v] = depth[u] + 1]++;

q.push(v);

}

}

return ~depth[s];

}

int isap\_dfs(int u, int t, int c) {

if (u == t) return c; int tmp = c;

for (int i = head[u]; tmp && ~i; i = e[i].next) {

int v = e[i].to;

if (depth[v] + 1 == depth[u] && e[i].cap) {

int d = isap\_dfs(v, t, min(tmp, e[i].cap));

e[i].cap -= d; e[i ^ 1].cap += d; tmp -= d;

}

}

if (tmp) {

if (!--group[depth[u]]) depth[t] = -1;

group[++depth[u]]++;

}

return c - tmp;

}

/\* ISAP (Improved Shortest Augmenting Path) - Maximum Flow

\* Complexity: O(V^2 E) \*/

int isap(int s, int t) {

if (!isap\_bfs(s, t)) return 0; int max\_flow = 0;

while (~depth[t]) max\_flow += isap\_dfs(s, t, INF);

return max\_flow;

}

## Min Cost Max Flow

struct edge { int to, cap, cost, rev; };

vector<edge> g[N];

int dist[N], h[N], prev\_v[N], prev\_e[N];

void add\_edge(int u, int v, int cap, int cost) {

g[u].push\_back((edge){v, cap, cost, g[v].size()});

g[v].push\_back((edge){u, 0, -cost, g[u].size() - 1});

}

int min\_cost\_max\_flow(int s, int t, int f, int n) {

fill(h, h + n, 0); int res = 0;

while (f) { fill(dist, dist + n, INF);

priority\_queue<pii, vector<pii>, greater<pii> > q;

dist[s] = 0; q.push(make\_pair(0, s));

while (!q.empty()) {

pii p = q.top(); q.pop();

int u = p.second;

if (dist[u] < p.first) continue;

for (int i = 0; i < g[u].size(); i++) {

edge & e = g[u][i];

if (e.cap && dist[e.to] > dist[u] + e.cost + h[u] - h[e.to]) {

dist[e.to] = dist[u] + e.cost + h[u] - h[e.to];

prev\_v[e.to] = u; prev\_e[e.to] = i;

q.push(make\_pair(dist[e.to], e.to));

}

}

}

if (dist[t] == INF) break;

for (int u = 0; u < n; u++) h[u] += dist[u];

int d = f;

for (int u = t; u != s; u = prev\_v[u])

d = min(d, g[prev\_v[u]][prev\_e[u]].cap);

f -= d; res += d \* h[t];

for (int u = t; u != s; u = prev\_v[u]) {

edge & e = g[prev\_v[u]][prev\_e[u]];

e.cap -= d; g[u][e.rev].cap += d;

}

}

return res;

}

## Bipartite Matching

### Hungarian Algorithm

/\* The Hungarian method

\* A combinatorial optimization algorithm that solves

\* the assignment problem in polynomial time.

\* Complexity: O(n^3) \*/

bool hungary\_find(int x, int n2) {

for (int i = 0; i < n2; i++) {

if (edges[x][i] && !visited[i]) {

visited[i] = true;

if (match[i] == -1 || hungary\_find(match[i], n2)) {

match[i] = x; return true;

}

}

}

return false;

}

int hungary(int n1, int n2) {

memset(match, -1, sizeof(match)); int r = 0;

for (int i = 0; i < n1; i++) {

memset(visited, 0, sizeof(visited));

if (hungary\_find(i, n2)) r++;

}

return r;

}

### KM Algorithm (BFS)

/\* KM Algorithm (Maximum weighted bipartite matching)

\* Complexity: O(n^3) \*/

int w[N][N]; // INPUT: weights for all edges

int linker[N]; // OUTPUT: matches

int pre[N]; int lx[N], ly[N], slack[N]; bool vis\_y[N];

void km\_bfs(int u, int n) {

for (int i = 0; i <= n; i++) {

pre[i] = 0; slack[i] = INF; vis\_y[i] = false;

} int y = 0, yy = 0; linker[y] = u;

while (true) {

int x = linker[y]; int d = INF; vis\_y[y] = true;

for (int i = 1; i <= n; i++) {

if (!vis\_y[i]) {

if (slack[i] > lx[x] + ly[i] - w[x][i]) {

slack[i] = lx[x] + ly[i] - w[x][i]; pre[i] = y;

} if (slack[i] < d) { d = slack[i]; yy = i; }

}

}

for (int i = 0; i <= n; i++) {

if (vis\_y[i]) { lx[linker[i]] -= d; ly[i] += d;

} else slack[i] -= d;

} y = yy;

if (!linker[y]) break;

}

while (y) { linker[y] = linker[pre[y]]; y = pre[y]; }

}

int km(int n) {

for (int i = 0; i <= n; i++) linker[i] = lx[i] = ly[i] = 0;

for (int i = 1; i <= n; i++) km\_bfs(i, n);

int r = 0;

for (int i = 1; i <= n; i++) r += w[linker[i]][i];

return r;

}

### KM Algorithm (DFS)

ll w[N][N], x[N], y[N], slack[N];

int prev\_x[N], prev\_y[N], son\_y[N], par[N];

void adjust(int v) {

son\_y[v] = prev\_y[v];

if (prev\_x[son\_y[v]] != -2) adjust(prev\_x[son\_y[v]]);

}

bool find(int v, int n) {

for (int i = 0; i < n; i++) {

if (prev\_y[i] == -1) {

if (slack[i] > x[v] + y[i] - w[v][i]) {

slack[i] = x[v] + y[i] - w[v][i]; par[i] = v;

}

if (x[v] + y[i] == w[v][i]) {

prev\_y[i] = v;

if (son\_y[i] == -1) { adjust(i); return true; }

if (prev\_x[son\_y[i]] != -1) continue;

prev\_x[son\_y[i]] = i;

if (find(son\_y[i], n)) return true;

}

}

} return false;

}

ll km(int n) {

for (int i = 0; i < n; i++) {

son\_y[i] = -1; y[i] = 0; x[i] = w[i][0];

for (int j = 1; j < n; j++) x[i] = max(x[i], w[i][j]);

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

prev\_x[j] = prev\_y[j] = -1; slack[j] = INF;

}

prev\_x[i] = -2; if (find(i, n)) continue;

bool flag = false;

while (!flag) {

ll m = INF;

for (int j = 0; j < n; j++)

if (prev\_y[j] == -1) m = min(m, slack[j]);

for (int j = 0; j < n; j++) {

if (prev\_x[j] != -1) x[j] -= m;

if (prev\_y[j] != -1) y[j] += m;

else slack[j] -= m;

}

for (int j = 0; j < n; j++) {

if (prev\_y[j] == -1 && !slack[j]) {

prev\_y[j] = par[j];

if (son\_y[j] == -1) {

adjust(j); flag = true; break;

}

prev\_x[son\_y[j]] = j;

if (find(son\_y[j], n)) {

flag = true; break;

}

}

}

}

}

ll res = 0;

for (int i = 0; i < n; i++) res += w[son\_y[i]][i];

return res;

}

## Strongly Connected Components

### Kosaraju

void scc\_dfs(int u) {

visited[u] = true;

for (int i = 0; i < G[u].size(); i++)

if (!visited[G[u][i]]) scc\_dfs(G[u][i]);

vs.push\_back(u);

}

void scc\_rdfs(int u, int k) {

visited[u] = true; component\_idx[u] = k;

for (int i = 0; i < rG[u].size(); i++)

if (!visited[rG[u][i]]) scc\_rdfs(rG[u][i], k);

}

/\* Kosaraju's algorithm - Strongly Connected Components \*/

int scc(int n) {

memset(visited, 0, sizeof(visited)); vs.clear();

for (int i = 0; i < n; i++)

if (!visited[i]) scc\_dfs(i);

memset(visited, 0, sizeof(visited)); int k = 0;

for (int i = vs.size() - 1; i >= 0; i--)

if (!visited[vs[i]]) scc\_rdfs(vs[i], k++);

return k;

}

### Tarjan

/\* Tarjan's algorithm - Strongly Connected Components

\* In the mathematical theory of Directed Graphs, a graph

\* is said to be Strongly Connected or Diconnected

\* if every vertex is reachable from every other vertex.

\* The Strongly Connected Components or Diconnected Components

\* of an arbitrary directed graph form a partition into subgraphs

\* that are themselves strongly connected.

\* Complexity: O(V + E) \*/

void tarjan(int i) {

int j; dfn[i] = low[i] = idx++;

in\_stack[i] = true; dfs\_stack[++top] = i;

for (vector<int>::iterator e = adj[i].begin(); e != adj[i].end(); e++) { j = \*e;

if (dfn[j] == -1) {

tarjan(j); low[i] = min(low[i], low[j]);

} else if (in\_stack[j]) low[i] = min(low[i], dfn[j]);

}

if (dfn[i] == low[i]) {

component\_number++;

do { j = dfs\_stack[top--]; in\_stack[j] = false;

component[component\_number].push\_back(j);

component\_idx[j] = component\_number;

} while (j != i);

}

}

## Biconnected Components, Cut Vertices & Bridges

/\* Tarjan's algorithm - Biconnected Component

\* In graph theory, a Biconnected Component is

\* a maximal biconnected subgraph.

\* Any connected graph decomposes into a tree of biconnected

\* components called the Block-Cut Tree of the graph.

\* The blocks are attached to each other at shared vertices

\* called Cut Vertices or Articulation Points. Specifically,

\* a Cut Vertex is any vertex whose removal increases the

\* number of connected components.

\* Complexity: O(V + E) \*/

void tarjan(int u, int pre) {

int children = 0; dfn[u] = low[u] = idx++;

for (int i = head[u]; ~i; i = e[i].next) {

if ((i ^ 1) == pre) continue; int v = e[i].to;

if (dfn[v] == -1) {

children++; tarjan(v, i); low[u] = min(low[u], low[v]);

if ((pre == -1 && children > 1) || (pre != -1 && low[v] >= dfn[u])) cut\_vertex.insert(u);

if (low[v] > dfn[u]) bridge.push\_back(make\_pair(u, v));

}

else if (dfn[v] < dfn[u]) low[u] = min(low[u], dfn[v]);

}

}

## Lowest Common Ancestors

### LCA

const int C = (int)log2(N) + 5; int anc[N][C + 1];

// Set pre[init] = init; Set depth[init] = 0;

// Before calling lca\_dfs(init);

void lca\_dfs(int x) {

anc[x][0] = pre[x];

for (int i = 1; i <= C; i++) anc[x][i] = anc[anc[x][i - 1]][i - 1];

for (vector<int>::iterator i = adj[x].begin(); i != adj[x].end(); i++)

if (\*i != pre[x]) {

pre[\*i] = x; depth[\*i] = depth[x] + 1; lca\_dfs(\*i);

}

}

int lca(int x, int y) { if (depth[x] < depth[y]) swap(x, y);

for (int i = C; i >= 0; i--)

if (depth[y] <= depth[anc[x][i]]) x = anc[x][i];

if (x == y) return x;

for (int i = C; i >= 0; i--)

if (anc[x][i] != anc[y][i]) { x = anc[x][i]; y = anc[y][i]; }

return anc[x][0] == anc[y][0] ? anc[x][0] : -1;

}

### RMQ

const int M = log2(N) + 5; vector<int> G[N];

int id[N], vs[N \* 2 - 1], dep[N \* 2 - 1], st[N \* 2 - 1][M];

void rmq\_lca\_dfs(int d, int u, int p, int & k) {

id[u] = k; vs[k] = u; dep[k++] = d;

for (int i = 0; i < G[u].size(); i++) {

if (G[u][i] != p) {

rmq\_lca\_dfs(d + 1, G[u][i], u, k);

vs[k] = u; dep[k++] = d;

}

}

}

void init\_rmq\_lca(int root) {

int k = 0; rmq\_lca\_dfs(0, root, -1, k);

for (int i = 0; i < k; i++) st[i][0] = i;

for (int j = 1; (1 << j) <= k; j++)

for (int i = 0; i + (1 << j) - 1 < k; i++)

if (dep[st[i][j - 1]] <= dep[st[i + (1 << (j - 1))][j - 1]])

st[i][j] = st[i][j - 1];

else st[i][j] = st[i + (1 << (j - 1))][j - 1];

}

int rmq\_lca\_query(int l, int r) {

int k = log2(r - l + 1);

if (dep[st[l][k]] <= dep[st[r - (1 << k) + 1][k]])

return st[l][k];

else return st[r - (1 << k) + 1][k];

}

int rmq\_lca(int u, int v) {

return vs[rmq\_lca\_query(min(id[u], id[v]), max(id[u], id[v]))];

}

### Tarjan

const int M = 1000; // The number of queries.

// lca\_ans should be preset -1.

int father[N], lca\_root[N], lca\_ans[M];

struct lca\_query { int id, x; };

vector<lca\_query> lca\_queries[N];

int find(int x) {

if (father[x] != x) father[x] = find(father[x]);

return father[x];

}

/\* Lowest Common Ancestor (LCA) \*/

void lca\_tarjan(int x, int r) {

lca\_root[x] = r;

for (vector<int>::iterator i = adj[x].begin(); i != adj[x].end(); i++) {

if (lca\_root[\*i] == -1) { lca\_tarjan(\*i, r); father[\*i] = x; }

}

for (vector<lca\_query>::iterator i = lca\_queries[x].begin(); i != lca\_queries[x].end(); i++)

if (lca\_root[i->x] == r) lca\_ans[i->id] = find(i->x);

}

# Number Theory

## Basics

template <class T> T gcd(T a, T b) { return b ? gcd(b, a % b) : a; }

/\* Solves the equation: ax + by = gcd(a, b) \*/

template <class T> T ext\_gcd(T a, T b, T & x, T & y) {

if (!b) { x = 1; y = 0; return a; }

T q = ext\_gcd(b, a % b, y, x);

y -= a / b \* x; return q;

}

/\* Solves the equation: tx = 1 (mod p)

\* which is equivalent to solve the equation: tx + py = 1

\* If x does not exist, then -1 will be returned. \*/

template <class T> T inv(T t, T p) {

T d, x, y; d = ext\_gcd(t, p, x, y);

return d == 1 ? (x % p + p) % p : -1;

}

/\* Euler's totient function

\* Counts the positive integers up to a given integer n

\* that are relatively prime to n. \*/

template <class T> T euler(T n) { T r = n;

for (T i = 2; i \* i <= n; i++) {

if (n % i == 0) {

r = r / i \* (i - 1);

while (n % i == 0) n /= i;

}

}

if (n > 1) r = r / n \* (n - 1);

return r;

}

## Modular Linear Equations

/\* Chinese remainder theorem(CRT)

\* Solves the equations:

\* x = a1 (mod m1)

\* x = a2 (mod m2)

\* ...

\* x = an (mod mn)

\* where a1, a2, ..., an are any integers,

\* and m1, m2, ..., mn are pairwise coprime. \*/

template <class T> T CRT(int n, T a[], T m[]) {

T M = 1, r = 0;

for (int i = 0; i < n; i++) M \*= m[i];

for (int i = 0; i < n; i++) {

T w = M / m[i]; r = (r + w \* inv(w, m[i]) \* a[i]) % M;

}

return (r + M) % M;

}

/\* Solves the equations:

\* A[i] \* x = B[i] (mod M[i])

\* If x does not exist, then (0, -1) will be returned.

\* If x exists, the minimum x and the interval will be returned. \*/

template <class T>

pair<T, T> modular\_linear\_equations(int n, T A[], T B[], T M[]) {

T x = 0, m = 1;

for (int i = 0; i < n; i++) {

T a = A[i] \* m, b = B[i] - A[i] \* x, d = gcd(M[i], a);

if (b % d != 0) return pair<T, T>(0, -1);

T t = b / d \* inv(a / d, M[i] / d) % (M[i] / d);

x += m \* t; m \*= M[i] / d;

}

x = (x % m + m) % m; return pair<T, T>(x, m);

}

## Sieve

/\* Sieve of Eratosthenes

\* Generates a list of primes.

\* With N no more than 10000000.

\* Complexity: O(n log(log(n))) \*/

void sieve\_of\_eratosthenes() {

memset(valid, 0, sizeof(valid));

for (int i = 2; i \* i < N; i++)

for (int j = i; !bit\_val(valid, i) && j \* i < N; j++)

bit\_on(valid, i \* j);

}

/\* Euler's Sieve

\* Generates a list of primes.

\* With N no more than 10000000.

\* Complexity: O(n) \*/

void eulers\_sieve() {

memset(valid, 0, sizeof(valid)); memset(prime, 0, sizeof(prime));

memset(phi, 0, sizeof(phi)); memset(mu, 0, sizeof(mu));

cnt = 0; phi[1] = 1; mu[1] = 1;

for (int i = 2; i < N; i++) {

if (!bit\_val(valid, i)) {

prime[cnt++] = i; phi[i] = i - 1; mu[i] = -1;

}

for (int j = 0; j < cnt && i \* prime[j] < N; j++) {

bit\_on(valid, i \* prime[j]);

if (i % prime[j]) {

phi[i \* prime[j]] = phi[i] \* (prime[j] - 1);

mu[i \* prime[j]] = -mu[i];

} else {

phi[i \* prime[j]] = phi[i] \* prime[j];

mu[i \* prime[j]] = 0; break;

}

}

}

}

## Lucas

/\* p must be a prime \*/

template <typename T> T C(T n, T m, T p) {

if (m > n) return 0; T a = 1, b = 1;

for (T i = n - m + 1; i <= n; ++i) a = a \* i % p;

for (T i = 1; i <= m; ++i) b = b \* i % p;

return a \* mod\_pow(b, p - 2, p) % p;

}

/\* p < 100,000 \*/

template <typename T> T lucas(T n, T m, T p) {

if (m > n) return 0; T r = 1;

for ( ; m; n /= p, m /= p) r = r \* C(n % p, m % p, p) % p;

return r;

}

/\* n! mod p^k, pk = p^k \*/

template <typename T> T fac(T n, T p, T pk) {

if (n <= 1) return 1; T r = 1;

if (n >= pk) {

for (T i = 2; i < pk; i++) if (i % p) (r \*= i) %= pk;

r = mod\_pow(r, n / pk, pk);

}

for (T i = 2; i <= n % pk; i++) if (i % p) (r \*= i) %= pk;

return r \* fac(n / p, p, pk) % pk;

}

template <typename T> T ext\_C(T n, T m, T p, T pk) {

T a = fac(n, p, pk); T b = inv(fac(m, p, pk), pk);

T c = inv(fac(n - m, p, pk), pk); T k = 0;

for (T i = n; i; i /= p) k += i / p;

for (T i = m; i; i /= p) k -= i / p;

for (T i = n - m; i; i /= p) k -= i / p;

return a \* b % pk \* c % pk \* mod\_pow(p, k, pk) % pk;

}

/\* p is not guaranteed to be a prime \*/

template <typename T> T ext\_lucas(T n, T m, T p) {

T r = 0, tmp = p;

for (T i = 2; i \* i <= tmp; i++) {

if (tmp % i == 0) { T pk = 1;

while (tmp % i == 0) { pk \*= i; tmp /= i; }

(r += ext\_C(n, m, i, pk) \* (p / pk) % p \* inv(p / pk, pk) % p) %= p;

}

}

if (tmp > 1) (r += ext\_C(n, m, tmp, tmp) \* (p / tmp) % p \* inv(p / tmp, tmp) % p) %= p;

return r;

}

## Discrete Logarithm

/\* Baby Step Giant Step (BSGS)

\* Finds the smallest non-negative integer x such that

\* a ^ x = b (mod p) where a > 0, b > 0, p > 0 AND (a, p) = 1.

\* Complexity: O(p^0.5) \*/

template <typename T> T BSGS(T a, T b, T p) {

T t = ceil(sqrt(p)); map<T, T> m;

for (T j = 0, x = b % p; j <= t; j++, x = x \* a % p) m[x] = j;

for (T i = 1, y = mod\_pow(a, t, p), x = y; i <= t; i++, x = x \* y % p)

if (m.find(x) != m.end()) return i \* t - m[x];

return -1;

}

/\* Extended Baby Step Giant Step

\* Finds the smallest non-negative integer x such that

\* a ^ x = b (mod p) where a > 0, b > 0, p > 0. \*/

template <typename T> T ext\_BSGS(T a, T b, T p) {

// NOTE: Without the following for-loop, you will get WRONG ANSWER!

for (T i = 0, x = 1 % p, y = b % p; i < 50; i++, x = x \* a % p)

if (x == y) return i;

if (gcd(a, p) == 1) return BSGS(a, b, p);

T n = 0, t = 1, d;

while ((d = gcd(a, p)) != 1) {

if (b % d) return -1;

b /= d; p /= d; n++;

// NOTE: The following line DOES NOT make sense!

t = t \* (a / d) % p;

}

T r = BSGS(a, b \* inv(t, p), p);

return r == -1 ? -1 : r + n;

}

## Primitive Root

template <typename T> bool g\_test(const vector<T> & v, T g, T p) {

for (int i = 0; i < v.size(); i++)

if (mod\_pow(g, (p - 1) / v[i], p) == 1) return false;

return true;

}

/\* p must be a prime. \*/

template <typename T> T primitive\_root(T p) {

T t = p - 1, g = 1; vector<T> v;

for (T i = 2; i \* i <= t; i++)

if (t % i == 0) { v.push\_back(i); while (t % i == 0) t /= i; }

if (t != 1) v.push\_back(t);

while (true) { if (g\_test(v, g, p)) return g; g++; }

}

## N-th Root

/\* x^2 = a (mod n). n is a prime. Complexity: O(log^2 n) \*/

template <typename T> T mod\_sqr(T a, T n) {

T b, k, i, x;

if (n == 2) return a % n;

if (mod\_pow(a, (n - 1) / 2, n) == 1) {

if (n % 4 == 3) x = mod\_pow(a, (n + 1) / 4, n);

else {

for (b = 1; mod\_pow(b, (n - 1) / 2, n) == 1; b++);

i = (n - 1) / 2; k = 0;

do {

i /= 2; k /= 2;

if ((mod\_pow(a, i, n) \* mod\_pow(b, k, n) + 1) % n == 0)

k += (n - 1) / 2;

} while (i % 2 == 0);

x = mod\_pow(a, (i + 1) / 2, n) \* mod\_pow(b, k / 2, n) % n;

}

if (x \* 2 > n) x = n - x;

return x;

}

return -1;

}

/\* Finds all the x (0 <= x < p) such that x ^ n = a (mod p)

\* where p is a prime. Complexity: O(p^0.5) \*/

template <typename T> vector<T> nth\_root(T n, T a, T p) {

vector<T> r;

if (!a) { r.push\_back(0); return r; }

T g = primitive\_root(p);

T m = BSGS(g, a, p);

if (m == -1) return r;

T A = n, B = p - 1, C = m, x, y;

T d = ext\_gcd(A, B, x, y);

if (C % d) return r;

x = x \* (C / d) % B;

T delta = B / d;

for (T i = 0; i < d; i++) {

x = ((x + delta) % B + B) % B;

r.push\_back(mod\_pow(g, x, p));

}

sort(r.begin(), r.end());

r.erase(unique(r.begin(), r.end()), r.end());

return r;

}

## Miller-Rabin & Pollard-Rho

template <class T> T quick\_mult(T a, T b, T mod) {

b %= mod; T r = 0, t = a % mod;

while (b) {

if (b & 1) { r += t; if (r >= mod) r -= mod; }

t <<= 1; if (t >= mod) t -= mod; b >>= 1;

}

return r;

}

template <class T> T quick\_pow(T a, T n, T mod) {

T r = 1, t = a % mod;

while (n) {

if (n & 1) r = quick\_mult(r, t, mod);

t = quick\_mult(t, t, mod);

n >>= 1;

}

return r;

}

/\* Uses a ^ (n - 1) = 1 (mod n) to check whether n is a prime.

\* Returns false if n may be a prime. \*/

template <class T> bool witness(T a, T n, T x, T t) {

T r = quick\_pow(a, x, n), last = r;

for (T i = 1; i <= t; i++) {

r = quick\_mult(r, r, n);

if (r == 1 && last != 1 && last != n - 1) return true;

last = r;

}

if (r != 1) return true;

return false;

}

const int S = 8;

/\* Returns true if n may be a prime.

\* Complexity: O(S log(n)) \*/

template <class T> bool miller\_rabin(T n) {

if (n < 2) return false; if (n == 2) return true;

if ((n & 1) == 0) return false; T x = n - 1, t = 0;

while ((x & 1) == 0) { x >>= 1; t++; }

srand(time(NULL));

for (int i = 0; i < S; i++) {

T a = rand() % (n - 1) + 1;

if (witness(a, n, x, t)) return false;

}

return true;

}

template <class T> T \_gcd(T a, T b) {

T t; while (b) { t = a; a = b; b = t % b; }

if (a >= 0) return a; return -a;

}

/\* Finds a prime factor. \*/

template <class T> T pollard\_rho(T x, int c) {

T i = 1, k = 2; srand(time(NULL));

T x0 = rand() % (x - 1) + 1, y = x0;

while (true) { i++;

x0 = (quick\_mult(x0, x0, x) + c) % x;

T d = \_gcd(y - x0, x);

if (d != 1 && d != x) return d;

if (y == x0) return x;

if (i == k) { y = x0; k += k; }

}

}

/\* Uses Miller-Rabin & Pollard-Rho to find

\* all the prime factors of the given integer n.

\* (Expected) Complexity: O(n^(1/4)) \*/

template <class T, class U>

void find\_prime\_factors(T n, map<T, U> & m, int k = 107) {

if (n == 1) return;

if (miller\_rabin(n)) { m[n]++; return; }

T p = n; int c = k; while (p >= n) p = pollard\_rho(p, c--);

find\_prime\_factors(p, m, k); find\_prime\_factors(n / p, m, k);

}

## Fast Fourier Transform

### FFT

int rev(int idx, int n) { int r = 0;

for (int i = 0; (1 << i) < n; i++) {

r <<= 1; if (idx & (1 << i)) r |= 1;

}

return r;

}

/\* FFT - Fast Fourier Transform

\* Complexity: O(n lg n)

\* Polynomial: A(x) = Sum(a[i] x^i, i = 0 .. n - 1)

\* y = (y[0], y[1], y[n - 1]), where y[k] = A(x[k])

\* a = (a[0], a[1], a[n - 1])

\* y = DFT(a) (DFT - Discrete Fourier Transform)

\* Convolution Theorem:

\* For any two vectors a and b of length n, where n is a power of 2,

\* a (\*) b = DFT'(DFT(a) dot DFT(b))

\* Parameters:

\* The size of a must be a power of 2.

\* If op == 1, DFT, or op == -1, DFT'. \*/

vector<complex<double> > FFT(const vector<complex<double> > & a, int op) { int n = a.size(); vector<complex<double> > v(n);

for (int i = 0; i < n; i++) v[rev(i, n)] = a[i];

for (int s = 1; (1 << s) <= n; s++) { int m = (1 << s);

complex<double> wm = complex<double>(cos(op \* 2 \* acos(-1) / m), sin(op \* 2 \* acos(-1) / m));

for (int k = 0; k < n; k += m) {

complex<double> w = complex<double>(1, 0);

for (int j = 0; j < (m >> 1); j++) {

complex<double> t = w \* v[k + j + (m >> 1)];

complex<double> u = v[k + j];

v[k + j] = u + t; v[k + j + (m >> 1)] = u - t;

w = w \* wm;

}

}

}

if (op == -1)

for (int i = 0; i < n; i++)

v[i] = complex<double>(v[i].real() / n, v[i].imag() / n);

return v;

}

### NTT

/\* Number-theoretic transform (NTT)

\* p = r \* 2^k + 1

\* 23068673 = 11 \* 2^21 + 1 104857601 = 25 \* 2^22 + 1

\* 167772161 = 5 \* 2^25 + 1 469762049 = 7 \* 2^26 + 1

\* 998244353 = 119 \* 2^23 + 1 1004535809 = 479 \* 2^21 + 1 \*/

template <typename T> vector<T> NTT(const vector<T> & a, T p, int op) {

int n = a.size(); T g = primitive\_root(p); vector<T> v(n);

for (int i = 0; i < n; i++) v[rev(i, n)] = a[i];

for (int s = 1; (1 << s) <= n; s++) {

int m = (1 << s); T wm = mod\_pow(g, (p - 1) / m, p);

if (op == -1) wm = mod\_pow(wm, p - 2, p);

for (int k = 0; k < n; k += m) {

T w = 1;

for (int j = 0; j < (m >> 1); j++) {

T t = w \* v[k + j + (m >> 1)] % p;

T u = v[k + j] % p;

v[k + j] = (u + t) % p;

v[k + j + (m >> 1)] = ((u - t) % p + p) % p;

w = w \* wm % p;

}

}

}

if (op == -1) { T inv = mod\_pow(n, p - 2, p);

for (int i = 0; i < n; i++) v[i] = v[i] \* inv % p;

}

return v;

}

### FWT

/\* Fast Walsh-Hadamard transform (FWT)

\* p must be a prime. \*/

void FWT(vector<int> & a, int p, int op) {

const int inv2 = mod\_pow(2LL, p - 2, p);

const int n = a.size();

for (int i = 1; i < n; i <<= 1) {

for (int m = i << 1, j = 0; j < n; j += m) {

for (int k = 0; k < i; k++) {

int x = a[j + k], y = a[i + j + k];

// xor:

if (op == 1) {

a[j + k] = (x + y) % p;

a[i + j + k] = (x + p - y) % p;

} else {

a[j + k] = 1LL \* (x + y) \* inv2 % p;

a[i + j + k] = 1LL \* (x + p - y) \* inv2 % p;

}

// and:

if (op == 1) a[j + k] = (x + y) % p;

else a[j + k] = (x + p - y) % p;

// or:

if (op == 1) a[i + j + k] = (y + x) % p;

else a[i + j + k] = (y + p - x) % p;

}

}

}

}

# Matrix

## Basic Definitions

template <class T> class matrix {

private: int m; int n; vector<vector<T> > a;

public:

static matrix<T> identity\_matrix(int n) {

matrix<T> m(n, n);

for (int i = 0; i < n; i++) m(i, i) = 1;

return m;

}

matrix(int \_m = 0, int \_n = 0)

: m(\_m), n(\_n), a(m, vector<T>(n)) { }

matrix(const initializer\_list<initializer\_list<T> > & v)

: m(v.size()), n(v.begin()->size()), a(m, vector<T>(n)) {

int x = 0;

for (auto i = v.begin(); i != v.end(); i++, x++) {

int y = 0;

for (auto j = i->begin(); j != i->end(); j++, y++)

a[x][y] = \*j;

}

}

int get\_m() const { return m; }

int get\_n() const { return n; }

T & operator()(int x, int y) { return a[x][y]; }

const T & operator()(int x, int y) const { return a[x][y]; }

matrix<T> operator\*(const matrix<T> & b) const {

if (n == b.m) { matrix<T> t(m, b.n);

for (int i = 0; i < m; i++) for (int j = 0; j < b.n; j++)

for (int k = 0; k < n; k++)

t.a[i][j] = ((t.a[i][j] + a[i][k] \* b.a[k][j] % MOD) % MOD + MOD) % MOD;

// (t.a[i][j] += (a[i][k] \* b.a[k][j]) % MOD) %= MOD;

return t;

} else return matrix<T>(0, 0);

}

template <class U> matrix<T> pow(U k) const {

if (m == n) {

matrix<T> t = identity\_matrix(n); matrix<T> b(\*this);

while (k) { if (k & 1) t = t \* b;

b = b \* b; k >>= 1;

}

return t;

} else return matrix<T>(0, 0);

}

};

## Gaussian Elimination

pair<int, vector<pair<bool, double> > > gaussian\_elimination(matrix<double> & mat) {

int m = mat.get\_m(), n = mat.get\_n() - 1, r = 0;

vector<pair<bool, double> > v(n, make\_pair(true, 0));

for (int i = 0; i < n && r < m; i++) {

if (fabs(mat(r, i)) < eps)

for (int j = r + 1; j < m; j++)

if (fabs(mat(j, i)) > eps) {

for (int k = i; k <= n; k++)

swap(mat(j, k), mat(r, k));

break;

}

if (fabs(mat(r, i)) > eps) {

v[i].first = false;

for (int k = n; k >= i; k--) mat(r, k) /= mat(r, i);

for (int j = 0; j < m; j++)

if (j != r && fabs(mat(j, i)) > eps)

for (int k = n; k >= i; k--)

mat(j, k) -= mat(j, i) \* mat(r, k);

r++;

}

}

for (int i = r; i < m; i++)

if (fabs(mat(i, n)) > eps)

return make\_pair(-1, vector<pair<bool, double> >());

for (int i = 0, j = 0; i < n; i++)

if (!v[i].first) { v[i].second = mat(j, n); j++; }

return make\_pair(r == n, v);

}

# String

## KMP

### KMP

vector<int> compute\_fail\_function(const string & pattern) {

vector<int> fail(pattern.size()); fail[0] = 0;

int m = pattern.size(); int i = 1, j = 0;

while (i < m) {

if (pattern[j] == pattern[i]) {

fail[i] = j + 1; i++; j++;

} else if (j > 0) j = fail[j - 1];

else { fail[i] = 0; i++; }

}

return fail;

}

vector<int> KMP\_match(const string & text, const string & pattern) {

vector<int> v; int n = text.size(); int m = pattern.size();

vector<int> fail = compute\_fail\_function(pattern);

int i = 0, j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

if (j == m - 1) { v.push\_back(i - m + 1); j = fail[j];

} else j++; i++;

} else if (j > 0) j = fail[j - 1]; else i++;

}

return v;

}

### Ext. KMP

vector<int> get\_next(const string & pattern) {

int m = pattern.size(); vector<int> next(m);

next[0] = m; int i = 0;

while (i + 1 < m && pattern[i] == pattern[i + 1]) i++;

next[1] = i; int k = 1;

for (i = 2; i < m; i++) {

if (next[i - k] + i < next[k] + k) next[i] = next[i - k];

else { int j = next[k] + k - i;

if (j < 0) j = 0;

while (i + j < m && pattern[j] == pattern[i + j]) j++;

next[i] = j; k = i;

}

}

return next;

}

vector<int> ext\_KMP(const string & text, const string & pattern) {

int n = text.size(); int m = pattern.size();

vector<int> extend(n); vector<int> next = get\_next(pattern);

int i = 0; while (i < n && i < m && pattern[i] == text[i]) i++;

extend[0] = i; int k = 0;

for (i = 1; i < n; i++) {

if (next[i - k] + i < extend[k] + k) extend[i] = next[i - k];

else { int j = extend[k] + k - i; if (j < 0) j = 0;

while (i + j < n && j < m && pattern[j] == text[i + j]) j++;

extend[i] = j; k = i;

}

}

return extend;

}

## Manacher

int manacher(const string & text, char delimiter = 1, char begin\_letter = 2, char end\_letter = 3) { int m = text.size(); int n = (m << 1) + 3;

string str(n, delimiter);

str[0] = begin\_letter;

str[n - 1] = end\_letter;

for (int i = 0; i < m; i++) str[(i << 1) + 2] = text[i];

int p = 0, q = 0, r = 0; vector<int> len(n, 1);

for (int i = 1; i < n - 1; i++) {

if (q > i) len[i] = min(q - i, len[(p << 1) - i]);

while (str[i - len[i]] == str[i + len[i]]) len[i]++;

if (i + len[i] > q) { q = i + len[i]; p = i; }

r = max(r, len[i]);

}

return r - 1;

}

## Suffix Array

### Solution

pair<vector<int>, vector<int> > construct\_sa\_lcp(const vector<int> & s) { const int n = s.size(); vector<int> sa(n), lcp(n), a(n), b(n);

vector<int> \* x = &a, \* y = &b; int m = 0;

for (int i = 0; i < n; i++) { sa[i] = i; m = max(m, s[i]); }

vector<int> cnt(max(m, n) + 5);

for (int i = 0; i < n; i++) cnt[(\*x)[i] = s[i]]++;

for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];

for (int i = n - 1; i >= 0; i--) sa[--cnt[(\*x)[i]]] = i;

for (int k = 1; k <= n; k <<= 1) {

auto cmp = [&] (int i, int j) {

if ((\*y)[i] == (\*y)[j]) {

int ri = i + k < n ? (\*y)[i + k] : -1;

int rj = j + k < n ? (\*y)[j + k] : -1;

return ri < rj;

} else return (\*y)[i] < (\*y)[j];

};

int p = 0;

for (int i = n - k; i < n; i++) (\*y)[p++] = i;

for (int i = 0; i < n; i++)

if (sa[i] >= k) (\*y)[p++] = sa[i] - k;

for (int i = 0; i <= m; i++) cnt[i] = 0;

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

for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];

for (int i = n - 1; i >= 0; i--) sa[--cnt[(\*x)[(\*y)[i]]]] = (\*y)[i];

swap(x, y);

(\*x)[sa[0]] = 0;

for (int i = 1; i < n; i++)

(\*x)[sa[i]] = (\*x)[sa[i - 1]] + cmp(sa[i - 1], sa[i]);

m = (\*x)[sa[n - 1]];

}

for (int i = 0, h = 0; i < n; i++) {

if ((\*x)[i]) {

int j = sa[(\*x)[i] - 1];

while (i + h < n && j + h < n && s[i + h] == s[j + h]) h++;

lcp[(\*x)[i]] = h;

if (h) h--;

}

}

return make\_pair(sa, lcp);

}

### Solution

void radix(int s[], int a[], int b[], int n, int m) {

for (int i = 0; i < n; i++) wv[i] = s[a[i]];

for (int i = 0; i < m; i++) wu[i] = 0;

for (int i = 0; i < n; i++) wu[wv[i]]++;

for (int i = 1; i < m; i++) wu[i] += wu[i - 1];

for (int i = n - 1; i >= 0; i--) b[--wu[wv[i]]] = a[i];

}

inline int F(int x, int tb) { return x / 3 + (x % 3 == 1 ? 0 : tb); }

inline int G(int x, int tb) { return x < tb ? x \* 3 + 1 : (x - tb) \* 3 + 2; }

inline int c0(int s[], int a, int b) { return s[a] == s[b] && s[a + 1] == s[b + 1] && s[a + 2] == s[b + 2]; }

inline int c12(int k, int s[], int a, int b) {

if (k == 2) return s[a] < s[b] || s[a] == s[b] && c12(1, s, a + 1, b + 1);

else return s[a] < s[b] || s[a] == s[b] && wv[a + 1] < wv[b + 1]; }

/\* Makes Suffix Array for s.

\* Complexity: O(n)

\* ATTENTION: Additional space needed!

\* Let N be the maximum size of the given array s.

\* The size of any array should be N \* 3.

\* s[len(s)] = 0 (empty string)

\* n = len(s) + 1 m > max(s[i]) \*/

void dc3(int s[], int sa[], int n, int m) {

int i, j, \* sn = s + n, \* san = sa + n, ta = 0, tb = (n + 1) / 3, tbc = 0, p;

s[n] = s[n + 1] = 0;

for (i = 0; i < n; i++) if (i % 3) wa[tbc++] = i;

radix(s + 2, wa, wb, tbc, m);

radix(s + 1, wb, wa, tbc, m);

radix(s, wa, wb, tbc, m);

for (p = 1, sn[F(wb[0], tb)] = 0, i = 1; i < tbc; i++)

sn[F(wb[i], tb)] = c0(s, wb[i - 1], wb[i]) ? p - 1 : p++;

if (p < tbc) dc3(sn, san, tbc, p);

else for (i = 0; i < tbc; i++) san[sn[i]] = i;

for (i = 0; i < tbc; i++) if (san[i] < tb) wb[ta++] = san[i] \* 3;

if (n % 3 == 1) wb[ta++] = n - 1;

radix(s, wb, wa, ta, m);

for (i = 0; i < tbc; i++) wv[wb[i] = G(san[i], tb)] = i;

for (i = 0, j = 0, p = 0; i < ta && j < tbc; p++)

sa[p] = c12(wb[j] % 3, s, wa[i], wb[j]) ? wa[i++] : wb[j++];

for ( ; i < ta; p++) sa[p] = wa[i++];

for ( ; j < tbc; p++) sa[p] = wb[j++];

}

## Automaton

### Palindrome Automaton

class palindrome\_automaton {

private: vector<vector<int> > ch;

vector<int> fail, len, chr, sz;

int last, n\_chr, n\_node;

int new\_node(int l) {

ch[n\_node].assign(CHAR\_SET, 0);

len[n\_node] = l; return n\_node++;

}

int get\_fail(int x) const {

while (chr[n\_chr - len[x] - 1] != chr[n\_chr]) x = fail[x];

return x;

}

int get\_chr(int c) const { return c; }

public:

explicit palindrome\_automaton(int \_n = N)

: ch(\_n, vector<int>(CHAR\_SET)), fail(\_n),

len(\_n), chr(\_n), sz(\_n) { clear(); }

void clear() {

n\_node = 0; new\_node(0); new\_node(-1);

fail[0] = 1; last = 0;

n\_chr = 0; chr[n\_chr] = -1;

sz.assign(sz.size(), 0);

}

bool add(int c) {

c = get\_chr(c); chr[++n\_chr] = c;

int u = get\_fail(last); bool b = false;

if (!ch[u][c]) { int v = new\_node(len[u] + 2);

fail[v] = ch[get\_fail(fail[u])][c];

ch[u][c] = v; b = true;

}

sz[last = ch[u][c]]++;

return b;

}

int size() const { return n\_node; }

const vector<int> & calc\_size() {

for (int i = n\_node; i >= 0; i--) sz[fail[i]] += sz[i];

return sz;

}

/\* Tip for matching:

\* --------- str[0] = -1, str[1..n], p = 1 ---------

while (p != 1 && (!ch[p][get\_chr(str[i])] || str[i - len[p] - 1] != str[i])) p = fail[p];

if (!ch[p][get\_chr(str[i])]) { p = 1; continue; }

p = ch[p][get\_chr(str[i])]; \*/

};

### Suffix Automaton

#### Standard

class suffix\_automaton {

private: vector<vector<int> > ch;

vector<int> link, len, sz, cnt, rk;

int n, last; int get\_chr(int c) const { return c; }

public:

suffix\_automaton(int \_n = N) : ch(\_n, vector<int>(CHAR\_SET)),

link(\_n), len(\_n), sz(\_n), cnt(\_n), rk(\_n), n(1), last(1) { }

int extend(int c) {

c = get\_chr(c); int p = last, np = ++n;

len[np] = len[p] + 1;

for ( ; p && !ch[p][c]; p = link[p]) ch[p][c] = np;

if (!p) link[np] = 1;

else { int q = ch[p][c];

if (len[q] == len[p] + 1) link[np] = q;

else { int nq = ++n; ch[nq] = ch[q];

link[nq] = link[q]; len[nq] = len[p] + 1;

link[q] = link[np] = nq;

for ( ; p && ch[p][c] == q; p = link[p])

ch[p][c] = nq;

}

}

sz[last = np]++;

return len[last] - len[link[last]];

}

int size() const { return n; }

const vector<int> & calc\_size() {

for (int i = 1; i <= n; i++) cnt[len[i]]++;

for (int i = 1; i <= n; i++) cnt[i] += cnt[i - 1];

for (int i = 1; i <= n; i++) rk[cnt[len[i]]--] = i;

for (int i = n; i >= 1; i--) sz[link[rk[i]]] += sz[rk[i]];

return sz;

}

/\* Tip for matching:

\* --------- str[0..n-1], p = 1 ---------

if (ch[p][get\_chr(str[i])]) { // Accepted!

p = ch[p][get\_chr(str[i])];

} else { // Failed!

while (p != 1 && !ch[p][get\_chr(str[i])])

p = link[p];

if (!ch[p][get\_chr(str[i])]) { continue; }

p = ch[p][get\_chr(str[i])];

} \*/

};

#### Extended

const int M = N \* 2; const int MAXN = M \* 27; const int CHAR\_SET = 26;

int rt[M], ls[MAXN], rs[MAXN], val[MAXN], pos[MAXN], n\_node, m;

void push\_up(int o) {

val[o] = max(val[ls[o]], val[rs[o]]);

if (val[o]) {

if (val[o] == val[ls[o]]) pos[o] = pos[ls[o]];

else pos[o] = pos[rs[o]];

} else pos[o] = 0;

}

int merge(int u, int v, int l, int r) {

if (!u) return v; if (!v) return u; int o = ++n\_node;

if (l == r) {

val[o] = val[u] + val[v];

if (val[o]) pos[o] = l; else pos[o] = 0;

return o;

}

int mid = (l + r) >> 1;

ls[o] = merge(ls[u], ls[v], l, mid);

rs[o] = merge(rs[u], rs[v], mid + 1, r);

push\_up(o); return o;

}

void add(int & o, int l, int r, int x, int y) {

if (!o) o = ++n\_node;

if (l == r) {

val[o] += y;

if (val[o]) pos[o] = l; else pos[o] = 0;

return;

}

int mid = (l + r) >> 1;

if (x <= mid) add(ls[o], l, mid, x, y);

else add(rs[o], mid + 1, r, x, y);

push\_up(o);

}

pair<int, int> get(int o, int l, int r, int x, int y) {

if (x <= l && y >= r) return {pos[o], val[o]};

int mid = (l + r) >> 1;

if (y <= mid) return get(ls[o], l, mid, x, y);

else if (x > mid) return get(rs[o], mid + 1, r, x, y);

else {

auto res1 = get(ls[o], l, mid, x, y);

auto res2 = get(rs[o], mid + 1, r, x, y);

if (res1.second < res2.second) return res2;

else return res1;

}

}

class suffix\_automaton {

private:

int ch[M][CHAR\_SET], link[M], len[M], cnt[M], rk[M], n, last;

int pos[N], anc[M][23];

vector<int> g[M];

int get\_chr(int c) const { return c - 'a'; }

int extend(int c, int id) {

c = get\_chr(c); int p = last;

if (!ch[p][c]) {

int np = ++n; len[np] = len[p] + 1;

for ( ; p && !ch[p][c]; p = link[p]) ch[p][c] = np;

if (!p) link[np] = 1;

else {

int q = ch[p][c];

if (len[q] == len[p] + 1) link[np] = q;

else {

int nq = ++n;

for (int i = 0; i < CHAR\_SET; i++)

ch[nq][i] = ch[q][i];

link[nq] = link[q]; len[nq] = len[p] + 1;

link[q] = link[np] = nq;

for ( ; p && ch[p][c] == q; p = link[p])

ch[p][c] = nq;

}

}

last = np;

} else if (len[ch[p][c]] != len[p] + 1) {

int q = ch[p][c], nq = ++n;

for (int i = 0; i < CHAR\_SET; i++)

ch[nq][i] = ch[q][i];

link[nq] = link[q]; len[nq] = len[p] + 1; link[q] = nq;

for ( ; p && ch[p][c] == q; p = link[p]) ch[p][c] = nq;

last = nq;

} else last = ch[p][c];

if (id) add(rt[last], 1, m, id, 1);

return last;

}

void dfs(int u, int p) {

anc[u][0] = p;

for (int i = 1; i < 23; i++)

anc[u][i] = anc[anc[u][i - 1]][i - 1];

for (auto v : g[u]) dfs(v, u);

}

public:

suffix\_automaton(int m = 0) : n(1), last(1) { }

void insert(const string & s, int id) {

last = 1;

for (int i = 0; i < s.length(); i++) {

if (!id) pos[i + 1] = extend(s[i], id);

else extend(s[i], id);

}

}

/\* Tips for construction:

\* You can also construct S.A. on a tree such as a trie:

\* ------------------------------------------------------------

// build(tree\_root);

void build(int u) {

par[u] = extend(s[u]);

for (auto v : g[u]) {

last = par[u];

build(v);

}

}

\*/

void build() {

for (int i = 2; i <= n; i++) g[link[i]].push\_back(i);

dfs(1, 1);

for (int i = 1; i <= n; i++) cnt[len[i]]++;

for (int i = 1; i <= n; i++) cnt[i] += cnt[i - 1];

for (int i = 1; i <= n; i++) rk[cnt[len[i]]--] = i;

for (int i = n; i >= 1; i--)

rt[link[rk[i]]] = merge(rt[link[rk[i]]], rt[rk[i]], 1, m);

}

pair<int, int> query(int x, int y, int l, int r) {

l = r - l + 1; r = pos[r];

for (int i = 22; i >= 0; i--)

if (len[anc[r][i]] >= l) r = anc[r][i];

return get(rt[r], 1, m, x, y);

}

int size() const { return n; }

};

### Trie & Aho-Corasick

class trie\_2 {

private: vector<vector<int> > ch; vector<int> id, sz, fail, q;

int n, cnt, front, rear; int get\_key(char c) const { return c; }

public:

explicit trie\_2(int \_n = N) : ch(\_n, vector<int>(CHAR\_SET)),

id(\_n), sz(\_n), fail(\_n), q(\_n), n(1), cnt(1) { }

void insert(const char \* str) { int u = 1;

while (true) { sz[u]++;

if (\*str == 0) { if (!id[u]) id[u] = n++; break; }

int v = get\_key(\*str); if (!ch[u][v]) ch[u][v] = ++cnt;

u = ch[u][v]; ++str;

}

}

int find(const char \* str) const { int u = 1;

while (true) { if (\*str == 0) return sz[u];

int v = get\_key(\*str); if (!ch[u][v]) return 0;

u = ch[u][v]; ++str;

}

return 0;

}

void build\_trie\_2() { front = 0; rear = 0;

for (int i = 0; i < CHAR\_SET; i++)

if (ch[1][i]) { fail[ch[1][i]] = 1;

q[rear++] = ch[1][i];

} else ch[1][i] = 1;

while (front != rear) { int u = q[front++];

for (int i = 0; i < CHAR\_SET; i++)

if (ch[u][i]) { fail[ch[u][i]] = ch[fail[u]][i];

q[rear++] = ch[u][i];

} else ch[u][i] = ch[fail[u]][i];

}

}

vector<pair<int, int> > aho\_corasick\_2(const char \* str) const {

vector<pair<int, int> > r; int len = strlen(str), u = 1;

for (int i = 0; i < len; i++) {

int v = get\_key(str[i]); v = u = ch[u][v];

while (v != 1) {

if (id[v]) r.push\_back(make\_pair(id[v], i)); v = fail[v];

}

}

return r;

}

};

# Tree

## Binary Indexed Tree

template <class T> class binary\_indexed\_tree {

private: int N; vector<T> val;

int lowbit(int x) const { return x & -x; }

public:

explicit binary\_indexed\_tree(int n) : N(n + 1), val(N) { }

T query(int n) const { T r = 0;

while (n > 0) { r += val[n]; n -= lowbit(n); }

return r;

}

void update(int i, const T & add) {

while (i < N) { val[i] += add; i += lowbit(i); }

}

};

template <class T> class binary\_indexed\_tree\_2 {

private: binary\_indexed\_tree<T> bit0; binary\_indexed\_tree<T> bit1;

T query\_sum(int n) const

{ return bit1.query(n) \* n + bit0.query(n); }

public:

explicit binary\_indexed\_tree\_2(int n) : bit0(n), bit1(n) { }

T query(int l, int r) const

{ return query\_sum(r) - query\_sum(l - 1); }

void update(int l, int r, const T & add) {

bit0.update(l, -add \* (l - 1)); bit0.update(r + 1, add \* r);

bit1.update(l, add); bit1.update(r + 1, -add);

}

};

## Segment Tree

template <class T> class segment\_tree {

private: int N; vector<T> val; vector<T> lazy;

void push\_down(int root, int istart, int iend) {

if (lazy[root] != 0) {

lazy[root << 1] += lazy[root];

lazy[root << 1 | 1] += lazy[root];

int mid = (istart + iend) >> 1;

val[root << 1] += (mid - istart + 1) \* lazy[root];

val[root << 1 | 1] += (iend - mid) \* lazy[root];

lazy[root] = 0; }

}

void build(int root, const vector<T> & arr, int istart, int iend) {

lazy[root] = 0;

if (istart == iend) val[root] = arr[istart];

else { int mid = (istart + iend) >> 1;

build(root << 1, arr, istart, mid);

build(root << 1 | 1, arr, mid + 1, iend);

val[root] = val[root << 1] + val[root << 1 | 1]; }

}

T query(int root, int istart, int iend, int qstart, int qend) {

if (qstart > iend || qend < istart) return 0;

if (qstart <= istart && qend >= iend) return val[root];

push\_down(root, istart, iend);

int mid = (istart + iend) >> 1;

return query(root << 1, istart, mid, qstart, qend) + query(root << 1 | 1, mid + 1, iend, qstart, qend);

}

void update(int root, int istart, int iend, int ustart, int uend, const T & add) {

if (ustart > iend || uend < istart) return;

if (ustart <= istart && uend >= iend) {

lazy[root] += add;

val[root] += (iend - istart + 1) \* add;

return; }

push\_down(root, istart, iend);

int mid = (istart + iend) >> 1;

update(root << 1, istart, mid, ustart, uend, add);

update(root << 1 | 1, mid + 1, iend, ustart, uend, add);

val[root] = val[root << 1] + val[root << 1 | 1];

}

public:

/\* (ATTENTION: the zero-th position should NOT be used) \*/

segment\_tree(const vector<T> & arr)

: N(arr.size() - 1), val(N << 2), lazy(N << 2)

{ build(1, arr, 1, N); }

T query(int l, int r) { return query(1, 1, N, l, r); }

void update(int l, int r, const T & add) { update(1, 1, N, l, r, add); }

};

## Persistent Segment Tree

class persistent\_segment\_tree {

private: int n; const int M, N, MAXN;

vector<int> root, left, right, sz;

void build(int & rt, int l, int r) {

rt = n++; if (l == r) return; int mid = (l + r) >> 1;

build(left[rt], l, mid); build(right[rt], mid + 1, r);

}

void insert(int & rt, int pre, int l, int r, int x) { rt = n++;

left[rt] = left[pre]; right[rt] = right[pre];

sz[rt] = sz[pre] + 1;

if (l == r) return; int mid = (l + r) >> 1;

if (x <= mid) insert(left[rt], left[pre], l, mid, x);

else insert(right[rt], right[pre], mid + 1, r, x);

}

int query(int u, int v, int l, int r, int k) const {

if (l == r) return l; int mid = (l + r) >> 1;

int x = sz[left[v]] - sz[left[u]];

if (k <= x) return query(left[u], left[v], l, mid, k);

else return query(right[u], right[v], mid + 1, r, k - x);

}

public:

/\* ATTENTION: For any 1 <= i <= n,

\* arr[i] should be between [1, max\_val],

\* where n is the length of arr.

\* The zero-th position should NOT be used! \*/

persistent\_segment\_tree(const vector<int> & arr, int max\_val)

: n(0), M(arr.size()), N(max\_val),

MAXN((N << 2) + M \* ((int)log2(N) + 5)),

root(M), left(MAXN), right(MAXN), sz(MAXN) {

build(root[0], 1, N);

for (int i = 1; i < M; i++)

insert(root[i], root[i - 1], 1, N, arr[i]);

}

int query(int l, int r, int k) const {

return query(root[l - 1], root[r], 1, N, k);

}

};

class persistent\_segment\_tree\_2 {

private: int n; const int M, N, MAXN;

vector<int> root, tree, left, right, sz, val, u, v;

inline int low\_bit(int x) const { return x & -x; }

int sum(const vector<int> & v, int x) const {

int r = 0;

for ( ; x; x -= low\_bit(x)) r += sz[left[v[x]]];

return r;

}

void build(int & rt, int l, int r) { rt = n++;

if (l == r) return; int mid = (l + r) >> 1;

build(left[rt], l, mid); build(right[rt], mid + 1, r);

}

void insert(int & rt, int pre, int l, int r, int x, int y) {

rt = n++;

left[rt] = left[pre]; right[rt] = right[pre];

sz[rt] = sz[pre] + y;

if (l == r) return; int mid = (l + r) >> 1;

if (x <= mid) insert(left[rt], left[pre], l, mid, x, y);

else insert(right[rt], right[pre], mid + 1, r, x, y);

}

int query(int low, int high, int rt1, int rt2, int l, int r, int k) {

if (l == r) return l; int mid = (l + r) >> 1;

int x = sz[left[rt2]] - sz[left[rt1]] + sum(v, high) - sum(u, low - 1);

if (k <= x) {

for (int i = low - 1; i; i -= low\_bit(i)) u[i] = left[u[i]];

for (int i = high; i; i -= low\_bit(i)) v[i] = left[v[i]];

return query(low, high, left[rt1], left[rt2], l, mid, k);

} else {

for (int i = low - 1; i; i -= low\_bit(i)) u[i] = right[u[i]];

for (int i = high; i; i -= low\_bit(i)) v[i] = right[v[i]];

return query(low, high, right[rt1], right[rt2], mid + 1, r, k - x);

}

}

public:

persistent\_segment\_tree\_2(const vector<int> & arr, int max\_val, int num\_update) : n(0), M(arr.size()), N(max\_val),

MAXN((N << 2) + (M + (num\_update << 1) \* (int)log2(M)) \* ((int)log2(N) + 5)), root(M), tree(M), left(MAXN), right(MAXN), sz(MAXN), val(M), u(M), v(M) {

build(root[0], 1, N);

for (int i = 1; i < M; i++) {

tree[i] = root[0];

insert(root[i], root[i - 1], 1, N, val[i] = arr[i], 1);

}

}

void update(int x, int y) {

for (int i = x; i < M; i += low\_bit(i)) {

insert(tree[i], tree[i], 1, N, val[x], -1);

insert(tree[i], tree[i], 1, N, y, 1);

}

val[x] = y;

}

int query(int l, int r, int k) {

for (int i = l - 1; i; i -= low\_bit(i)) u[i] = tree[i];

for (int i = r; i; i -= low\_bit(i)) v[i] = tree[i];

return query(l, r, root[l - 1], root[r], 1, N, k);

}

};

## Treap

class treap {

int ls[N], rs[N], sz[N], pri[N], val[N], cnt[N], rt, n, s[N], top;

int new\_node() { if (n < N) return n++; return s[--top]; }

void free\_node(int x) { s[top++] = x; }

void rotate\_left(int & o) {

int k = ls[o]; ls[o] = rs[k];

sz[o] = sz[ls[o]] + cnt[o] + sz[rs[o]];

rs[k] = o; o = k;

sz[o] = sz[ls[o]] + cnt[o] + sz[rs[o]];

}

void rotate\_right(int & o) {

int k = rs[o]; rs[o] = ls[k];

sz[o] = sz[ls[o]] + cnt[o] + sz[rs[o]];

ls[k] = o; o = k;

sz[o] = sz[ls[o]] + cnt[o] + sz[rs[o]];

}

void \_insert(int & o, int x) {

if (!o) { o = new\_node(); ls[o] = rs[o] = 0; sz[o] = 1;

pri[o] = rand(); val[o] = x; cnt[o] = 1; return; }

if (x < val[o]) \_insert(ls[o], x);

else if (val[o] < x) \_insert(rs[o], x);

else cnt[o]++;

sz[o] = sz[ls[o]] + cnt[o] + sz[rs[o]];

if (pri[ls[o]] < pri[o]) rotate\_left(o);

if (pri[rs[o]] < pri[o]) rotate\_right(o);

}

void \_erase(int & o, int x) {

if (!o) return;

if (x < val[o]) \_erase(ls[o], x);

else if (val[o] < x) \_erase(rs[o], x);

else if (cnt[o] > 1) cnt[o]--;

else if (ls[o] && rs[o]) {

int ptr = ls[o]; while (rs[ptr]) ptr = rs[ptr];

val[o] = val[ptr]; cnt[o] = cnt[ptr];

cnt[ptr] = 1; \_erase(ls[o], val[o]);

} else { int t = o; o = ls[o] ? ls[o] : rs[o]; free\_node(t); }

sz[o] = sz[ls[o]] + cnt[o] + sz[rs[o]];

}

int \_get\_kth(int o, int k) {

if (!o) return -1;

if (k <= sz[ls[o]]) return \_get\_kth(ls[o], k);

else if (k <= sz[ls[o]] + cnt[o]) return val[o];

else return \_get\_kth(rs[o], k - sz[ls[o]] - cnt[o]);

}

void walk\_tree(int o) const {

if (!o) return; walk\_tree(ls[o]);

for (int i = 0; i < cnt[o]; i++) cout << val[o] << ' ';

walk\_tree(rs[o]);

}

public:

void init() { pri[0] = INT\_MAX; top = 0; n = 1; rt = 0; }

void insert(int x) { \_insert(rt, x); }

void erase(int x) { \_erase(rt, x); }

int get\_kth(int k) { return \_get\_kth(rt, k); }

void walk\_tree() const { walk\_tree(rt); cout << endl; }

};

## Heavy-Light Decomposition

// dfs1(1, 0, 0); dfs2(1, 1);

// Build linear data structure with: arr[rk[i]]

int fa[N], son[N], sz[N];

void dfs1(int u, int p, int d) {

depth[u] = d; fa[u] = p; sz[u] = 1;

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to; if (v == p) continue;

dfs1(v, u, d + 1); sz[u] += sz[v];

if (son[u] == -1 || sz[v] > sz[son[u]])

son[u] = v; }

}

int top[N], id[N], rk[N], pos = 1;

void dfs2(int u, int t) {

top[u] = t; id[u] = pos; rk[pos++] = u;

if (son[u] == -1) return; dfs2(son[u], t);

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (v != son[u] && v != fa[u])

dfs2(v, v); }

}

void operate\_path(int u, int v) {

int fu = top[u], fv = top[v];

while (fu != fv) {

if (depth[fu] >= depth[fv]) {

operate(id[fu], id[u]); u = fa[fu];

} else { operate(id[fv], id[v]); v = fa[fv]; }

fu = top[u]; fv = top[v];

} if (u != v) {

if (id[u] < id[v]) operate(id[u], id[v]);

else operate(id[v], id[u]);

} else operate(id[u], id[v]);

}

## Centroid Decomposition of Tree

void get\_centroid(int u, int p) {

sz[u] = 1; msz[u] = 0;

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (v == p || is\_centroid[v]) continue;

get\_centroid(v, u);

sz[u] += sz[v];

msz[u] = max(msz[u], sz[v]);

}

msz[u] = max(msz[u], tree\_size - sz[u]);

if (centroid == -1 || msz[u] < msz[centroid])

centroid = u;

}

// tree\_size = n;

// get\_centroid(1, centroid = -1);

// solve(centroid);

void solve(int u) {

get\_centroid(u, centroid = -1);

is\_centroid[u] = true;

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (is\_centroid[v]) continue;

tree\_size = sz[v];

get\_centroid(v, centroid = -1);

solve(centroid);

}

}

## DSU on Tree

int sz[N], son[N];

void dfs1(int u, int p) {

sz[u] = 1;

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to; if (v == p) continue;

dfs1(v, u); sz[u] += sz[v];

if (son[u] == -1 || sz[v] > sz[son[u]])

son[u] = v; }

}

bool big[N]; int num[N], col[N]; ll ans[N], sum, mx;

void add(int u, int p, int x) {

num[col[u]] += x;

if (x > 0) {

if (num[col[u]] > mx) {

mx = num[col[u]]; sum = col[u];

} else if (num[col[u]] == mx) sum += col[u];

}

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (v != p && !big[v])

add(v, u, x); }

}

// dfs1(1, -1); dfs2(1, -1, false);

void dfs2(int u, int p, bool keep) {

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (v != p && v != son[u])

dfs2(v, u, false); }

if (~son[u]) {

dfs2(son[u], u, true);

big[son[u]] = true; }

add(u, p, 1); ans[u] = sum;

if (~son[u]) big[son[u]] = false;

if (!keep) { add(u, p, -1); sum = mx = 0; }

}

## Auxiliary Tree

int l[N], r[N];

namespace original\_tree {

struct edge {int to, next;} e[N << 1];

const int C = (int)log2(N) + 5;

int head[N], cnt, dep[N], anc[N][C + 1];

void add(int u, int v) { e[cnt] = {v, head[u]}; head[u] = cnt++; }

void add\_edge(int u, int v) { add(u, v); add(v, u); }

// dfs(0, root, root, k);

void dfs(int d, int u, int p, int & k) {

l[u] = k++;

dep[u] = d; anc[u][0] = p;

for (int i = 1; i <= C; i++)

anc[u][i] = anc[anc[u][i - 1]][i - 1];

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to;

if (v == p) continue;

dfs(d + 1, v, u, k);

}

r[u] = k++;

}

int lca(int u, int v) {

if (dep[u] < dep[v]) swap(u, v);

for (int i = C; i >= 0; i--)

if (dep[v] <= dep[anc[u][i]]) u = anc[u][i];

if (u == v) return u;

for (int i = C; i >= 0; i--) {

if (anc[u][i] != anc[v][i]) {

u = anc[u][i]; v = anc[v][i];

}

}

return anc[u][0] == anc[v][0] ? anc[u][0] : -1;

}

}

namespace auxiliary\_tree {

struct edge {int to, next;} e[N];

int head[N], cnt, a[N << 1], n, b[N], s[N], top;

void add(int u, int v) { e[cnt] = {v, head[u]}; head[u] = cnt++; }

void build() {

auto cmp = [] (int i, int j) { return l[i] < l[j]; };

sort(a, a + n, cmp);

for (int k = n, i = 0; i < k - 1; i++)

a[n++] = original\_tree::lca(a[i], a[i + 1]);

sort(a, a + n, cmp);

n = unique(a, a + n) - a;

top = 0;

s[top++] = a[0];

for (int i = 1; i < n; i++) {

while (r[s[top - 1]] < l[a[i]]) top--;

add(s[top - 1], a[i]);

s[top++] = a[i];

}

}

void dfs(int u) {

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to; dfs(v);

}

}

void solve() {

// Input: n, a

for (int i = 0; i < n; i++) b[a[i]] = 1;

build(); dfs(a[0]); // tree DP dfs

// Output: e.g. dp[a[0]]

for (int i = 0; i < n; i++) {

head[a[i]] = -1;

b[a[i]] = 0;

}

cnt = 0;

}

}

# Divide & Conquer

## CDQ

// no duplicated values; sort by a first; cdq(0, cnt - 1);

void cdq(int l, int r) {

if (l == r) return; int mid = (l + r) >> 1;

cdq(l, mid); cdq(mid + 1, r); int p = l, q = mid + 1, cnt = l;

while (p <= mid && q <= r) {

if (d[p].b <= d[q].b) {

update(d[p].c, d[p].w); t[cnt++] = d[p++];

} else {

d[q].f += query(d[q].c); t[cnt++] = d[q++];

}

}

while (p <= mid) { update(d[p].c, d[p].w); t[cnt++] = d[p++]; }

while (q <= r) { d[q].f += query(d[q].c); t[cnt++] = d[q++]; }

for (int i = l; i <= mid; i++) update(d[i].c, -d[i].w);

for (int i = l; i <= r; i++) d[i] = t[i];

}

## FFT

/\* Given g[1], ..., g[n - 1], find f[0], ..., f[n - 1],

\* where f[i] = sum(f[i - j] \* g[j], j = 1..i), and f[0] = 1.

\* f[0] = 1; solve(0, n - 1);

\* Complexity: O(n log^2 n) \*/

void solve(int l, int r) {

if (l == r) return; int mid = (l + r) >> 1; solve(l, mid);

int sz = 1; while (sz <= (mid + r - 2 \* l - 1)) sz <<= 1;

for (int i = l; i <= mid; i++) a[i - l] = f[i];

for (int i = mid - l + 1; i < sz; i++) a[i] = 0;

for (int i = 1; i <= r - l; i++) b[i - 1] = c[i];

for (int i = r - l; i < sz; i++) b[i] = 0;

NTT(a, sz, 1); NTT(b, sz, 1);

for (int i = 0; i < sz; i++) a[i] = a[i] \* b[i] % p;

NTT(a, sz, -1);

for (int i = mid + 1; i <= r; i++) f[i] = (f[i] + a[i - l - 1]) % p;

solve(mid + 1, r);

}

## Heuristics

#include <iostream> #include <cstdio> #include <cmath>

using namespace std; const int N = 300005;

typedef long long ll; int a[N], A[N], B[N], vis[N], st[N][23], k;

int query(int l, int r) {

l--; r--; int k = log2(r - l + 1);

if (a[st[l][k]] >= a[st[r - (1 << k) + 1][k]]) return st[l][k];

else return st[r - (1 << k) + 1][k];

}

ll ans;

void solve(int l, int r) {

if (l > r) return;

int pos = query(l, r);

if (pos - l < r - pos) {

for (int i = l, len = a[pos] - k; i <= pos; i++) {

int no\_repeat\_rightmost = min(B[i], r);

int valid\_leftmost = max(i + len - 1, pos);

if (no\_repeat\_rightmost < valid\_leftmost) continue;

ans += no\_repeat\_rightmost - valid\_leftmost + 1;

}

} else {

for (int i = pos, len = a[pos] - k; i <= r; i++) {

int no\_repeat\_leftmost = max(A[i], l);

int valid\_rightmost = min(i - len + 1, pos);

if (valid\_rightmost < no\_repeat\_leftmost) continue;

ans += valid\_rightmost - no\_repeat\_leftmost + 1;

}

}

solve(l, pos - 1);

solve(pos + 1, r);

}

int main() {

int T, n; scanf("%d", &T);

while (T--) {

scanf("%d%d", &n, &k);

for (int i = 1; i <= n; i++) {

scanf("%d", &a[i]);

vis[i] = 0;

}

A[1] = 1;

for (int i = 2; i <= n; i++) {

if (vis[a[i]]) A[i] = max(A[i - 1], vis[a[i]] + 1);

else A[i] = A[i - 1];

vis[a[i]] = i;

}

for (int i = 1; i <= n; i++) vis[i] = 0;

B[n] = n;

for (int i = n - 1; i >= 1; i--) {

if (vis[a[i]]) B[i] = min(B[i + 1], vis[a[i]] - 1);

else B[i] = B[i + 1];

vis[a[i]] = i;

}

for (int i = 0; i < n; i++) st[i][0] = i + 1;

for (int j = 1; j < 23; j++) {

for (int i = 0; i + (1 << j) - 1 < n; i++) {

if (a[st[i][j - 1]] >= a[st[i + (1 << (j - 1))][j - 1]]) st[i][j] = st[i][j - 1];

else st[i][j] = st[i + (1 << (j - 1))][j - 1];

}

}

ans = 0; solve(1, n); printf("%lld\n", ans);

}

return 0;

}

# Misc

## Game

vector<int> get\_SG(const vector<int> & v, int n) { vector<int> SG(n);

for (int i = 1; i < n; i++) { set<int> s;

for (const auto & x : v)

if (x <= i) s.insert(SG[i - x]);

for (int j = 0; ; j++)

if (!s.count(j)) { SG[i] = j; break; }

}

return SG;

}

## Disjoint Sets

class disj\_sets {

private: vector<int> s;

public: explicit disj\_sets(int n) : s(n, -1) { }

int find(int x) { return s[x] < 0 ? x : s[x] = find(s[x]); }

void union\_sets(int x, int y) {

int root1 = find(x), root2 = find(y);

if (root1 == root2) return;

if (s[root2] < s[root1]) s[root1] = root2;

else {

if (s[root1] == s[root2]) --s[root1];

s[root2] = root1;

}

}

};

## Sparse Table

const int M = (int)log2(N) + 5; int st[N][M];

void init\_st(int a[], int n) {

for (int i = 0; i < n; i++) st[i][0] = a[i];

for (int j = 1; (1 << j) <= n; j++)

for (int i = 0; i + (1 << j) - 1 < n; i++)

st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);

}

int query\_min(int l, int r) {

int k = (int)log2(r - l + 1);

return min(st[l][k], st[r - (1 << k) + 1][k]);

}

## Hash

ull BKDR\_hash(const char \* str) {

// seed can be 31, 131, 1313, 13131, 131313, ...

const ull seed = 131313; ull hash = 0;

while (\*str) hash = hash \* seed + (\*str++);

return hash;

}

ull b[N], h[N];

void init\_hash(const char \* str) {

const ull seed = 131313; b[0] = 1; h[0] = 0;

for (int i = 1; \*str; i++) {

b[i] = b[i - 1] \* seed;

h[i] = h[i - 1] \* seed + (\*str++);

}

}

ull BKDR\_hash(int l, int r)

{ return h[r + 1] - h[l] \* b[r - l + 1]; }

## DLX

struct DLX { const int M; int n, m, sz, n\_ans;

vector<int> U, D, L, R, col, row, H, S, ans;

DLX(int \_n, int \_m) : M(\_n \* \_m + \_m + 5), n(\_n), m(\_m),

sz(m), n\_ans(-1), U(M), D(M), L(M), R(M),

col(M), row(M), H(n + 1, -1), S(m + 1), ans(n) {

for (int i = 0; i <= m; i++) {

U[i] = D[i] = i; L[i] = i - 1; R[i] = i + 1; }

L[0] = m; R[m] = 0;

}

void add(int r, int c) {

++S[col[++sz] = c]; row[sz] = r;

U[sz] = c; D[sz] = D[c];

U[D[c]] = sz; D[c] = sz;

if (~H[r]) {

L[sz] = H[r]; R[sz] = R[H[r]];

L[R[H[r]]] = sz; R[H[r]] = sz;

} else H[r] = L[sz] = R[sz] = sz;

}

void remove(int c) {

L[R[c]] = L[c]; R[L[c]] = R[c];

for (int i = D[c]; i != c; i = D[i])

for (int j = R[i]; j != i; j = R[j]) {

U[D[j]] = U[j]; D[U[j]] = D[j];

--S[col[j]]; }

}

void restore(int c) {

for (int i = U[c]; i != c; i = U[i])

for (int j = L[i]; j != i; j = L[j])

++S[col[U[D[j]] = D[U[j]] = j]];

L[R[c]] = R[L[c]] = c;

}

bool dance(int d = 0) {

if (R[0] == 0) { n\_ans = d; return true; }

int c = R[0];

for (int i = R[0]; i; i = R[i])

if (S[i] < S[c]) c = i;

remove(c);

for (int i = D[c]; i != c; i = D[i]) {

ans[d] = row[i];

for (int j = R[i]; j != i; j = R[j]) remove(col[j]);

if (dance(d + 1)) return true;

for (int j = L[i]; j != i; j = L[j]) restore(col[j]);

} restore(c);

return false;

}

};

## Counting DP Template

ll dfs(int pos, int state, bool lead, bool limit) {

if (pos == -1) return 1;

if (!limit && !lead && dp[pos][state] != -1) return dp[pos][state];

int up = limit ? a[pos] : 9; ll ans = 0;

for (int i = 0; i <= up; i++)

ans += dfs(pos - 1, state,

lead && i == 0, limit && i == a[pos]);

if (!limit && !lead) dp[pos][state] = ans;

return ans;

}

ll solve(ll x) {

int pos = 0;

while (x) { a[pos++] = x % 10; x /= 10; }

return dfs(pos - 1, STATE0, true, true);

}

## Mo’s Algorithm

### Mo’s Algorithm on Sequence

/\* Mo's Algorithm. Complexity: O(n^(3/2)) \*/

void MO\_2(int n, int m) {

const int block = sqrt(n);

sort(q, q + m, [&] (const query & a, const query & b) {

return a.l / block == b.l / block ? a.r < b.r : a.l < b.l;

});

for (int i = 0, l = 1, r = 0; i < m; i++) {

while (l < q[i].l) sub(l++); while (l > q[i].l) add(--l);

while (r < q[i].r) add(++r); while (r > q[i].r) sub(r--);

res[q[i].id] = ans;

}

}

/\* Mo's Algorithm. Complexity: O(n^(5/3)) \*/

void MO\_3(int n, int m) {

const int block = pow(n, 2.0 / 3.0);

sort(q, q + m, [&] (const query & a, const query & b) {

if (a.l / block != b.l / block) return a.l < b.l;

if (a.r / block != b.r / block) return a.r < b.r;

return a.t < b.t;

});

for (int l = 0, r = -1, t = 0, i = 0; i < m; i++) {

while (l < q[i].l) sub(l++); while (l > q[i].l) add(--l);

while (r > q[i].r) sub(r--); while (r < q[i].r) add(++r);

while (t < q[i].t) upd(i, t++); while (t > q[i].t) upd(i, --t);

res[q[i].id] = ans;

}

}

### Mo’s Algorithm on Tree

struct query { int u, v, id, lca, l, r; } q[N];

int s[N], t[N], vs[N \* 2], vis[N], res[N], ans;

void \_add(int u) { num[col[u]]++; if (num[col[u]] == 1) ans++; }

void \_sub(int u) { num[col[u]]--; if (num[col[u]] == 0) ans--; }

void add(int u) { if (vis[u]) \_sub(u); else \_add(u); vis[u] ^= 1; }

void dfs(int u, int p, int & k) {

s[u] = ++k; vs[k] = u;

for (int i = head[u]; ~i; i = e[i].next) {

int v = e[i].to; if (v == p) continue; dfs(v, u, k);

} t[u] = ++k; vs[k] = u;

}

void MO\_2\_on\_tree(int m) {

int k = 0; dfs(1, -1, k);

for (int i = 0; i < m; i++) {

if (s[q[i].u] > s[q[i].v]) swap(q[i].u, q[i].v);

q[i].id = i; q[i].lca = lca(q[i].u, q[i].v);

if (q[i].lca == q[i].u) {

q[i].l = s[q[i].u]; q[i].r = s[q[i].v]; q[i].lca = -1;

} else { q[i].l = t[q[i].u]; q[i].r = s[q[i].v]; }

} const int block = sqrt(k);

sort(q, q + m, [&] (const query & a, const query & b) {

return a.l / block == b.l / block ? a.r < b.r : a.l < b.l;

});

for (int i = 0, l = 1, r = 0; i < m; i++) {

while (l < q[i].l) add(vs[l++]);

while (l > q[i].l) add(vs[--l]);

while (r < q[i].r) add(vs[++r]);

while (r > q[i].r) add(vs[r--]);

if (~q[i].lca) add(q[i].lca);

res[q[i].id] = ans;

if (~q[i].lca) add(q[i].lca);

}

}

## C++ Code Template

#include <bits/stdc++.h> #include <bits/extc++.h> //#include <ext/rope>

//#include <ext/pb\_ds/assoc\_container.hpp>

//#include <ext/pb\_ds/priority\_queue.hpp>

using namespace std; using namespace chrono;

using namespace \_\_gnu\_cxx; using namespace \_\_gnu\_pbds;

const int N = 1000005; const int INF = 0x3f3f3f3f;

const double PI = acos(-1); const double eps = 1e-8;

#define ms(x, y) memset((x), (y), sizeof(x))

#define mc(x, y) memcpy((x), (y), sizeof(y))

typedef long long ll; typedef unsigned long long ull;

#define fi first #define se second #define mp make\_pair

typedef pair<int, int> pii; typedef pair<ll, int> pli;

#define bg begin #define ed end #define pb push\_back

#define al(x) (x).bg(), (x).ed() #define st(x) sort(al(x))

#define un(x) (x).erase(unique(al(x)), (x).ed())

#define fd(x, y) (lower\_bound(al(x), (y)) - (x).bg() + 1)

#define ls(x) ((x) << 1) #define rs(x) (ls(x) | 1)

/// int order\_of\_key(T);

/// iterator find\_by\_order(int);

template <typename T>

using rbtree = tree<T, null\_type, less<T>, rb\_tree\_tag,

tree\_order\_statistics\_node\_update>;

/// point\_iterator push(T);

/// void modify(point\_iterator, T);

template <typename T>

using pheap = \_\_gnu\_pbds::priority\_queue<T, greater<T>, pairing\_heap\_tag>;

template <class T> bool read\_int(T & x) { char c;

while (!isdigit(c = getchar()) && c != '-' && c != EOF);

if (c == EOF) return false; T flag = 1;

if (c == '-') { flag = -1; x = 0; } else x = c - '0';

while (isdigit(c = getchar())) x = x \* 10 + c - '0';

x \*= flag; return true; }

template <class T, class ...R> bool read\_int(T & a, R & ...b) {

if (!read\_int(a)) return false; return read\_int(b...); }

mt19937 gen(steady\_clock::now().time\_since\_epoch().count());

int main() {

time\_point<steady\_clock> start = steady\_clock::now();

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr); cout.tie(nullptr); cerr.tie(nullptr);

// int size = 256 << 20; // 256 M

// char \* p = (char \*)malloc(size) + size;

// #if (defined \_WIN64) or (defined \_\_unix)

// \_\_asm\_\_("movq %0, %%rsp\n" :: "r"(p));

// #else

// \_\_asm\_\_("movl %0, %%esp\n" :: "r"(p));

// #endif

// int T, n; scanf("%d", &T); while (T--) { scanf("%d", &n); }

cerr << endl << "------------------------------" << endl << "Time: "

<< duration<double, milli>(steady\_clock::now() - start).count()

<< " ms." << endl;

// exit(0);

return 0;

}

## Java Code Template

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

public class Main {

public static void main(String[] args) {

init(); Integer x;

while ((x = nextInt()) != null) {

System.out.println(x);

}

}

public static BufferedReader reader;

public static StringTokenizer tokenizer;

public static void init() {

reader = new BufferedReader(new InputStreamReader(System.in), 32768);

tokenizer = null;

}

public static String next() {

while (tokenizer == null || !tokenizer.hasMoreTokens()) {

try {

tokenizer = new StringTokenizer(reader.readLine());

} catch (Exception e) {

return null;

}

}

return tokenizer.nextToken();

}

public static String nextLine() {

try {

return reader.readLine();

} catch (IOException e) {

return null;

}

}

public static Integer nextInt() {

try {

return Integer.valueOf(next());

} catch (Exception e) {

return null;

}

}

}

## LIS

int LIS() {

int len = 1; dp[1] = a[1]; pos[1] = 1;

for (int i = 2; i <= n; i++) {

if (a[i] >= dp[len]) { // without "=" if strictly increasing

dp[++len] = a[i]; pos[i] = len;

} else {

// lower\_bound if strictly increasing

int idx = upper\_bound(dp + 1, dp + len + 1, a[i]) - dp;

dp[idx] = a[i]; pos[i] = idx;

}

}

int mx = INF, tmp = len;

for (int i = n; tmp && i >= 1; i--) {

if (pos[i] == tmp && mx >= a[i]) { // without "=" if strictly increasing

mx = a[i];

tmp--;

}

}

return len;

}

# References

## Lucas’s Theorem

For non-negative integers and and a prime , the following congruence relation holds:

where

and

are the base expansions of and respectively. This uses the convention that

## Facts about primes

* The th prime, , is about times the natural log of :
* The number of primes not exceeding we have what’s known as the *Prime number theorem*:

## Möbius Inversion Formula

If , then .

If , then .

where is the *Möbius function*.

For any positive integer , define as the sum of the primitive th roots of unity. It has values in depending on the factorization of into prime factors:

* if is a square-free positive integer with an even number of prime factors.
* is is a square-free positive integer with an odd number of prime factors.
* if has a squared prime factor.
* Related equations:

## Binomial Transform

The *binomial transform* takes the sequence to the sequence via the transformation

The inverse transform is

## Wilson’s theorem

In number theory, *Wilson’s theorem* states that a natural number is a prime number if and only if the product of all the positive integers less is one less than a multiple of . That is, the factorial statisfies

exactly when is a prime number.

## Euler’s theorem

In number theory, *Euler’s theorem* (a.k.a. the *Fermat-Euler theorem* or *Euler’s totient theorem*) states that if and are coprime positive integers, then

where is *Euler’s totient function*.

* Related equation:

## Fermat’s little theorem

*Fermat’s little theorem* states that if is a prime number, then for any integer ,

If is not divisible by ,

## Fermat’s Last Theorem

In number theory, *Fermat’s Last Theorem* (sometimes called *Fermat’s conjecture*) states that no three positive integers , , and satisfy the equation for any integer value of greater than 2.

## Catalan number

The Catalan numbers satisfy the recurrence relations

and

An alternative expression for is

Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, counts the number of expressions containing pairs of parentheses which are correctly matched.

## Stirling numbers

### Stirling numbers of the first kind

The symbol stands for the number of ways to arrange objects into cycles.

### Stirling numbers of the second kind

The symbol stands for the number of ways to partition a set of things into nonempty subsets.

## Bell number

The number of ways a set of elements can be partitioned into nonempty subsets is called a *Bell number* and is denoted .

The integers can be defined by the sum:

The *Bell numbers* can also be generated using the sum and recurrence relation:

*Touchard’s congruence* states:

when is prime.

## Burnside’s lemma

Let be a finite group that acts on a set . For each in let denote the set of elements in that are fixed by (also said to be left invariant by ). Burnside’s lemma asserts the following formula for the number of orbits, denoted :

## Pólya enumeration theorem

If is a bracelet of beads in a circle, is a finite set of colors – the colors of the beads – so that is the set of colored arrangements of beads, then the group acts on . The Pólya enumeration theorem counts the number of orbits under of colored arrangements of beads by the following formula:

where is the number of colors and is the number of cycles of the group element when considered as a permutation of .

## Pick's theorem

Given a simple polygon (consisting of straight, non-intersecting line segments or “sides” that are joined pair-wise to form a closed path) constructed on a grid of equal-distanced points (i.e., points with integer coordinates) such that all the polygon’s vertices are grid points, *Pick’s theorem* provides a simple formula for calculating the area of the polygon in terms of the number of lattice points in the interior located in the polygon and the number of lattice points on the boundary placed on the polygon’s perimeter:

## Solving Recurrences by Generating Functions

* Steps:

1. Write down a single equation that expresses . The equation should be valid for all integers , assuming that .
2. Multiply both sides of the equation by and sum over all . The left side will be .
3. Solve the resulting equatin, getting a closed form for .
4. Expand into a power series and read off the coefficient of ; this is a closed form for .

* Comments:

In step 4, usually, we will have

where

Let

then

* Rational Expansion Theorem for Distinct Roots

If , where and the numbers are distinct, and if is a polynomial of degree less than , then

where

* General Expansion Theorem for Rational Generating Functions

If , where and the numbers are distinct, and if is a polynomial of degree less than , then

where each is a polynomial of degree with leading coefficient

## 杜教筛

设积性函数的前缀和，即，寻找一个积性函数，让与作卷积，

并计算前缀和，

得，

复杂度：

## 类欧几里得

设

则

* **相关公式**
* **证明**

## 博弈论

1. **巴什博奕(Bash Game)：只有一堆n个物品，两个人轮流从这堆物品中取物，规定每次至少取一个，最多取m个。最后取光者得胜。**

**结论：**如果n=(m+1)r+s，(r为任意自然数，s≤m)，那么先取者要拿走s个物品，如果后取者拿走k(≤m)个，那么先取者再拿走m+1-k个，结果剩下(m+1)(r-1)个，以后保持这样的取法，那么先取者肯定获胜。总之，要保持给对手留下(m+1)的倍数，就能最后获胜。那么这个时候只要n%(m+1)!=0,先取者一定获胜。

1. **威佐夫博奕(Wythoff Game)：有两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。**

**结论：**若两堆物品的初始值为(x, y)，且x<y，则令z=y-x;

记w=(int)[((sqrt(5)+1)/2)\*z];

若w=x，则先手必败，否则先手必胜。

1. **尼姆博弈(Nimm Game)：有任意堆物品，每堆物品的个数是任意的，双方轮流从中取物品，每一次只能从一堆物品中取部分或全部物品，最少取一件，取到最后一件物品的人获胜。**

**结论：**把每堆物品数全部异或起来，如果得到的值为0，那么先手必败，否则先手必胜。

1. **斐波那契博弈：有一堆物品，两人轮流取物品，先手最少取一个，至多无上限，但不能把物品取完，之后每次取的物品数不能超过上一次取的物品数的二倍且至少为一件，取走最后一件物品的人获胜。**

**结论：**先手胜当且仅当n不是斐波那契数（n为物品总数）

## 公式

### 求和公式

### 三角形的面积

1. **三角形面积公式（行列式）**
2. **海伦公式**

### 三角函数

1. **基本定义及属性**
2. **两角和差**
3. **倍角公式**
4. **半角公式**
5. **万能公式**
6. **和差化积**
7. **积化和差**
8. **辅助角公式**
9. **正弦定理**

其中为外接圆半径

1. **余弦定理**