Contents

Geometry 5

pointOnLine 5

pointOnRay 5

pointOnSegment 5

retuen y of point on line given the line and x of this point 5

return the y of a point that falling up segments untile it fall to ground (as water fall) 6

prependecularVecor 6

paralelVector 6

check if a point inside the rectangle 6

check if a point inside the circle 6

check if a point inside the triangle 7

line using it's a, b, c 7

Point To Segment 7

check if to lines are parallel 7

check if to lines are the same 7

check if to lines are the same using points 7

check if 2 lines are intersecting 7

get angle (AOB): in radian 8

Triangle side given other side and angle 8

Triangle angel given other side and angle 8

Line Inside Rectangle 8

check if 2 segments intersect 8

check if a line intersects with a rectangle using its (left dawn) and (right up) points 8

convert the angle from degree to radian 9

convert the angle from radian to degree 9

find angle Anticlockwise from v1 to v2 9

find angle p2p0p1, anti-clock p0p1 to p0p2 9

get vector as a point from it length and angle 9

get vector with length R according to a Ray ab 9

get triangle Area using 3 points 9

given the area of rectangle using 3 medians 9

check Overlapping Rectangles using (left down) and (right up) point for each 9
Given are the (x, y) coordinates of the endpoints of two adjacent sides of a parallelogram. Find the (x, y) coordinates of the fourth point. 10
get all possible points that complete the parallelogram with another 3 points(triangle) 10
check if a polygon with angles = A is regular polygon 10
Check if 4 point create a Rectangle 10
Check if 4 point create a Square 10
Point distance to Line 10
Point distance to segment 10
given 2 diagonal points for the square find the other two 11
Circle rad given 2 points and cen 11
circleCircleIntersectionArea 11
distance between two points in arc 11
Segment and Segment intersection 11
3D geometry 12
3D Point 12
The volume of Triangular Pyramid 12
Data structure 14
Fenwick tree 14
Fenwick tree 2d 15
Fenwick tree max 15
Fenwick tree update range 16
Segment tree 17
Segment tree without lazy 18
Max sum range node 19
Ordered set 19
Sparse table 20
SQRT Decomposition 20
Implicit treap 22
Ordered multiset 24
Heavy light decomposition 26
LCA 28
Centroid decomposition 29

DSU 30
DSU apps 31
BST 32
AVL 34
Heap 36
DSU bipartiteness 36
Big int 37
Graph 40
Kruskal 40
Prim 41
SMST 41
Dijkstra 44
Floyed 45
Difference constraints 46
SPFA 47
SCC 47
articulation_points_and_bridges 48
Edge classification 49
2-sat 49
Maximum bipartite matching 50
String 51
Hashing 51
KMP 51
Trie tree 52
Suffix array 53
Aho corasick 54
Math 56
Combinatorics 56
Matrices 57
Matrix class 58
Mod inverse 59
(a^n)%p=result , return n 60
NCR pre calculation 61

Primes 61
Summations 62
Misc 63
Bitmask 63
coordinateCompress........ 64
Random numbers 64
Custom hash 64
Max histogram area........ 64
Sorting........ 65
LIS binary Search........ 66
Mo algorithm 67
floyd cycle detection algorithm 67

Geometry

```
#include<bits/stdc++.h>
using namespace std;
typedef complex<long double> point;
#define sz(a) ((int)(a).size())
#define all(n) (n).begin(),(n).end()
#define EPS 1e-9
#define OO 1e9
#define PI acos(-1.0)
#define X real()
#define Y imag()
#define vec(a,b) ((b)-(a))
#define polar(r,t) ((r)*exp(point(0,(t))))
#define angle(v) (atan2((v).Y,(v).X))
#define length(v) ((long double)hypot((v).Y,(v).X))
#define lengthSqr(v) (dot(v,v))
#define dot(a,b) ((conj(a)*(b)).real())
#define cross(a,b) ((conj(a)*(b)).imag())
#define rotate(v,t) (polar(v,t))
#define rotateabout(v,t,a) (rotate(vec(a,v),t)+(a))
#define reflect(p,m) ((conj((p)/(m)))*(m))
#define normalize(p) ((p)/length(p))
#define same(a,b) (lengthSqr(vec(a,b))<EPS)
#define mid(a,b) (((a)+(b))/point(2,0))
#define perp(a) (point(-(a).Y,(a).X))
#define colliner pointOnLine
enum STATE {IN, OUT, BOUNDRY};
int dcmp(double x, double y) { return fabs(x - y) <= EPS ? 0 : x < y ? -1 : 1; }
double fixAngle(double A) { return A > 1 ? 1 : (A < -1 ? -1 : A);}
double fixMod(double a, double b) {return fmod(fmod(a, b) + b, b);}
```

pointOnLine

```
bool pointOnLine(const point& a, const point& b, const point& p) {

	return fabs(cross(vec(a, b), vec(a, p))) < EPS;
}
```

pointOnRay

```
inline bool pointOnRay(point a, point b, point p) {
	return dot(vec(a, b), vec(a, p)) > -EPS && pointOnLine(a, b, p);
}
```

pointOnSegment

```
inline bool pointOnSegment(point a, point b, point p) {
	return dot(vec(a, b), vec(a, p)) > -EPS && pointOnLine(a, b, p) && dot(vec(b, a), vec(b, p)) > -
EPS;
}
```

retuen y of point on line given the line and x of this point

```
long double getYfromXforline(point a, point b, long double x) {
	if (a.X == b.X)return b.Y;
	long double m = (a.Y - b.Y) / (a.X - b.X);
	long double c = a.Y - m * a.X;
	return m * x + c;
}
```

// retuen x of point on line given the line and y of this point

```
long double getXfromYforline(point a, point b, long double y) {
	if (a.X == b.X)return b.X;
	long double m = (a.Y - b.Y) / (a.X - b.X);
	long double c = a.Y - m * a.X;
```

```
	return (y - c) / m;
}

namespace std {
	bool operator <(const point& a, const point& b) {
		return a.X != b.X ? a.X < b.X : a.Y < b.Y;
	}
	bool operator >(const point& a, const point& b) {
		return a.X != b.X ? a.X > b.X : a.Y > b.Y;
	}
	// sortsegments butome up , note the point of 1 segment must be sorted
	// the functions that coled must be up of this poisition in code
	bool operator <(const vector<point>& a, const vector<point>& b) {
		point p(a[0].X, getYfromXforline(b[0], b[1], a[0].X));
		if (pointOnSegment(b[0], b[1], p))return a[0].Y < p.Y;
		p = point(a[1].X, getYfromXforline(b[0], b[1], a[1].X));
		if (pointOnSegment(b[0], b[1], p))return a[1] < p;
		p = point(b[0].X, getYfromXforline(a[0], a[1], b[0].X));
		if (pointOnSegment(a[0], a[1], p))return p < b[0];
		p = point(b[1].X, getYfromXforline(a[0], a[1], b[1].X));
		if (pointOnSegment(a[0], a[1], p))return p < b[1];
		return max(a[0].Y, a[1].Y) < max(b[0].Y, b[1].Y);
	}
}
```

return the y of a point that falling up segments untile it fall to ground (as water fall)

```
double waterFallOnsegments(vector< vector<point> > v, point p) {
	for (int i = (int)v.size() - 1; i >= 0; i--) {
		double x = p.X, y = getYfromXforline(v[i][0], v[i][1], p.X);
		if (y <= p.Y&&pointOnSegment(v[i][0], v[i][1], point(x, y))) {
			if (v[i][0].Y < v[i][1].Y)p = v[i][0];
			else p = v[i][1];
		}
	}
	return p.X;
}
point readpoint() { long double x, y; cin >> x >> y; return point(x, y); }
void printpoint(point P) { cout << P.X << " " << P.Y; }
// somtimes point is -0.0 it is ronge mmust be 0,0
void fixpoint(point &p) {
	if (p.X<EPS&&p.X>-EPS)p = point(0, p.Y);
	if (p.Y<EPS&&p.Y>-EPS)p = point(p.X, 0);
}
```

prependecularVecor

```
bool prependecularVec(const point &a, const point &b) {
	return dot(a, b) == 0;
}
```

paralelVector

```
bool paralelVec(const point &a, const point &b) {
	return cross(a, b) == 0;
}
```

check if a point inside the rectangle

```
bool pointInsidReqt(point LD, point RU, point P) {
	//note that ">= and <=" not "> and <" it based on your problem...
	return P.X >= LD.X && P.X <= RU.X && P.Y >= LD.Y && P.Y <= RU.Y;
}
```

check if a point inside the circle

```
bool pointInsidCircle(point center, long double rad, point P) {
```

```
        long double d = length(vec(center, P));
        return d<rad;//note that " < " not " <= " it based on your problem.......
}
```

check if a point inside the triangle

```
bool pointInsidTrian(point P1, point P2, point P3, point P) {
        long double A = fabs(cross(vec(P1, P2), vec(P1, P3)) / 2);
        long double A1 = fabs(cross(vec(P, P2), vec(P, P3)) / 2);
        long double A2 = fabs(cross(vec(P, P1), vec(P, P3)) / 2);
        long double A3 = fabs(cross(vec(P, P1), vec(P, P2)) / 2);
        //total area==areas of 3 triangles and the point is not in the porder =>>(A1>0 ,..)
        return fabs(A - A1 - A2 - A3) < EPS&&A1&&A2&&A3;
}
```

line using it's a, b, c

```
struct line {
        long double a, b, c;
        line(point p1, point p2) {
                if (abs(p1.X - p2.X) < EPS) { a = 1.0; b = 0.0; c = -p1.X; }
                else {
                        a = -(long double)(p1.Y - p2.Y) / (p1.X - p2.X);
                        b = 1.0;
                        c = -(long double)(a * p1.X) - p1.Y;
                }
        }
};
```

Point To Segment

```
point pointToSegment(point p0, point p1, point p2) {
        ld d1, d2;
        point v1 = p1 - p0, v2 = p2 - p0;
        if ((d1 = dp(v1, v2)) <= 0) return p0;
        if ((d2 = dp(v1, v1)) <= d1) return p1;
        ld t = d1 / d2;
        return (p0 + v1 * t);
}
```

check if to lines are parallel

```
bool areParallel(line l1, line l2) {
        return abs(l1.a - l2.a) < EPS && abs(l1.b - l2.b) < EPS;
}
```

check if to lines are the same

```
bool areSame(line l1, line l2) {
        return areParallel(l1, l2) && abs(l1.c - l2.c) < EPS;
}
```

check if to lines are the same using points

```
bool samelLine(const point &a1, const point &b1,const point &a2, const point &b2)
{
    return paralelVec(vec(a1,b1),vec(a2,b2))&&pointOnLine(a1,b1,a2);
}
```

check if 2 lines are intersecting

```
int linIntersect(point a, point b, point p, point q, point &ret) {
        line l1(a, b), l2(p, q);
        if (areParallel(l1, l2)) return areSame(l1, l2) ? 2 : 0;            // no intersection

                        // solve system of 2 linear algebraic equations with 2 unknowns
        double x, y;
        x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
```

```
	// special case: test for vertical line to avoid division by zero
	if (abs(l1.b) > EPS) y = -(l1.a * x + l1.c);
	else  y = -(l2.a * x + l2.c);
	ret = point(x, y);
	return 1;
}
```

get angle (AOB): in radian

```
double angl(point a, point o, point b) {
	point oa = vec(o, a), ob = vec(o, b);
	return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}
// wr answer in team formation :D
double getAngle_A_abc(double a, double b, double c) {
	return acos(fixAngle((b * b + c * c - a * a) / (2 * b * c)));
}
// sin(A)/a = sin(B)/b = sin(C)/c
// a^2 = b^2 + c^2 - 2b*c*cos(A)
```

Triangle side given other side and angle

```
double getSide_a_bAB(double b, double A, double B) {
	return (sin(A) * b) / sin(B);
}
```

Triangle angel given other side and angle

```
double getAngle_A_abB(double a, double b, double B) {
	return asin(fixAngle((a * sin(B)) / b));
}
```

Line Inside Rectangle

```
bool lineInsideRectangle(double x1, double x2, double y1, double y2, point st, point ed) {
	if (x2 < x1) swap(x1, x2);
	if (y2 < y1) swap(y1, y2);
	double mnX = min(st.X, ed.X), mxX = max(st.X, ed.X),
		mnY = min(st.Y, ed.Y), mxY = (st.Y, ed.Y);
	return dcmp(x1, mnX) <= 0 && dcmp(x2, mxX) >= 0 && dcmp(y1, mnY) <= 0 && dcmp(y2, mxY) >= 0;
}
```

check if 2 segments intersect

```
bool segmentIntersect(point a, point b, point p, point q, point &ret) {
	int ans = linIntersect(a, b, p, q, ret);
	if (ans == 0)return 0;
	// if we check on ray we must use pointOnray() insted of pointOnSegment
	if (ans == 1)return pointOnSegment(a, b, ret) && pointOnSegment(p, q, ret);
	if (a > b)swap(a, b);
	if (p > q)swap(p, q);
	if (b < p || q < a)return 0;
	ret = max(a, q); return 1;
}
```

check if a line intersects with a rectangle using its (left dawn) and (right up) points

```
bool linReqtIntersect(point LD, point RU, point a, point b) {
	point ret;
	point LU = point(LD.X, RU.Y);
	point RD = point(RU.X, LD.Y);
	if (pointInsidReqt(LD, RU, a))return 1;
	if (pointInsidReqt(LD, RU, b))return 1;
	if (segmentIntersect(LU, RU, a, b, ret))return 1;
	if (segmentIntersect(LD, RD, a, b, ret))return 1;
	if (segmentIntersect(LD, LU, a, b, ret))return 1;
```

```
	if (segmentIntersect(RD, RU, a, b, ret))return 1;
	return 0;
}
double norm_sq(point v) { return v.X * v.X + v.Y * v.Y; }
```

convert the angle from degree to radian

```
double toRad(double d) { return (d * PI) / 180.0; }
```

convert the angle from radian to degree

```
double toDeg(double d) {
	if (d < 0)  d += 2 * PI; return (d * 180 / PI);
}
```

find angle Anticlockwise from v1 to v2

```
double getAngle_2vec(point v1, point v2) {
	return toDeg(atan2(cross(v1, v2), dot(v1, v2)));
}
```

find angle p2p0p1, anti-clock p0p1 to p0p2

```
double getAngle_3points(point p0, point p1, point p2) {
	return getAngle_2vec(vec(p0, p1), vec(p0, p2));
}
```

get vector as a point from it length and angle

```
point getvec(double r, double ang) {
	//if the angle in degree
	ang = toRad(ang);
	double x = cos(ang)*r;
	double y = sin(ang)*r;
	return point(x, y);
}
point scale(point v, double r) {
	return point(v.X*r, v.Y*r);
}
```

get vector with length R according to a Ray ab

```
point getVectorWithLengthR(point a, point b, double r) {
	return scale(normalize(vec(a, b)), r); }
```

get triangle Area using 3 points

```
long double triangleArea3points(const point& a, const point& b, const point& c) {
	return fabs(cross(a, b) + cross(b, c) + cross(c, a)) / 2;
}
```

given the area of rectangle using 3 medians

```
double traingleArea_medians(double m1, double m2, double m3) {
	if (m1 <= 0.0 || m2 <= 0.0 || m3 <= 0.0)return -1; //impossipole
	double s = 0.5*(m1 + m2 + m3);
	double midian_area = (s*(s - m1)*(s - m2)*(s - m3));
	double area = 4.0 / 3.0*sqrt(midian_area);
	if (midian_area <= 0.0 || area <= 0.0)return -1;//impossipole
	return area;
}
```

check Overlapping Rectangles using (left down) and (right up) point for each

```
bool overlapReqt(point LD1, point RU1, point LD2, point RU2, point &LD3, point &RU3) {
	long double x, y;
	x = max(LD1.X, LD2.X);
	y = max(LD1.Y, LD2.Y);
	LD3 = point(x, y); // the left down poitn for the new rectangle
```

```
	x = min(RU1.X, RU2.X);
	y = min(RU1.Y, RU2.Y);
	RU3 = point(x, y);// the left right up for the new rectangle
	return LD3.X < RU3.X&&LD3.Y < RU3.Y;
}
```

Given are the (x, y) coordinates of the endpoints of two adjacent sides of a parallelogram. Find the (x, y) coordinates of the fourth point.

```
point getForthPoint(point a1, point a2, point b1, point b2) {
	if (same(a1, b1))return a2 + vec(b1, b2);
	if (same(a1, b2))return a2 + vec(b2, b1);
	if (same(a2, b1))return a1 + vec(b1, b2);
	return a1 + vec(b2, b1);
}
```

get all possible points that complete the parallelogram with another 3 points(triangle)

```
vector<point> getParalel_Tria(point p1, point p2, point p3) {
	vector<point> v(3);
	v[0] = p3 + vec(p2, p1);
	v[1] = p1 + vec(p3, p2);
	v[2] = p3 + vec(p1, p2);
	return v;//each point from them can added to the old 3 points to get the paralelogram
}
```

check if a polygon with angles = A is regular polygon

```
bool isRegulPoleg_Angle(double A) {
	for (int i = 3; i < 400; i++)
		if (A == ((i - 2.0)*180.0) / i)return 1;
	return 0;
}
```

Check if 4 point create a Rectangle

```
// the order needed colocwise
bool isRect(point a, point b, point c, point d){
	return vec(a, b) == vec(d, c) && vec(a, d) == vec(b, c) && prependecularVec(vec(a, b), vec(a, d));
}
```

Check if 4 point create a Square

```
// the order needed colocwise
bool isSquar(point a, point b, point c, point d){
	return vec(a, b) == vec(d, c) && vec(a, d) == vec(b, c)
		&& lengthSqr(vec(a, d)) == lengthSqr(vec(a, b)) && prependecularVec(vec(a, b), vec(a, d));
}
```

Point distance to Line

```
double pointDistToLine(point p, point a, point b, point &c) {//line a,b
	// formula: c = a + u * ab
	point ap = vec(a, p), ab = vec(a, b);
	long double u = dot(ap, ab) / lengthSqr(ab);
	c = translate(a, scale(ab, u)); // translate a to c
	return dist(p, c); }
```

Point distance to segment

```
// dis p to ab
double pointDistToLineSegment(point p, point a, point b, point &c) {
	point ap = vec(a, p), ab = vec(a, b);
	long double u = dot(ap, ab) / lengthSqr(ab);
	if (u < 0.0) {// closer to a
		c = point(a.X, a.Y); return dist(p, a);
	} // Euclidean distance between p and a
	if (u > 1.0) { // closer to b
```

```
		c = point(b.X, b.Y); return dist(p, b);
	} // Euclidean distance between p and b
	c = translate(a, scale(ab, u)); // translate a to c //between a,b
	return dist(p, c);
}
```

given 2 diagonal points for the square find the other two

```
void square_points(point p1, point p3, point& p2, point& p4) {
	// center point
	ld cx = (p1.X + p3.X) / 2.;
	ld cy = (p1.Y + p3.Y) / 2.;
	// half-diagonal
	ld dx = (p1.X - p3.X) / 2.;
	ld dy = (p1.Y - p3.Y) / 2.;
	ld x2 = cx - dy, y2 = cy + dx;    // second corner
	ld x4 = cx + dy, y4 = cy - dx;    // Fourth corner
	p2 = point(x2, y2);
	p4 = point(x4, y4);
	return;
}
```

Circle rad given 2 points and cen

```
bool circle2PtsRad(point p1, point p2, double r, point& c) {
	double d2 = (p1.X - p2.X) * (p1.X - p2.X) +
		(p1.Y - p2.Y) * (p1.Y - p2.Y);
	double det = r * r / d2 - 0.25;
	if (det < 0.0) return false;
	double h = sqrt(det);
	c.real((p1.X + p2.X) * 0.5 + (p1.Y - p2.Y) * h);
	c.imag((p1.Y + p2.Y) * 0.5 + (p2.X - p1.X) * h);
	return true;
	// to get the other center, reverse p1 and p2
}
```

circleCircleIntersectionArea

```
ld circleCircleIntersectionArea(point cen1, ld r1, point cen2, ld r2) {
	ld dis = hypot(cen1.X - cen2.X, cen1.Y - cen2.Y);
	if (dis > r1 + r2)return 0;
	if (dis <= fabs(r2 - r1) && r1 >= r2)
		return PI * r2 * r2;
	if (dis <= fabs(r2 - r1) && r1 < r2)
		return PI * r1 * r1;
	ld a = r1 * r1, b = r2 * r2;
	ld ang1 = acos((a + dis * dis - b) / (2 * r1 * dis)) * 2;
	ld ang2 = acos((b + dis * dis - a) / (2 * r2 * dis)) * 2;
	ld ret1 = .5 * b * (ang2 - sin(ang2));
	ld ret2 = .5 * a * (ang1 - sin(ang1));
	return ret1 + ret2;
}
```

distance between two points in arc

```
double calcArc(point p1, point p2, point cen) {
	double d = length(vec(p1, p2));
	double ang = (angle(vec(cen, p1)) - angle(vec(cen, p2))) * 180 / PI;
	if (ang < 0) ang += 360;
	ang = min(ang, 360 - ang);
	return r * ang * PI / 180;
}
```

Segment and Segment intersection

```
bool onSegment(point p, point q, point r) {
```

```
	if (q.X <= max(p.X, r.X) && q.X >= min(p.X, r.X) &&
		q.Y <= max(p.Y, r.Y) && q.Y >= min(p.Y, r.Y))
		return true;
	return false; }
ld orientation(point p, point q, point r) {
	ld val = (q.Y - p.Y) * (r.X - q.X) - (q.X - p.X) * (r.Y - q.Y);
	if (val == 0) return 0;
	return (val > 0) ? 1 : 2;
}
bool doIntersect(point p1, point q1, point p2, point q2) {
	ld o1 = orientation(p1, q1, p2);
	ld o2 = orientation(p1, q1, q2);
	ld o3 = orientation(p2, q2, p1);
	ld o4 = orientation(p2, q2, q1);
	if (fabs(o1 - o2) > EPS && fabs(o3 - o4) > EPS)
		return true;
	if (o1 == 0 && onSegment(p1, p2, q1)) return true;
	if (o2 == 0 && onSegment(p1, q2, q1)) return true;
	if (o3 == 0 && onSegment(p2, p1, q2)) return true;
	if (o4 == 0 && onSegment(p2, q1, q2)) return true;
	return false;
}
```

3D geometry

```
// give the sperical distance between 2 points using 2 angels for each (long, lat)
double spherical_distance(double lat1, double lon1, double lat2, double lon2, double rad) {
	double dlon = lon2 - lon1;
	double dlat = lat2 - lat1;
	double a = pow((sin(dlat / 2)), 2) + cos(lat1) * cos(lat2) * pow(sin(dlon / 2), 2);
	double c = 2 * atan2(sqrt(a), sqrt(1 - a));
	return rad * c;
}
```

3D Point

```
struct point3D{
	long double x, y, z;
	point3D(){};
	point3D(long double x1, long double y1, long double z1){
		x = x1; y = y1; z = z1;
	}
	bool read(){ if (cin >> x >> y >> z)return 1; return 0; }
	long double dis(point3D other){
		return sqrt(pow(x - other.x, 2) + pow(y - other.y, 2) + pow(z - other.z, 2));
	}
};
```

The volume of Triangular Pyramid

```
long double cosRule(long double a, long double b, long double c){
	// Handle denom = 0
	long double res = (b * b + a * a - c*c) / (2 * b *a);
	if (res > 1)
		res = 1;
	if (res < -1)
		res = -1;
	return res;
}
ld triangularPyramidVolume(ld AB, ld AC, ld BC, ld AD, ld BD, ld CD){
	ld cos1 = cosRule(AD, BD, AB),cos2 = cosRule(BD, CD, BC),cos3 = cosRule(CD, AD, AC);
	ld  temp = sqrt(1 + 2 * cos1 * cos2 * cos3 - cos1*cos1 - cos2*cos2 - cos3*cos3);
	return  AD * BD * CD * temp / 6;
}
```

```
long double pyramidVolume(long double ab, long double ac, long double ad,
        long double bc, long double bd, long double cd) {
        long double w = ab, v = ac, u = ad, U = bc, V = bd, W = cd;
        long double X = (w - U + v) * (U + v + w);
        long double x = (U - v + w) * (v - w + U);
        long double Y = (u - V + w) * (V + w + u);
        long double y = (V - w + u) * (w - u + V);
        long double Z = (v - W + u) * (W + u + v);
        long double z = (W - u + v) * (u - v + W);
        long double a = sqrt(x * Y * Z);
        long double b = sqrt(X * y * Z);
        long double c = sqrt(X * Y * z);
        long double d = sqrt(x * y * z);
        long double volume = -a + b + c + d;
        volume *= a - b + c + d;
        volume *= a + b - c + d;
        volume *= a + b + c - d;
        volume = sqrt(volume) / (192.0 * u * v * w);
        return volume;
}
```

Data structure

Fenwick tree

```
template<typename T>
struct fenwick_tree {
	/* can convert it to map
	 * build what you need only
	 * will be: memory O(q*logn) ,time O(logn*logn) */
	vector<T> BIT;
	int n;
	fenwick_tree(int n) :
		n(n), BIT(n + 1) {
	}
	T getAccum(int idx) {
		T sum = 0;
		while (idx) {
			sum += BIT[idx];
			idx -= (idx & -idx);
		}
		return sum;
	}
	void add(int idx, T val) {
		assert(idx != 0);
		while (idx <= n) {
			BIT[idx] += val;
			idx += (idx & -idx);
		}
	}
	T getValue(int idx) {
		return getAccum(idx) - getAccum(idx - 1); }
	// ordered statistics tree
	// get index that has value >= accum
	// values must by positive
	int getIdx(T accum) {
		int start = 1, end = n, rt = -1;
		while (start <= end) {
			int mid = start + end >> 1;
			T val = getAccum(mid);
			if (val >= accum)
				rt = mid, end = mid - 1;
			else
				start = mid + 1;
		}
		return rt;
	}
	//need review (from topcoder)
	//first index less than or equal accum  O(logn)
	int find(T accum) {  //equal getIdx
		int i = 1, idx = 0;
		while ((1 << i) <= MAX)
			i <<= 1;
		for (idx = 0; i > 0; i >>= 1) {
			int tidx = idx + i;
			if (tidx > MAX)
				continue;
			if (accum >= BIT[tidx]) {
				idx = tidx;
				accum -= BIT[tidx];
			}
		}
		return idx;//idx+1 if you need first greater
	}
};
```

Fenwick tree 2d

```
template<typename T>
struct fenwick_tree_2d {
#define Lbit(x) (x&-x)
	int n, m;
	vector<vector<T>> BIT;
	fenwick_tree_2d(int n, int m) :
		n(n), m(m), BIT(n + 1, vector<T>(m + 1)) {
	}
	T getAccum(int i, int j) {
		T sum = 0;
		for (; i; i -= Lbit(i))
			for (int idx = j; idx > 0; idx -= Lbit(idx))
				sum += BIT[i][idx];
		return sum;
	}
	void add(int i, int j, int val) {
		assert(i != 0 && j != 0);
		for (; i <= n; i += Lbit(i))
			for (int idx = j; idx <= m; idx += Lbit(idx))
				BIT[i][idx] += val;
	}
	T getRectangeSum(int x1, int y1, int x2, int y2) {
		if (y1 > y2)
			swap(y1, y2);
		if (x1 > x2)
			swap(x1, x2);
		return getAccum(x2, y2) - getAccum(x1 - 1, y2) - getAccum(x2, y1 - 1)
			+ getAccum(x1 - 1, y1 - 1);
	}
};
```

Fenwick tree max

```
//fenwick tree for get maximum from 1 to idx
// update a[idx] = max(a[idx],val)
//can't remove values
template<typename T>
struct fenwick_tree {
	vector<T> BIT;
	int n;
	fenwick_tree(int n) :
		n(n), BIT(n + 1) {
	}
	T getMax(int idx) {
		T mx = numeric_limits<T>::min();
		while (idx) {
			mx = max(mx, BIT[idx]);
			idx -= (idx & -idx);
		}
		return mx;
	}
	void add(int idx, T val) {
		assert(idx != 0);
		while (idx <= n) {
			BIT[idx] = max(BIT[idx], val);
			idx += (idx & -idx);
		}
	}
};
```

Fenwick tree update range

```
/*
 x[i] = a[i] - a[i-1] //a is original array
 y[i] = x[i]*(i-1)

 sum(1,3) = a[1] + a[2] + a[3] = (x[1]) + (x[2] + x[1]) + (x[3] + x[2] + x[1])

 = 3*(x[1] + x[2] + x[3]) - 0*x[1] - 1*x[2] - 2*x[3] //same equation but more complex
 = sumX(1,3) * 3 - sumY(1,3)
 so sum(1,n) = sumX(1,n)*n - sumY(1,n)

 update:
 x[l] += val,x[r+1] -= val
 y[l] += val *(l-1),y[r+1] -= r*val
 */

template<typename T>
class fenwick_tree {
	int n;
	vector<T> x, y;
	T getAccum(vector<T>& BIT, int idx) {
		T sum = 0;
		while (idx) {
			sum += BIT[idx];
			idx -= (idx & -idx);
		}
		return sum;
	}
	void add(vector<T>& BIT, int idx, T val) {
		assert(idx != 0);
		while (idx <= n) {
			BIT[idx] += val;
			idx += (idx & -idx);
		}
	}
	T prefix_sum(int idx) {
		return getAccum(x, idx) * idx - getAccum(y, idx);
	}
public:
	fenwick_tree(int n) :
		n(n), x(n + 1), y(n + 1) {
	}
	void update_range(int l, int r, T val) {
		add(x, l, val);
		add(x, r + 1, -val);
		add(y, l, val * (l - 1));
		add(y, r + 1, -val * r);
	}
	T range_sum(int l, int r) {
		return prefix_sum(r) - prefix_sum(l - 1);
	}
};
```

Segment tree

```
/*
 for efficient memory (2*n)
 #define LEFT (idx+1)
 #define MID ((start+end)>>1)
 #define RIGHT (idx+((MID-start+1)<<1))
 */
template<typename node> class segment_tree {
#define LEFT (idx<<1)
#define RIGHT (idx<<1|1)
#define MID ((start+end)>>1)
	int left_range, right_range;
	vector<node> tree;
	inline void pushup(int idx) {
		tree[idx] = node(tree[LEFT], tree[RIGHT]);
	}
	inline void pushdown(int idx, int start, int end) {
		if (!tree[idx].have_lazy || start == end)
			return;
		tree[LEFT].apply(start, MID, tree[idx].lazy_value);
		tree[RIGHT].apply(MID + 1, end, tree[idx].lazy_value);
		tree[idx].clear_lazy();
	}
	void build(int idx, int start, int end) {
		if (start == end)
			return;
		build(LEFT, start, MID);
		build(RIGHT, MID + 1, end);
		pushup(idx);
	}
	template<typename T>
	void build(int idx, int start, int end, const vector<T>& arr) {
		if (start == end) {
			tree[idx] = arr[start];
			return;
		}
		build(LEFT, start, MID, arr);
		build(RIGHT, MID + 1, end, arr);
		pushup(idx);
	}
	node query(int idx, int start, int end, int from, int to) {
		if (from <= start && end <= to)
			return tree[idx];
		pushdown(idx, start, end);
		if (to <= MID)
			return query(LEFT, start, MID, from, to);
		if (MID < from)
			return query(RIGHT, MID + 1, end, from, to);
		return node(query(LEFT, start, MID, from, to),
			query(RIGHT, MID + 1, end, from, to));
	}
	template<typename ... T>
	void update(int idx, int start, int end, int from, int to,const T&... val) {
		if (to < start || end < from)
			return;
		if (from <= start && end <= to) {
			tree[idx].apply(start, end, val...);
			return;
		}
		pushdown(idx, start, end);
		update(LEFT, start, MID, from, to, val...);
		update(RIGHT, MID + 1, end, from, to, val...);
		pushup(idx);
	}
```

```cpp
	void init(int l, int r) {
		left_range = l;
		right_range = r;
		tree = vector < node >((r - l + 1) << 2);
	}
public:
	segment_tree(int l, int r) {
		init(l, r);
		build(1, l, r);
	}
	template<typename T>
	segment_tree(int l, int r, const vector<T>& v) {
		init(l, r);
		build(1, l, r, v);
	}
	node query(int l, int r) {
		assert(left_range <= l && l <= r && r <= right_range);
		return query(1, left_range, right_range, l, r);
	}
	template<typename ... T>
	void update(int l, int r, const T&... val) {
		assert(left_range <= l && l <= r && r <= right_range);
		update(1, left_range, right_range, l, r, val...);
	}
#undef LEFT
#undef RIGHT
#undef MID
};
```

Segment tree without lazy

```cpp
struct node {
	node() { //set Default value
	}
	node(const node& a, const node& b) {}
	void apply(int val) {}
};
struct segment_tree {
	int n; //0 to n-1
	vector<node> tree;
	segment_tree(int n) {
		resize(n);
		build();
	}
	template<typename T>
	segment_tree(const vector<T>& arr) {
		resize(arr.size());
		for (int i = 0; i < arr.size(); i++)
			tree[n + i] = arr[i];
		build();
	}
	void resize(int n) {
		int p = 1;
		while (p < n)
			p <<= 1;
		this->n = p;
		tree = vector < node >(p << 1);
	}
	void build() {
		for (int i = n - 1; i > 0; i--)
			tree[i] = node(tree[i << 1], tree[i << 1 | 1]);
	}
```

```cpp
	template<typename T>
	void update(int p, const T& value) {
		tree[p += n].apply(value);
		for (int i = p / 2; i > 0; i >>= 1)
			tree[i] = node(tree[i << 1], tree[i << 1 | 1]);
	}
	node query(int l, int r) {  //[l, r]
		node resl, resr;  //set default value in node
		for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
			if (l & 1) {
				resl = node(resl, tree[l]);
				l++;
			}
			if (r & 1) {
				r--;
				resr = node(tree[r], resr);
			}
		}
		return node(resl, resr);
	}
	int kth_one(int k, int i = 1) {
		if (k > tree[i])
			return -1;
		if (i >= n)
			return i - n;
		if (tree[i << 1] >= k)
			return kth_one(k, i << 1);
		return kth_one(k - tree[i << 1], i << 1 | 1);
	}
};
```

Max sum range node

```cpp
struct MSR_Node {
	ll left, right, mid, sum;
	MSR_Node(const ll& val) {
		left = right = mid = sum = val;
	}
	MSR_Node(const MSR_Node& a, const MSR_Node& b) {
		left = max(a.left, a.sum + b.left);
		right = max(b.right, b.sum + a.right);
		mid = max({ a.mid, b.mid, a.right + b.left });
		sum = a.sum + b.sum;
	}
	ll getMax() {
		return max({ left, right, mid, sum });
	}
};
```

Ordered set

```cpp
#include<bits/stdc++.h>
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
template<typename key>
using ordered_set = tree<key, null_type, less<key>, rb_tree_tag, tree_order_statistics_node_update>;
/*
 find_by_order(k) :
 It returns to an iterator to the k-th element (counting from zero) in the set in O(logn) time.
 To find the first element k must be zero.
 order_of_key(k) :
 It returns to the number of items that are strictly smaller than our item k in O(logn) time.
 */
```

Sparse table

```
template<typename T>
struct sparse_table {
	vector<vector<T>> sparseTable;
	using F = function<T(T, T)>;
	F merge;
	static int LOG2(int x) { //floor(log2(x))
		return 31 - __builtin_clz(x);
	}
	sparse_table(vector<T>& v, F merge) :
		merge(merge) {
		int n = v.size();
		int logN = LOG2(n);
		sparseTable = vector < vector < T >>(logN + 1);
		sparseTable[0] = v;
		for (int k = 1, len = 1; k <= logN; k++, len <<= 1) {
			sparseTable[k].resize(n);
			for (int i = 0; i + len < n; i++)
				sparseTable[k][i] = merge(sparseTable[k - 1][i],
					sparseTable[k - 1][i + len]);
		}
	}
	T query(int l, int r) {
		int k = LOG2(r - l + 1); // max k ==> 2^k <= length of range
		//check first 2^k from left and last 2^k from right //overlap
		return merge(sparseTable[k][l], sparseTable[k][r - (1 << k) + 1]);
	}
	T query_shifting(int l, int r) {
		T res;
		bool first = true;
		for (int i = (int)sparseTable.size() - 1; i >= 0; i--)
			if (l + (1 << i) - 1 <= r) {
				if (first)
					res = sparseTable[i][l];
				else
					res = merge(res, sparseTable[i][l]);
				first = false;
				l += (1 << i);
			}
		return res;
	}
};
```

SQRT Decomposition

```
//zero based SQRT_Decomposition with lazy propagation
template<typename update_type, typename query_type>
class SQRT_Decomposition {
	struct Bucket {
		int l, r;
		update_type lazy;
		Bucket(int l, int r) :
			l(l), r(r) {
			//set default value to lazy
			//build bucket for the first time
		}
		void build() {
			//update all bucket with lazy if have
			//rebuild the bucket
			//clear lazy
		}
		//update all bucket
		void update(const update_type& val) {
			//just update lazy
```

```
		}
		//update range in bucket
		void update(int start, int end, const update_type& val) {
			if (start == l && end == r) {
				update(val);
				return;
			}
			//update bucket
			//rebuild the bucket if need
		}
		//query about all bucket
		query_type query() {
			//calc with lazy
		}
		//query about range in bucket
		query_type query(int start, int end) {
			if (start == l && end == r)
				return query();
			//push lazy if have
			//calc
		}
	};

	int n, sqrtN;
	vector<Bucket> bucket;
	int begin(int idx) {
		return idx * sqrtN;
	}
	int end(int idx) {
		return min(sqrtN * (idx + 1), n) - 1;
	}
	int which_block(int idx) {
		return idx / sqrtN;
	}
public:
	SQRT_Decomposition(int n) {
		this->n = n;
		sqrtN = sqrt(n);
		for (int i = 0; i < n; i += sqrtN)
			bucket.push_back(Bucket(i, min(i + sqrtN, n) - 1));
	}

	void update(int left, int right, update_type val) {
		int st = which_block(left), ed = which_block(right);
		bucket[st].update(left, min(bucket[st].r, right), val);
		for (int i = st + 1; i < ed; i++)
			bucket[i].update(val);
		if (st != ed)
			bucket[ed].update(bucket[ed].l, right, val);
	}

	query_type query(int left, int right) {
		int st = which_block(left), ed = which_block(right);
		query_type rt = bucket[st].query(left, min(bucket[st].r, right));
		for (int i = st + 1; i < ed; i++)
			rt += bucket[i].query();
		if (st != ed)
			rt += bucket[ed].query(bucket[ed].l, right);
		return rt;
	}
};
```

Implicit treap

```
#if __cplusplus >= 201402L
template<typename T>
vector<T> create(size_t n) {
	return vector<T>(n);
}
template<typename T, typename ... Args>
auto create(size_t n, Args ... args) {
	return vector<decltype(create<T>(args...))>(n, create<T>(args...));
}
#endif
enum DIR {
	L, R
};

template<typename T>
struct cartesian_tree {
	static cartesian_tree<T>* sentinel;
	T key;
	int priority = 0, size = 0;
	bool reverse = false;
	cartesian_tree* child[2];
	cartesian_tree() {
		key = T();
		priority = 0;
		child[L] = child[R] = this;
	}
	cartesian_tree(const T& x, int y) :
		key(x), priority(y) {
		size = 1;
		child[L] = child[R] = sentinel;
	}
	void push_down() {
		if (!reverse)
			return;
		reverse = 0;
		child[L]->doRevese();
		child[R]->doRevese();
	}
	void doReverse() {
		reverse ^= 1;
		swap(child[L], child[R]);
	}
	void push_up() {
		size = child[L]->size + child[R]->size + 1;
	}

};

template<typename T>
cartesian_tree<T>* cartesian_tree<T>::sentinel = new cartesian_tree<T>();

template<typename T, template<typename > class cartesian_tree>
class implicit_treap { //1 based
	typedef cartesian_tree<T> node;
	typedef cartesian_tree<T>* nodeptr;
#define emptyNode cartesian_tree<T>::sentinel
	nodeptr root;
	void split(nodeptr root, nodeptr& l, nodeptr& r, int firstXElment) {
		if (root == emptyNode) {
			l = r = emptyNode;
			return;
		}
		root->push_down();
```

```
		if (firstXElment <= root->child[L]->size) {
			split(root->child[L], l, root->child[L], firstXElment);
			r = root;
		}
		else {
			split(root->child[R], root->child[R], r,
				firstXElment - root->child[L]->size - 1);
			l = root;
		}
		root->push_up();
	}

	nodeptr merge(nodeptr l, nodeptr r) {
		l->push_down();
		r->push_down();
		if (l == emptyNode || r == emptyNode)
			return (l == emptyNode ? r : l);
		if (l->priority > r->priority) {
			l->child[R] = merge(l->child[R], r);
			l->push_up();
			return l;
		}
		r->child[L] = merge(l, r->child[L]);
		r->push_up();
		return r;
	}
	vector<nodeptr> split_range(int s, int e) { // [x<s,s<=x<=e,e<x]
		nodeptr l, m, r, tmp;
		split(root, l, tmp, s - 1);
		split(tmp, m, r, e - s + 1);
		return { l,m,r };
	}
public:
	implicit_treap() :
		root(emptyNode) {
	}
	int size() {
		return root->size;
	}
	void insert(int pos, const T& key) {
		nodeptr tmp = new node(key, rand());
		nodeptr l, r;
		split(root, l, r, pos - 1);
		root = merge(merge(l, tmp), r);
	}
	void push_back(const T& value) {
		root = merge(root, new node(value, rand()));
	}
	T getByIndex(int pos) {
		vector<nodeptr> tmp = split_range(pos, pos);
		nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
		T rt = m->key;
		root = merge(merge(l, m), r);
		return rt;
	}
	void erase(int pos) {
		vector<nodeptr> tmp = split_range(pos, pos);
		nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
		delete m;
		root = merge(l, r);
	}
	void cyclic_shift(int s, int e) { //to the right
		vector<nodeptr> tmp = split_range(s, e);
		nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
```

```
		nodeptr first, second;
		split(m, first, second, e - s);
		root = merge(merge(merge(l, second), first), r);
	}
	void reverse_range(int s, int e) {
		vector<nodeptr> tmp = split_range(s, e);
		nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
		m->reverse ^= 1;
		root = merge(merge(l, m), r);
	}
	node range_query(int s, int e) {
		vector<nodeptr> tmp = split_range(s, e);
		nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
		node rt = *m;
		root = merge(merge(l, m), r);
		return rt;
	}
};
```

Ordered multiset

```
enum DIR {
	L, R
};

template<typename T>
struct cartesian_tree {
	static cartesian_tree<T>* sentinel;
	T key;
	int priority = 0, frequency = 0, size = 0;
	cartesian_tree* child[2];
	cartesian_tree() {
		key = T();
		priority = 0;
		child[L] = child[R] = this;
	}
	cartesian_tree(const T& x, int y) :
		key(x), priority(y) {
		size = frequency = 1;
		child[L] = child[R] = sentinel;
	}
	void push_down() {

	}
	void push_up() {
		size = child[L]->size + child[R]->size + frequency;
	}
};

template<typename T>
cartesian_tree<T>* cartesian_tree<T>::sentinel = new cartesian_tree<T>();

template<typename T>
void split(cartesian_tree<T>* root, T key, cartesian_tree<T>*& l,
	cartesian_tree<T>*& r) {
	if (root == cartesian_tree<T>::sentinel) {
		l = r = cartesian_tree<T>::sentinel;
		return;
	}
	root->push_down();
	if (root->key <= key) {
		split(root->child[R], key, root->child[R], r);
		l = root;
	}
	else {
```

```
		split(root->child[L], key, l, root->child[L]);
		r = root;
	}
	root->push_up();
}

template<typename T>
cartesian_tree<T>* merge(cartesian_tree<T>* l, cartesian_tree<T>* r) {
	l->push_down();
	r->push_down();
	if (l == cartesian_tree<T>::sentinel || r == cartesian_tree<T>::sentinel)
		return (l == cartesian_tree<T>::sentinel ? r : l);
	if (l->priority > r->priority) {
		l->child[R] = merge(l->child[R], r);
		l->push_up();
		return l;
	}
	r->child[L] = merge(l, r->child[L]);
	r->push_up();
	return r;
}

template<typename T, template<typename > class cartesian_tree>
class treap {
	typedef cartesian_tree<T> node;
	typedef node* nodeptr;
#define emptyNode node::sentinel
	nodeptr root;
	void insert(nodeptr& root, nodeptr it) {
		if (root == emptyNode) {
			root = it;
		}
		else if (it->priority > root->priority) {
			split(root, it->key, it->child[L], it->child[R]);
			root = it;
		}
		else
			insert(root->child[root->key < it->key], it);
		root->push_up();
	}
	bool increment(nodeptr root, const T& key) {
		if (root == emptyNode)
			return 0;
		if (root->key == key) {
			root->frequency++;
			root->push_up();
			return root;
		}
		bool rt = increment(root->child[root->key < key], key);
		root->push_up();
		return rt;
	}
	nodeptr find(nodeptr root, const T& key) {
		if (root == emptyNode || root->key == key)
			return root;
		return find(root->child[root->key < key], key);
	}
	void erase(nodeptr& root, const T& key) {
		if (root == emptyNode)
			return;
		if (root->key == key) {
			if (--(root->frequency) == 0)
				root = merge(root->child[L], root->child[R]);
		}
```

```
		else
			erase(root->child[root->key < key], key);
		root->push_up();
	}
	T kth(nodeptr root, int k) {
		if (root->child[L]->size >= k)
			return kth(root->child[L], k);
		k -= root->child[L]->size;
		if (k <= root->frequency)
			return root->key;
		return kth(root->child[R], k - root->frequency);
	}
	int order_of_key(nodeptr root, const T& key) {
		if (root == emptyNode)
			return 0;
		if (key < root->key)
			return order_of_key(root->child[L], key);
		if (key == root->key)
			return root->child[L]->size;
		return root->child[L]->size + root->frequency
			+ order_of_key(root->child[R], key);
	}
public:
	treap() :
		root(emptyNode) {
	}
	void insert(const T& x) {
		if (increment(root, x)) //change it to find(x) to make it as a set
			return;
		insert(root, new node(x, rand()));
	}
	void erase(const T& x) {
		erase(root, x);
	}
	bool find(const T& x) {
		return (find(root, x) != emptyNode);
	}
	int get_kth_number(int k) {
		assert(1 <= k && k <= size());
		return kth(root, k);
	}
	int order_of_key(const T& x) {
		return order_of_key(root, x);
	}
	int size() {
		return root->size;
	}
};
```

Heavy light decomposition

```
class heavy_light_decomposition { //1-based,if value in node,just update it after build chains
	int n, is_value_in_edge;
	vector<int> parent, depth, heavy, root, pos_in_array, pos_to_node, size;
	const static int merge(int a, int b); //implement it
	struct array_ds { //implement it
		int n;
		array_ds(int n) :
			n(n) {
		}
		void update(int pos, int value);
		int query(int l, int r);
	} seg;
	struct TREE {
```

```
		int cnt_edges = 1;
		vector<vector<int>> adj;
		//need for value in edges
		vector<vector<int>> edge_idx;
		//edge_to need for undirected tree //end of edge in directed tree
		vector<int> edge_to, edge_cost;
		TREE(int n) :
			adj(n + 1), edge_idx(n + 1), edge_to(n + 1), edge_cost(n + 1) {
		}
		void add_edge(int u, int v, int c) {
			adj[u].push_back(v);
			adj[v].push_back(u);
			edge_idx[u].push_back(cnt_edges);
			edge_idx[v].push_back(cnt_edges);
			edge_cost[cnt_edges] = c;
			cnt_edges++;
		}
	} tree;
	int dfs_hld(int node) {
		int size = 1, max_sub_tree = 0;
		for (int i = 0; i < (int)tree.adj[node].size(); i++) {
			int ch = tree.adj[node][i], edge_idx = tree.edge_idx[node][i];
			if (ch != parent[node]) {
				tree.edge_to[edge_idx] = ch;
				parent[ch] = node;
				depth[ch] = depth[node] + 1;
				int child_size = dfs_hld(ch);
				if (child_size > max_sub_tree)
					heavy[node] = ch, max_sub_tree = child_size;
				size += child_size;
			}
		}
		return size;
	}
public:
	heavy_light_decomposition(int n, bool is_value_in_edge) :
		n(n), is_value_in_edge(is_value_in_edge), seg(n + 1), tree(n + 1) {
		heavy = vector<int>(n + 1, -1);
		parent = depth = root = pos_in_array = pos_to_node = size = vector<int>(
			n + 1);
	}
	void add_edge(int u, int v, int c = 0) {
		tree.add_edge(u, v, c);
	}
	void build_chains(int src = 1) {
		parent[src] = -1;
		dfs_hld(src);
		for (int chain_root = 1, pos = 1; chain_root <= n; chain_root++) {
			if (parent[chain_root] == -1
				|| heavy[parent[chain_root]] != chain_root)
				for (int j = chain_root; j != -1; j = heavy[j]) {
					root[j] = chain_root;
					pos_in_array[j] = pos++;
					pos_to_node[pos_in_array[j]] = j;
				}
		}
		if (is_value_in_edge)
			for (int i = 1; i < n; i++)
				update_edge(i, tree.edge_cost[i]);
	}
	void update_node(int node, int value) { // O(update in seg)
		seg.update(pos_in_array[node], value);
	}
	void update_edge(int edge_idx, int value) { // O(update in seg)
```

```
			update_node(tree.edge_to[edge_idx], value);
	}
	void update_path(int u, ll val) {//update from node to root
		while (u >= 1) {
			seg.update(pos_in_array[root[u]], pos_in_array[u], val);
			u = parent[root[u]];
		}
	}
	int query_in_path(int u, int v) { //O(logn * (query in seg))
		vector<pair<int, int>> tmp[2];
		bool idx = 1;
		while (root[u] != root[v]) {
			if (depth[root[u]] > depth[root[v]]) {
				swap(u, v);
				idx = !idx;
			}
			//if value in edges ,you need value of root[v] also (connecter edge)
			tmp[idx].push_back({ pos_in_array[root[v]], pos_in_array[v] });
			v = parent[root[v]];
		}
		if (depth[u] > depth[v]) {
			swap(u, v);
			idx = !idx;
		}
		if (!is_value_in_edge || u != v)
			tmp[idx].push_back({ pos_in_array[u] + is_value_in_edge,
					pos_in_array[v] });

		//initial value,check handling if u == v
		int query_res = 0;
		for (auto& it : tmp[0])
			query_res = merge(query_res, seg.query(it.first, it.second));
		for (int i = tmp[1].size() - 1; i >= 0; i--)
			query_res = merge(query_res,
				seg.query(tmp[1][i].first, tmp[1][i].second));
		return query_res; //u is LCA
	}
};
```

LCA

```
class LCA {
	int n, logN, root = 1;
	vector<int> depth;
	vector<vector<int>> adj, lca;
	void dfs(int node, int parent) {
		lca[node][0] = parent;
		depth[node] = (~parent ? depth[parent] + 1 : 0);
		for (int k = 1; k <= logN; k++) {
			int up_parent = lca[node][k - 1];
			if (~up_parent)
				lca[node][k] = lca[up_parent][k - 1];
		}
		for (int child : adj[node])
			if (child != parent)
				dfs(child, node);
	}
public:
	LCA(const vector<vector<int>>& _adj, int root = 1) :
		root(root), adj(_adj) {
		adj = _adj;
		n = adj.size() - 1;
		logN = log2(n);
		lca = vector<vector<int>>(n + 1, vector<int>(logN + 1, -1));
```

```
		depth = vector<int>(n + 1);
		dfs(root, -1);
	}
	// return first = LCA,second = distance between the two nodes
	pair<int, int> get_LCA(int u, int v) {
		if (depth[u] < depth[v])
			swap(u, v);
		int dis = 0;
		for (int k = logN; k >= 0; k--)
			if (depth[u] - (1 << k) >= depth[v])
				u = lca[u][k], dis += (1 << k);
		if (u == v)
			return { u,dis };
		for (int k = logN; k >= 0; k--) {
			if (lca[u][k] != lca[v][k]) {
				u = lca[u][k];
				v = lca[v][k];
				dis += (1 << k + 1);
			}
		}
		return { lca[u][0],dis + 2 };
	}
	int shifting(int node, int dist) {
		for (int i = logN; i >= 0 && ~node; i--)
			if (dist & (1 << i))
				node = lca[node][i];
		return node;
	}
};
```

Centroid decomposition

```
class centroid_decomposition {
	vector<bool> centroidMarked;
	vector<int> size;
	void dfsSize(int node, int par) {
		size[node] = 1;
		for (int ch : adj[node])
			if (ch != par && !centroidMarked[ch]) {
				dfsSize(ch, node);
				size[node] += size[ch];
			}
	}
	int getCenter(int node, int par, int size_of_tree) {
		bool is_centroid = true;
		int heaviest_child = -1;
		for (int ch : adj[node])
			if (ch != par && !centroidMarked[ch]) {
				if (size[ch] > size_of_tree / 2)
					is_centroid = false;
				if (heaviest_child == -1 || size[ch] > size[heaviest_child])
					heaviest_child = ch;
			}
		if (is_centroid && size_of_tree - size[node] <= size_of_tree / 2)
			return node;
		assert(heaviest_child != -1);
		return getCenter(heaviest_child, node, size_of_tree);
	}
	int getCentroid(int src) {
		dfsSize(src, -1);
		int centroid = getCenter(src, -1, size[src]);
		centroidMarked[centroid] = true; //need to mark it after solve?
		return centroid;
	}
```

```
	int decomposeTree(int root) {
		root = getCentroid(root);
		solve(root);
		for (int ch : adj[root]) {
			if (centroidMarked[ch])
				continue;
			int centroid_of_subtree = decomposeTree(ch);
			//note: root and centroid_of_subtree probably not have a direct edge in adj
			centroidTree[root].push_back(centroid_of_subtree);
			//centroidTree[centroid_of_subtree].push_back(root);
		}
		return root;
	}
	void calc(int node, int par) {
		//TO-DO
		for (int ch : adj[node])
			if (ch != par && !centroidMarked[ch])
				calc(ch, node);
	}
	void add(int node, int par) {
		//TO-DO
		for (int ch : adj[node])
			if (ch != par && !centroidMarked[ch])
				add(ch, node);
	}
	void remove(int node, int par) {
		//TO-DO
		for (int ch : adj[node])
			if (ch != par && !centroidMarked[ch])
				remove(ch, node);
	}
	void solve(int root) {
		//add root
		for (int ch : adj[root])
			if (!centroidMarked[ch]) {
				calc(ch, root);
				add(ch, root);
			}
		//remove root
		for (int ch : adj[root])
			if (!centroidMarked[ch])
				remove(ch, root);
	}
public:
	vector<vector<int>> adj, centroidTree;
	int n, root;
	centroid_decomposition(vector<vector<int>>& adj) :
		adj(adj) {
		n = (int)adj.size() - 1;
		size = vector<int>(n + 1);
		centroidTree = vector<vector<int>>(n + 1);
		centroidMarked = vector<bool>(n + 1);
		root = decomposeTree(1);
	}
};
```

DSU

```
struct DSU {
	vector<int> rank, parent, size;
	vector<vector<int>> component;
	int forsets;
	DSU(int n) {
		size = rank = parent = vector<int>(n + 1, 1);
```

```
		component = vector<vector<int>>(n + 1);
		forsets = n;
		for (int i = 0; i <= n; i++) {
			parent[i] = i;
			component[i].push_back(i);
		}
	}
	int find_set(int v) {
		if (v == parent[v])
			return v;
		return parent[v] = find_set(parent[v]);
	}
	void link(int par, int node) {
		parent[node] = par;
		size[par] += size[node];
		for (const int& it : component[node])
			component[par].push_back(it);
		component[node].clear();
		if (rank[par] == rank[node])
			rank[par]++;
		forsets--;
	}
	bool union_sets(int v, int u) {
		v = find_set(v), u = find_set(u);
		if (v != u) {
			if (rank[v] < rank[u])
				swap(v, u);
			link(v, u);
		}
		return v != u;
	}
	bool same_set(int v, int u) {
		return find_set(v) == find_set(u);
	}
	int size_set(int v) {
		return size[find_set(v)];
	}
};
```

DSU apps

```
void Painting_subarrays() {
	struct Query {
		int l, r, c;
		Query(int l, int r, int c) :
			l(l), r(r), c(c) {
		}
	};
	int n, q;
	cin >> n >> q;
	DSU uf(n);
	vector<int> ans(n + 1);
	vector<Query> query(q);
	for (int i = 0; i < q; i++)
		cin >> query[i].l >> query[i].r >> query[i].c;
	reverse(query.begin(), query.end());
	for (auto q : query) {
		int l = q.l, r = q.r, c = q.c;
		for (int cur = uf.find_set(l); cur <= r; cur = uf.find_set(cur)) {
			uf.parent[cur] = cur + 1;
			ans[cur] = c;
		}
	}
}
```

```
void RMQ() {
	struct Query {
		int l, r, idx;
		Query(int l, int r, int idx) :
			l(l), r(r), idx(idx) {
		}
	};
	int n, q;
	cin >> n >> q;
	vector<int> v(n);
	vector < vector < Query >> query(n);
	vector<int> ans(q);
	DSU uf(n);
	for (auto& a : v)
		cin >> a;
	for (int i = 0; i < q; i++) {
		int l, r;
		cin >> l >> r;
		query[r].push_back(Query(l, r, i));
	}
	stack<int> st;
	for (int i = 0; i < n; i++) {
		while (!st.empty() && v[st.top()] > v[i]) {
			uf.parent[st.top()] = i;
			st.pop();
		}
		st.push(i);
		for (auto q : query[i])
			ans[q.idx] = v[uf.find_set(q.l)];
	}
}
```

BST

```
class BST {
	struct node {
		int key;
		node* left, * right, * parent;
		node() {
			key = 0;
			left = right = parent = NULL;
		}
		node(int key, node* left = NULL, node* right = NULL, node* parent = NULL) :
			key(key), left(left), right(right), parent(parent) {
		}
	};

	typedef node* nodeptr;
	nodeptr minimum(nodeptr root) {
		if (root->left == NULL)
			return root;
		return minimum(root->left);
	}
	nodeptr maximum(nodeptr root) {
		if (root->right == NULL)
			return root;
		return maximum(root->right);
	}
	nodeptr successor(nodeptr cur) { //smallest key larger than cur
		if (cur->right != NULL)
			return minimum(cur->right);
		nodeptr tmp = cur->parent;
		while (tmp != NULL && tmp->right == cur)
```

```
			cur = tmp, tmp = tmp->parent;
		return tmp;
	}
	nodeptr bredecessor(nodeptr cur) { //biggest key less than cur
		if (cur->left != NULL)
			return maximum(cur->left);
		nodeptr tmp = cur->parent;
		while (tmp != NULL && tmp->left == cur)
			cur = tmp, tmp = tmp->parent;
		return tmp;
	}
	nodeptr find(nodeptr root, int key) {
		if (root == NULL)
			return NULL;
		if (key == root->key)
			return root;
		if (key < root->key)
			return find(root->left, key);
		return find(root->right, key);
	}
	nodeptr insert(nodeptr root, int key) {
		if (root == NULL)
			root = new node(key);
		else if (key < root->key) {
			root->left = insert(root->left, key);
			root->left->parent = root;
		}
		else if (key > root->key) {
			root->right = insert(root->right, key);
			root->right->parent = root;
		}
		return root;
	}
	nodeptr erase(nodeptr root, int key) {
		if (root == NULL)
			return root;
		if (key < root->key) {
			root->left = erase(root->left, key);
			root->left->parent = root;
		}
		else if (key > root->key) {
			root->right = erase(root->right, key);
			root->right->parent = root;
		}
		else {
			nodeptr tmp;
			if (root->left == NULL || root->right == NULL) {
				if (root->left == NULL)
					tmp = root->right;
				else
					tmp = root->left;
				free(root);
				return tmp;
			}
			else {
				tmp = successor(root);
				root->key = tmp->key;
				root->right = erase(root->right, tmp->key);
				root->right->parent = root;
			}
		}
		return root;
	}
public:
```

```
	nodeptr root;
	BST() :
		root(NULL) {
	}
	nodeptr find(int key) {
		return find(root, key);
	}
	void insert(int key) {
		root = insert(root, key);
	}
	void erase(int key) {
		root = erase(root, key);
	}
};

void inorder(nodeptr root) {
	if (root == NULL)
		return;
	inorder(root->left);
	cout << root->key << ' ';
	inorder(root->right);
}

void preorder(nodeptr root) {
	if (root == NULL)
		return;
	cout << root->key << ' ';
	preorder(root->left);
	preorder(root->right);
}

void postorder(nodeptr root) {
	if (root == NULL)
		return;
	postorder(root->left);
	postorder(root->right);
	cout << root->key << ' ';
}
```

AVL

```
struct AVLnode {
	int key, height;
	AVLnode* left, * right, * parent;
	static AVLnode* sentinel;
	AVLnode() {
		parent = left = right = sentinel;
		height = 0;
	}
	AVLnode(int key) : key(key) {
		parent = left = right = sentinel;
		height = 0;
	}
	void updateHeight() {
		height = 1 + max(left->height, right->height);
	}
	int balanceFactor() {
		return left->height - right->height;
	}
};

AVLnode* AVLnode::sentinel = new AVLnode();

class AVL : public BST {
```

```
	typedef AVLnode* nodeptr;
public:
	nodeptr root;
	AVL() : root(NULL) {}
	void insert(int key) { root = insert(root, key); }
private:
	nodeptr rightRotation(nodeptr Q) {
		nodeptr P = Q->left;
		Q->left = P->right;
		Q->left->parent = Q;
		P->right = Q;
		P->parent = Q->parent;
		Q->parent = P;
		Q->updateHeight();
		P->updateHeight();
		return P;
	}
	nodeptr leftRotation(nodeptr P) {
		nodeptr Q = P->right;
		P->right = Q->left;
		P->right->parent = P;
		Q->left = P;
		Q->parent = P->parent;
		P->parent = Q;
		Q->updateHeight();
		P->updateHeight();
		return Q;
	}
	nodeptr balance(nodeptr root) {
		if (root->balanceFactor() == 2) {
			if (root->left->balanceFactor() == -1)
				root->left = leftRotation(root->left);
			root = rightRotation(root);
		}
		else if (root->balanceFactor() == -2) {
			if (root->right->balanceFactor() == 1)
				root->right = rightRotation(root->right);
			root = leftRotation(root);
		}
		return root;
	}
	nodeptr insert(nodeptr root, int key) {
		if (root == AVLnode::sentinel)
			return root = new AVLnode(key);
		if (key < root->key) {
			root->left = insert(root->left, key);
			root->left->parent = root;
		}
		else if (key > root->key) {
			root->right = insert(root->right, key);
			root->right->parent = root;
		}
		root->updateHeight();
		root = balance(root);
		return root;
	}
};
```

Heap

```
template<class T, class cmp = less<T>>
class heap {
	vector<T> v;
	void check() const {
		assert(size() > 0);
	}
	int parent(const int& node) const {
		return (node == 0 ? -1 : (node - 1) / 2);
	}
	int right(const int& node) const {
		int r = 2 * node + 2;
		return (r < size() ? r : -1);
	}
	int left(const int& node) const {
		int l = 2 * node + 1;
		return (l < size() ? l : -1);
	}
	void reheapUp(const int& node) {
		if (node == 0 || cmp()(v[parent(node)], v[node]))
			return;
		swap(v[node], v[parent(node)]);
		reheapUp(parent(node));

	}
	void reheapDown(const int& node) {
		int child = left(node);
		if (child == -1)
			return;
		int rightChild = right(node);
		if (rightChild != -1 && cmp()(v[rightChild], v[child]))
			child = rightChild;
		if (cmp()(v[node], v[child]))
			return;
		swap(v[node], v[child]);
		reheapDown(child);
	}
public:
	int size() const {
		return v.size();
	}
	void push(const int& val) {
		v.push_back(val);
		reheapUp((int)v.size() - 1);
	}
	const T& top() const {
		check();
		return v[0];
	}
	void pop() {
		check();
		v[0] = v.back();
		v.pop_back();
		reheapDown(0);
	}
};
```

DSU bipartiteness

```
struct DSU_bipartiteness {
	vector<int> bipartite, rank;
	vector<pair<int, int>> parent;
	DSU_bipartiteness(int n) {
		bipartite = rank = vector<int>(n + 1, 1);
```

```
		parent = vector<pair<int, int>>(n + 1);
		for (int i = 0; i <= n; i++)
			parent[i] = { i, 0 };
	}
	pair<int, int> find_set(int x) {
		if (x == parent[x].first)
			return parent[x];
		int parity = parent[x].second;
		parent[x] = find_set(parent[x].first);
		parent[x].second ^= parity;
		return parent[x];
	}
	void union_sets(int x, int y) {
		pair<int, int> p = find_set(x);
		x = p.first;
		int paX = p.second;
		p = find_set(y);
		y = p.first;
		int paY = p.second;
		if (x == y) {
			if (paX == paY)
				bipartite[x] = false;
		}
		else {
			if (rank[x] < rank[y])
				swap(x, y);
			parent[y] = { x, paX ^ paY ^ 1 };
			bipartite[x] &= bipartite[y];
			if (rank[x] == rank[y])
				rank[x]++;
		}
	}
	bool is_bipartite(int x) {
		return bipartite[find_set(x).first];
	}
};
```

Big int

```
class BigInt {
private:
#define CUR (*this)
	const int BASE = 1000000000;
	vector<int> v;
public:
	BigInt() {
	}
	BigInt(const long long& val) {
		CUR = val;
	}
	BigInt(const string& val) {
		CUR = val;
	}
	int size() const {
		return v.size();
	}
	bool zero() const {
		return v.empty();
	}
	BigInt& operator =(long long val) {
		v.clear();
		while (val) {
			v.push_back(val % BASE);
			val /= BASE;
```

```
        }
        return CUR;
}
BigInt& operator =(const BigInt& a) {
        v = a.v;
        return CUR;
}
BigInt& operator=(const string& s) {
        CUR = 0;
        for (const char& ch : s)
                CUR = CUR * 10 + (ch - '0');
        return CUR;
}

/****************************compare****************************/
bool operator <(const BigInt& a) const {
        if (a.size() != size())
                return size() < a.size();
        for (int i = size() - 1; i >= 0; i--) {
                if (v[i] != a.v[i])
                        return v[i] < a.v[i];
        }
        return false;
}
bool operator >(const BigInt& a) const {
        return a < CUR;
}
bool operator ==(const BigInt& a) const {
        return (!(CUR < a) && !(a < CUR));
}
bool operator <=(const BigInt& a) const {
        return ((CUR < a) || !(a < CUR));
}

/******************************add******************************/
BigInt& operator +(const BigInt& a) const {
        BigInt res = CUR;
        int idx = 0, carry = 0;
        while (idx < a.size() || carry) {
                if (idx < a.size())
                        carry += a.v[idx];
                if (idx == res.size())
                        res.v.push_back(0);
                res.v[idx] += carry;
                carry = res.v[idx] / BASE;
                res.v[idx] %= BASE;
                idx++;
        }
        return res;
}
BigInt& operator +=(const BigInt& a) {
        CUR = CUR + a;
        return CUR;
}

/**************************multiply**************************/
BigInt& operator *(const BigInt& a) const {
        BigInt res;
        if (CUR.zero() || a.zero())
                return res;
        res.v.resize(size() + a.size());
        for (int i = 0; i < size(); i++) {
                if (v[i] == 0)
                        continue;
```

```
                long long carry = 0;
                for (int j = 0; carry || j < a.size(); j++) {
                        carry += 1LL * v[i] * (j < a.size() ? a.v[j] : 0);
                        while (i + j >= res.size())
                                res.v.push_back(0);
                        carry += res.v[i + j];
                        res.v[i + j] = carry % BASE;
                        carry /= BASE;
                }
        }
        while (!res.v.empty() && res.v.back() == 0)
                res.v.pop_back();
        return res;
    }
    BigInt& operator *=(const BigInt& a) {
        CUR = CUR * a;
        return CUR;
    }
    /****************************print****************************/
    friend ostream& operator<<(ostream& out, const BigInt& a) {
        out << (a.zero() ? 0 : a.v.back());
        for (int i = (int)a.v.size() - 2; i >= 0; i--)
                out << setfill('0') << setw(9) << a.v[i];
        return out;
    }
#undef CUR
};
```

Graph

Kruskal

```
struct edge {
	int from, to;
	ll weight;
	edge() {
		from = to = weight = 0;
	}
	edge(int from, int to, ll weight) :
		from(from), to(to), weight(weight) {
	}
	bool operator <(const edge& other) const {
		return weight < other.weight;
	}
};

vector<edge> edgeList;
//O(edges*log(edges))
pair<int, vector<edge>> MST_Kruskal(int n) {
	DSU uf(n);
	vector < edge > edges;
	int mstCost = 0;
	sort(edgeList.begin(), edgeList.end());
	for (auto e : edgeList)
		if (uf.union_sets(e.from, e.to)) {
			mstCost += e.weight;
			edges.push_back(e);
		}
	if (edges.size() != n - 1)
		return { INT_MAX,vector<edge>() };
	return { mstCost,edges };
}

int miniMax(int src, int dest, int n) {
	int max = INT_MIN;
	DSU uf(n);
	sort(edgeList.begin(), edgeList.end());
	for (auto e : edgeList) {
		if (uf.same_set(src, dest))
			return max;
		uf.union_sets(e.from, e.to);
		max = e.weight;
	}
	return max;
}

//O(edges*log(edges) + nodes*nodes)
pair<int, vector<edge>> SMST_Kruskal(int n) {
	DSU uf(n);
	sort(edgeList.begin(), edgeList.end());
	vector<edge> take, leave;
	int mstCost = 0;
	for (auto e : edgeList)
		if (uf.union_sets(e.from, e.to)) {
			mstCost += e.weight;
			take.push_back(e);
		}
		else
			leave.push_back(e);
	pair<int, vector<edge>> ret = { INT_MAX, vector<edge>() };
	for (int i = 0; i < take.size(); i++) {
		uf = DSU(n);
```

```
		vector < edge > edges;
		mstCost = 0;
		for (int j = 0; j < take.size(); j++) {
			if (i == j)
				continue;
			uf.union_sets(take[j].from, take[j].to);
			mstCost += take[j].weight;
			edges.push_back(take[j]);
		}
		for (edge e : leave) {
			if (uf.union_sets(e.from, e.to)) {
				mstCost += e.weight;
				edges.push_back(e);
				break;
			}
		}
		if (edges.size() == n - 1 && ret.first < mstCost)
			ret = { mstCost, edges };
	}
	return ret;
}
```

Prim

```
struct edge {
	int from, to, weight;
	edge() {
		from = to = weight = 0;
	}
	edge(int from, int to, int weight) :
		from(from), to(to), weight(weight) {
	}
	bool operator <(const edge& other) const {
		return weight > other.weight;
	}
};

vector<vector<edge>> adj;
vector<edge> prim(int node) {
	vector<bool> vis(adj.size());
	priority_queue<edge> q;
	vector<edge> edges;
	q.push(edge(-1, node, 0));
	while (!q.empty()) {
		edge e = q.top();
		q.pop();
		if (vis[e.to])
			continue;
		vis[e.to] = true;
		if (e.from != -1)
			edges.push_back(e);
		for (edge ch : adj[e.to])
			if (!vis[ch.to])
				q.push(ch);
	}
	return edges;//check it connected or not
}
```

SMST

```
struct edge {
	int from, to;
	ll weight;
	edge() {
		from = to = weight = 0;
```

```
	}
	edge(int from, int to, ll weight) :
		from(from), to(to), weight(weight) {
	}
	bool operator <(const edge& other) const {
		return weight < other.weight;
	}
};

struct DSU {
	vector<int> rank, parent, size;
	int forsets;
	DSU(int n) {
		size = rank = parent = vector<int>(n + 1, 1);
		forsets = n;
		for (int i = 0; i <= n; i++)
			parent[i] = i;
	}
	int find_set(int v) {
		if (v == parent[v])
			return v;
		return parent[v] = find_set(parent[v]);
	}
	void link(int par, int node) {
		parent[node] = par;
		size[par] += size[node];
		if (rank[par] == rank[node])
			rank[par]++;
		forsets--;
	}
	bool union_sets(int v, int u) {
		v = find_set(v), u = find_set(u);
		if (v == u)
			return false;
		if (rank[v] < rank[u])
			swap(v, u);
		link(v, u);
		return true;
	}
	bool same_set(int v, int u) {
		return find_set(v) == find_set(u);
	}
	int size_set(int v) {
		return size[find_set(v)];
	}
};

int MST_Kruskal(int n, vector<edge> edgeList, vector<edge>& take,
	vector<edge>& leave) {
	DSU uf(n);
	vector<edge> edges;
	sort(edgeList.begin(), edgeList.end());
	int mst_cost = 0;
	for (auto e : edgeList)
		if (uf.union_sets(e.from, e.to)) {
			take.push_back(e);
			mst_cost += e.weight;
		}
		else
			leave.push_back(e);
	return mst_cost;
}

struct LCA {
```

```cpp
#define INIT { -1, -2 }
	struct data {
		int lca = -1;
		pair<int, int> max_edges = INIT; //first max,second max (distinct)
	};
	pair<int, int> merge(pair<int, int> a, pair<int, int> b) {
		if (a.first < b.first)
			swap(a, b);
		if (b.first == a.first)
			a.second = max(a.second, b.second);
		else if (b.first > a.second)
			a.second = b.first;
		return a;
	}
	int logN;
	vector<vector<data>> lca;
	vector<vector<edge>> adj;
	vector<int> depth;
	void dfs(int node, int par) {
		for (edge e : adj[node])
			if (e.to != par) {
				depth[e.to] = depth[node] + 1;
				lca[e.to][0].max_edges.first = e.weight;
				lca[e.to][0].lca = node;
				dfs(e.to, node);
			}
	}
	LCA(int n, vector<edge>& edges) :
		adj(n + 1) {
		for (auto& e : edges) {
			adj[e.from].push_back(e);
			adj[e.to].push_back(edge(e.to, e.from, e.weight));
		}
		logN = log2(n);
		depth = vector<int>(n + 1);
		lca = vector<vector<data>>(n + 1, vector<data>(logN + 1));
		dfs(1, -1);
		for (int k = 1; k <= logN; k++)
			for (int node = 1; node <= n; node++) {
				int par = lca[node][k - 1].lca;
				if (~par) {
					lca[node][k].lca = lca[par][k - 1].lca;
					lca[node][k].max_edges = merge(lca[node][k - 1].max_edges,
						lca[par][k - 1].max_edges);
				}
			}
	}
	pair<int, int> max_two_edges(int u, int v) {
		pair<int, int> ans = INIT;
		if (depth[u] < depth[v])
			swap(u, v);
		for (int i = logN; i >= 0; i--)
			if (depth[u] - (1 << i) >= depth[v]) {
				ans = merge(ans, lca[u][i].max_edges);
				u = lca[u][i].lca;
			}
		if (u == v)
			return ans;
		for (int i = logN; i >= 0; i--)
			if (lca[u][i].lca != lca[v][i].lca) {
				ans = merge(ans, lca[u][i].max_edges);
				ans = merge(ans, lca[v][i].max_edges);
				u = lca[u][i].lca;
				v = lca[v][i].lca;
```

```
				}
				ans = merge(ans, lca[u][0].max_edges);
				ans = merge(ans, lca[v][0].max_edges);
				return ans;
		}
};

int main() {
		run();
		int t;
		cin >> t;
		for (int I = 1; I <= t; I++) {
				cout << "Case #" << I << " : ";
				int n, e;
				cin >> n >> e;
				vector<edge> edgeList(e);
				for (auto& it : edgeList)
						cin >> it.from >> it.to >> it.weight;
				vector<edge> take, leave;
				int mst_cost = MST_Kruskal(n, edgeList, take, leave);
				if (take.size() != n - 1) {
						cout << "No way\n";
						continue;
				}
				LCA tree(n, take);
				ll rt = INF;
				for (edge e : leave) {
						pair<int, int> p = tree.max_two_edges(e.from, e.to);
						rt = min(rt, mst_cost - p.first + e.weight);
				}
				if (rt == INF)
						cout << "No second way\n";
				else
						cout << rt << endl;
		}
}
```

Dijkstra

```
struct edge {
		int from, to, weight;
		edge() {
				from = to = weight = 0;
		}
		edge(int from, int to, int weight) :
				from(from), to(to), weight(weight) {
		}
		bool operator <(const edge& other) const {
				return weight > other.weight;
		}
};
vector<vector<edge>> adj;

//O(E*log(v))
void dijkstra(int src, int dest = -1) {
		priority_queue<edge> q;
		vector<int> dis(adj.size(), INT_MAX), prev(adj.size(), -1);
		q.push(edge(-1, src, 0));
		dis[src] = 0;
		while (!q.empty()) {
				edge e = q.top();
				q.pop();
				if (e.weight > dis[e.to])
						continue;
```

```
			prev[e.to] = e.from;
			if (e.to == dest)
				break;
			for (edge ne : adj[e.to])
				if (dis[ne.to] > dis[e.to] + ne.weight) {
					ne.weight = dis[ne.to] = dis[e.to] + ne.weight;
					q.push(ne);
				}
	}
	vector<int> path;
	while (dest != -1) {
		path.push_back(dest);
		dest = prev[dest];
	}
	reverse(path.begin(), path.end());
}
```

Floyed

```
vector<vector<int>> adj, par;
// adj[i][j] = oo , adj[i][i] = 0
// par[i][j] = i

void init(int n) {
	par = adj = vector<vector<int>>(n + 1, vector<int>(n + 1, oo));
	for (int i = 1; i <= n; i++)
		adj[i][i] = 0;
	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= n; j++)
			par[i][j] = i;
}

void floyd() {
	for (int k = 1; k < adj.size(); k++)
		for (int i = 1; i < adj.size(); i++)
			for (int j = 1; j < adj.size(); j++)
				if (adj[i][j] > adj[i][k] + adj[k][j]) {
					adj[i][j] = adj[i][k] + adj[k][j];
					par[i][j] = par[k][j];
				}
}

void buildPath(int src, int dest) {
	vector<int> path;
	while (src != dest) {
		path.push_back(dest);
		dest = par[src][dest];
	}
	path.push_back(src);
	reverse(path.begin(), path.end());
}
```

Bellmanford

```
#define oo 0x3f3f3f3fLL
struct edge {
	int from, to, weight;
	edge() {
		from = to = weight = 0;
	}
	edge(int from, int to, int weight) :
		from(from), to(to), weight(weight) {
	}
	bool operator <(const edge& other) const {
```

```
		return weight > other.weight;
	}
};

vector<edge> edgeList;
//O(V*E)
void bellmanford(int n, int src, int dest = -1) {
	vector<int> dis(n + 1, oo), prev(n + 1, -1);
	dis[src] = 0;
	bool negativeCycle = false;
	int last = -1, tmp = n;
	while (tmp--) {
		last = -1;
		for (edge e : edgeList)
			if (dis[e.to] > dis[e.from] + e.weight) {
				dis[e.to] = dis[e.from] + e.weight;
				prev[e.to] = e.from;
				last = e.to;
			}
		if (last == -1)
			break;
		if (tmp == 0)
			negativeCycle = true;
	}
	if (last != -1) {
		for (int i = 0; i < n; i++)
			last = prev[last];
		vector<int> cycle;
		for (int cur = last; cur != last || cycle.size() > 1; cur = prev[cur])
			cycle.push_back(cur);
		reverse(cycle.begin(), cycle.end());
	}
	vector<int> path;
	while (dest != -1) {
		path.push_back(dest);
		dest = prev[dest];
	}
	reverse(path.begin(), path.end());
}
```

Difference constraints

```
void difference_constraints() {
	int m;
	cin >> m;
	int cnt = 1;
	while (m--) {
		string x1, x2;
		int w; // x1 - x2 <= w
		cin >> x1 >> x2 >> w;
		map<string, int> id;
		if (id.find(x1) == id.end())
			id[x1] = cnt++;
		if (id.find(x2) == id.end())
			id[x2] = cnt++;
		edgeList.emplace_back(id[x2], id[x1], w);
	}
	for (int i = 1; i < cnt; i++)
		edgeList.emplace_back(cnt, i, 0);
	bellmanford(cnt, cnt);
}
```

SPFA

```
vector<vector<edge>> adj;

void spfa(int src) {
	enum visit {
		finished, in_queue, not_visited
	};
	int n = adj.size();
	vector<int> dis(n, INF), prev(n, -1), state(n, not_visited);
	dis[src] = 0;
	deque<int> q;
	q.push_back(src);
	while (!q.empty()) {
		int u = q.front();
		q.pop_front();
		state[u] = finished;
		for (auto& e : adj[u]) {
			if (dis[e.to] > dis[e.from] + e.cost) {
				dis[e.to] = dis[e.from] + e.cost;
				prev[e.to] = e.from;
				if (state[e.to] == not_visited) {
					state[e.to] = in_queue;
					q.push_back(e.to);
				}
				else if (state[e.to] == finished) {
					state[e.to] = in_queue;
					q.push_front(e.to);
				}
			}
		}
	}
}
```

SCC

```
vector<vector<int>> adj, scc;
vector<set<int>> dag;
vector<int> dfs_num, dfs_low, compId;
vector<bool> inStack;
stack<int> stk;
int timer;

void dfs(int node) {
	dfs_num[node] = dfs_low[node] = ++timer;
	stk.push(node);
	inStack[node] = 1;
	for (int child : adj[node])
		if (!dfs_num[child]) {
			dfs(child);
			dfs_low[node] = min(dfs_low[node], dfs_low[child]);
		}
		else if (inStack[child])
			dfs_low[node] = min(dfs_low[node], dfs_num[child]);
	//can be dfs_low[node] = min(dfs_low[node], dfs_low[child]);
	if (dfs_low[node] == dfs_num[node]) {
		scc.push_back(vector<int>());
		int v = -1;
		while (v != node) {
			v = stk.top();
			stk.pop();
			inStack[v] = 0;
			scc.back().push_back(v);
			compId[v] = scc.size() - 1;
		}
```

```
	}
}

void SCC() {
	timer = 0;
	dfs_num = dfs_low = compId = vector<int>(adj.size());
	inStack = vector<bool>(adj.size());
	scc = vector<vector<int>>();
	for (int i = 1; i < adj.size(); i++)
		if (!dfs_num[i])
			dfs(i);
}

void DAG() {
	dag = vector<set<int>>(scc.size());
	for (int i = 1; i < adj.size(); i++)
		for (int j : adj[i])
			if (compId[i] != compId[j])
				dag[compId[i]].insert(compId[j]);
}
```

articulation_points_and_bridges

```
vector<vector<int>> adj;
vector<int> dfs_num, dfs_low;
vector<bool> articulation_point;
vector<pair<int, int>> bridge;
stack<pair<int, int>> edges;
vector<vector<pair<int, int>>> BCC; //biconnected components
int timer, cntChild;

// O(n + m)
void dfs(int node, int par) {
	dfs_num[node] = dfs_low[node] = ++timer;
	for (int child : adj[node]) {
		if (par != child && dfs_num[child] < dfs_num[node])
			edges.push({ node, child });

		if (!dfs_num[child]) {
			if (par == -1)
				cntChild++;
			dfs(child, node);
			if (dfs_low[child] >= dfs_num[node]) {
				articulation_point[node] = 1;
				//get biconnected component
				BCC.push_back(vector<pair<int, int>>());
				pair<int, int> edge;
				do {
					edge = edges.top();
					BCC.back().push_back(edge);
					edges.pop();
				} while (edge.first != node || edge.second != child);
			}
			if (dfs_low[child] > dfs_num[node]) //can be (dfs_low[child] == dfs_num[child])
				bridge.push_back({ node, child });
			dfs_low[node] = min(dfs_low[node], dfs_low[child]);
		}
		else if (child != par)
			dfs_low[node] = min(dfs_low[node], dfs_num[child]);
	}
}

void articulation_points_and_bridges() {
	timer = 0;
```

```
	dfs_num = dfs_low = vector<int>(adj.size());
	articulation_point = vector<bool>(adj.size());
	bridge = vector<pair<int, int>>();
	for (int i = 1; i < adj.size(); i++)
		if (!dfs_num[i]) {
			cntChild = 0;
			dfs(i, -1);
			articulation_point[i] = cntChild > 1;
		}
}
```

Edge classification

```
vector<vector<int>> adj;
vector<int> start, finish;
int timer;

void dfsEdgeClassification(int node) {
	start[node] = timer++;
	for (int child : adj[node]) {
		if (start[child] == -1)
			dfsEdgeClassification(child);
		else {
			if (finish[child] == -1)
				; // Back Edge
			else if (start[node] < start[child])
				; // Forward Edge
			else
				; // Cross Edge
		}
	}
	finish[node] = timer++;
}
```

2-sat

```
#include"..\strongly_connected_component.h"
int n;
int Not(int x) {
	return (x > n ? x - n : x + n);
}

void addEdge(int a, int b) {
	adj[Not(a)].push_back(b);
	adj[Not(b)].push_back(a);
}

void add_xor_edge(int a, int b) {
	addEdge(Not(a), Not(b));
	addEdge(a, b);
}

bool _2SAT(vector<int>& value) {
	SCC();
	for (int i = 1; i <= n; i++)
		if (compId[i] == compId[Not(i)])
			return false;
	vector<int> assign(scc.size(), -1);
	for (int i = 0; i < scc.size(); i++)
		if (assign[i] == -1) {
			assign[i] = true;
			assign[compId[Not(scc[i].back())]] = false;
		}
	for (int i = 1; i <= n; i++)
		value[i] = assign[compId[i]];
```

```
	return true;
}
```

Maximum bipartite matching

```
vector<vector<int>> adj;
vector<int> rowAssign, colAssign, vis;
int test_id;
bool canMatch(int i) {
	for (int j : adj[i]) {
		if (vis[j] == test_id)
			continue;
		vis[j] = test_id;
		if (colAssign[j] == -1 || canMatch(colAssign[j])) {
			colAssign[j] = i;
			rowAssign[i] = j;
			return true;
		}
	}
	return false;
}

// O(rows * edges) //number of operation could by strictly less than order (1e5*1e5->AC)
int maximum_bipartite_matching(int rows, int cols) {
	int maxFlow = 0;
	rowAssign = vector<int>(rows, -1);
	colAssign = vector<int>(cols, -1);
	vis = vector<int>(cols);
	for (int i = 0; i < rows; i++) {
		test_id++;
		if (canMatch(i))
			maxFlow++;
	}
	vector<pair<int, int>> matches;
	for (int j = 0; j < cols; j++)
		if (~colAssign[j])
			matches.push_back({ colAssign[j], j });
	return maxFlow;
}
```

String

Hashing

```
struct hashing {
	int MOD, BASE;
	vector<int> Hash, modInv;
	hashing(string s, int MOD, int BASE, char first_char = 'a') :
		MOD(MOD), BASE(BASE), Hash(sz(s) + 1), modInv(sz(s) + 1) {
		modInv[0] = 1;
		ll base = 1;
		for (int i = 1; i <= sz(s); i++) {
			Hash[i] = (Hash[i - 1] + (s[i - 1] - first_char + 1) * base) % MOD;
			modInv[i] = power(base, MOD - 2, MOD);
			base = (base * BASE) % MOD;
		}
	}
	int getHash(int l, int r) { //1-based
		return ((Hash[r] - Hash[l - 1] + MOD) % MOD * modInv[l]) % MOD;
	}
};

//MOD = 1e9 + 9 ,BASE = 31
//MOD = 2000000011 ,BASE = 53 ->careful of overflow
//*************
//MOD = 998634293,BASE = 953
//MOD = 986464091,BASE = 1013
```

KMP

```
string pattern;
vector<int> longestPrefix;
int fail(int k, char nxt) {
	while (k > 0 && pattern[k] != nxt)
		k = longestPrefix[k - 1];
	if (nxt == pattern[k])
		k++;
	return k;
}

void failure_function() {
	int n = pattern.size();
	longestPrefix = vector<int>(n);
	for (int i = 1, k = 0; i < n; i++)
		longestPrefix[i] = k = fail(k, pattern[i]);
}

void KMP(const string& str) {
	int n = str.size();
	int m = pattern.size();
	for (int i = 0, k = 0; i < n; i++) {
		k = fail(k, str[i]);
		if (k == m) {
			cout << i - m + 1 << endl; //0-based
			k = longestPrefix[k - 1]; // if you want next match
		}
	}
}
```

Trie tree

```
class trie {
	struct trie_node {
		bool is_leaf = false;
		map<char, int> next;
		bool have_next(char ch) {
			return next.find(ch) != next.end();
		}
		int& operator[](char ch) {
			return next[ch];
		}
	};
	vector<trie_node> t;
public:
	trie() {
		t.push_back(trie_node());
	}
	void insert(const string& s) {
		int root = 0;
		for (const char& ch : s) {
			if (!t[root].have_next(ch)) {
				t.push_back(trie_node());
				t[root][ch] = t.size() - 1;
				//t[root][ch] = add_node();doesn't work if next is array
			}
			root = t[root][ch];
		}
		t[root].is_leaf = true;
	}
	bool find(const string& s) {
		int root = 0;
		for (const char& ch : s) {
			if (!t[root].have_next(ch))
				return false;
			root = t[root][ch];
		}
		return t[root].is_leaf;
	}
};
```

Suffix array

```
class suffix_array {
	int getOrder(int a) const {
		return (a < (int)order.size() ? order[a] : 0);
	}
	void radix_sort(int k) {
		vector<int> frq(n), tmp(n);
		for (auto& it : suf)
			frq[getOrder(it + k)]++;
		for (int i = 1; i < n; i++)
			frq[i] += frq[i - 1];
		for (int i = n - 1; i >= 0; i--)
			tmp[--frq[getOrder(suf[i] + k)]] = suf[i];
		suf = tmp;
	}
public:
	int n;
	string s;
	vector<int> suf, lcp, order; // order store position of suffix i in suf array
	suffix_array(const string& s) :
		n(s.size() + 1), s(s) {
		suf = order = vector<int>(n);
		vector<int> newOrder(n);
		for (int i = 0; i < n; i++)
			suf[i] = i;
		{ //sort according to first character
			vector<int> tmp(n);
			for (int i = 0; i < n; i++)
				tmp[i] = s[i];
			sort(all(tmp));
			for (int i = 0; i < n; i++)
				order[i] = (lower_bound(all(tmp), s[i]) - tmp.begin());
		}
		for (int len = 1; newOrder.back() != n - 1; len <<= 1) {
			auto cmp = [&](const int& a, const int& b) {
				if (order[a] != order[b])
					return order[a] < order[b];
				return getOrder(a + len) < getOrder(b + len);
			};
			//sort(all(suf), cmp); run in 500ms  (n<=4e5)
			radix_sort(len); //sort second part
			radix_sort(0); //sort first part
			newOrder[0] = 0;
			for (int i = 1; i < n; i++)
				newOrder[i] = newOrder[i - 1] + cmp(suf[i - 1], suf[i]);
			for (int i = 0; i < n; i++)
				order[suf[i]] = newOrder[i];
		}
		buildLCP();
	}
	/*
	 * longest common prefix
	 * O(n)
	 * lcp[i] = lcp(suf[i],suf[i-1])
	 */
	void buildLCP() {
		lcp = vector<int>(n);
		int k = 0;
		for (int i = 0; i < n - 1; i++) {
			int pos = order[i];
			int j = suf[pos - 1];
			while (s[i + k] == s[j + k])
				k++;
			lcp[pos] = k;
```

```
				if (k)
					k--;
			}
		}
		//LCP(i,j) : longest common prefix between suffix i and suffix j
		int LCP(int i, int j) {
			if (order[j] < order[i])
				swap(i, j);
			int mn = n - i - 1;
			for (int k = order[i] + 1; k <= order[j]; k++)
				mn = min(mn, lcp[k]);
			return mn;
		}
};

ll number_of_different_substrings(string s) {
		int n = s.size();
		suffix_array sa(s);
		ll cnt = 0;
		for (int i = 0; i <= n; i++)
			cnt += n - sa.suf[i] - sa.lcp[i];
		return cnt;
}

string largest_common_substring(const string& s1, const string& s2) {
		suffix_array sa(s1 + "#" + s2);
		vector<int> suf = sa.suf, lcp = sa.lcp;
		auto type = [&](int idx) {
			return idx < s1.size() + 1;
		};
		int mx = 0, idx = 0;
		int len = s1.size() + 1 + s2.size();
		for (int i = 1; i <= len; i++)
			if (type(suf[i - 1]) != type(suf[i]) && lcp[i] > mx) {
				mx = lcp[i];
				idx = min(suf[i - 1], suf[i]);
			}
		return s1.substr(idx, mx);
}
```

Aho corasick

```
struct aho_corasick {
		struct trie_node {
			vector<int> pIdxs; //probably take memory limit
			map<char, int> next;
			int fail;
			trie_node() :
				fail(-1) {
			}
			bool have_next(char ch) {
				return next.find(ch) != next.end();
			}
			int& operator[](char ch) {
				return next[ch];
			}
		};
		vector<trie_node> t;
		void insert(const string& s, int patternIdx) {
			int root = 0;
			for (const char& ch : s) {
				if (!t[root].have_next(ch)) {
					t.push_back(trie_node());
					t[root][ch] = t.size() - 1;
```

```
				}
				root = t[root][ch];
			}
			t[root].pIdxs.push_back(patternIdx);
	}
	vector<string> patterns;
	aho_corasick(const vector<string>& _patterns) {
			t.push_back(trie_node());
			patterns = _patterns;
			for (int i = 0; i < patterns.size(); i++)
				insert(patterns[i], i);
			buildAhoTree();
	}

	int next_state(int cur, char ch) {
			while (cur > 0 && !t[cur].have_next(ch))
				cur = t[cur].fail;
			if (t[cur].have_next(ch))
				return t[cur][ch];
			return 0;
	}
	void buildAhoTree() {
			queue<int> q;
			for (auto& child : t[0].next) {
				q.push(child.second);
				t[child.second].fail = 0;
			}
			while (!q.empty()) {
				int cur = q.front();
				q.pop();
				for (auto& child : t[cur].next) {
					int k = next_state(t[cur].fail, child.first);
					t[child.second].fail = k;
					vector<int>& idxs = t[child.second].pIdxs;
					//dp[child.second] = max(dp[child.second],dp[k]);
					idxs.insert(idxs.end(), all(t[k].pIdxs));
					q.push(child.second);
				}
			}
	}
	vector<vector<int>> match(const string& str) {
			int k = 0;
			vector<vector<int>> rt(patterns.size());
			for (int i = 0; i < str.size(); i++) {
				k = next_state(k, str[i]);
				for (auto& it : t[k].pIdxs)
					rt[it].push_back(i);
			}
			return rt;
	}
};
```

Math

Combinatorics

```
/*
 * nCr = n!/((n-r)! * r!)
 * nCr(n,r) = nCr(n,n-r)
 * nPr = n!/(n-r)!
 * nPr(circle) = nPr/r
 * nCr(n,r) = pascal[n][r]
 * catalan[n] = nCr(2n,n)/(n+1)
 */

ull nCr(int n, int r) {
	if (r > n)
		return 0;
	r = max(r, n - r);
	ull ans = 1, div = 1, i = r + 1;
	while (i <= n) {
		ans *= i++;
		ans /= div++;
	}
	return ans;
}

ull nPr(int n, int r) {
	if (r > n)
		return 0;
	ull p = 1, i = n - r + 1;
	while (i <= n)
		p *= i++;
	return p;
}

vector<vector<ull>> pascalTriangle(int n) {
	vector<vector<ull>> pascal(n + 1, vector<ull>(n + 1));
	for (int i = 0; i <= n; i++) {
		pascal[i][i] = pascal[i][0] = 1;
		for (int j = 1; j < n; j++)
			pascal[i][j] = pascal[i - 1][j] + pascal[i - 1][j - 1];
	}
	return pascal;
}

// return catalan number n-th using dp O(n^2)//max = 35 then overflow
vector<ull> catalanNumber(int n) {
	vector<ull> catalan(n + 1);
	catalan[0] = catalan[1] = 1;
	for (int i = 2; i <= n; i++) {
		ull& rt = catalan[i];
		for (int j = 0; j < n; j++)
			rt += catalan[j] * catalan[n - j - 1];
	}
	return catalan;
}

// count number of paths in matrix n*m
// go to right or down only
ull countNumberOfPaths(int n, int m) {
	return nCr(n + m - 2, n - 1);
}
```

Matrices

```
typedef vector<int> row;
typedef vector<row> matrix;

matrix initial(int n, int m, int val = 0) {
	return matrix(n, row(m, val));
}

matrix identity(int n) {
	matrix rt = initial(n, n);
	for (int i = 0; i < n; i++)rt[i][i] = 1;
	return rt;
}

matrix addIdentity(const matrix& a) {
	matrix rt = a;
	for (int i = 0; i < sz(a); i++)rt[i][i] += 1;
	return rt;
}

matrix add(const matrix& a, const matrix& b) {
	matrix rt = initial(sz(a), sz(a[0]));
	for (int i = 0; i < sz(a); i++)for (int j = 0; j < sz(a[0]); j++)
		rt[i][j] = a[i][j] + b[i][j];
	return rt;
}

matrix multiply(const matrix& a, const matrix& b) {
	matrix rt = initial(sz(a), sz(b[0]));
	for (int i = 0; i < sz(a); i++) for (int k = 0; k < sz(a[0]); k++) {
		if (a[i][k] == 0)continue;
		for (int j = 0; j < sz(b[0]); j++)
			rt[i][j] += a[i][k] * b[k][j];
	}
	return rt;
}

matrix power(const matrix& a, ll k) {
	if (k == 0)return identity(sz(a));
	if (k & 1)return multiply(a, power(a, k - 1));
	return power(multiply(a, a), k >> 1);
}
matrix power_itr(matrix a, ll k) {
	matrix rt = identity(sz(a));
	while (k) {
		if (k & 1)rt = multiply(rt, a);
		a = multiply(a, a); k >>= 1;
	}
	return rt;
}
matrix sumPower(const matrix& a, ll k) {
	if (k == 0)return initial(sz(a), sz(a));
	if (k & 1)return multiply(a, addIdentity(sumPower(a, k - 1)));
	return multiply(sumPower(a, k >> 1), addIdentity(power(a, k >> 1)));
}
matrix sumPowerV2(const matrix& a, ll k) {
	int n = sz(a);
	matrix rt = initial(2 * n, 2 * n);
	for (int i = 0; i < 2 * n; i++)
		for (int j = 0; j < n; j++)
			rt[i][j] = a[i % n][j];
	for (int i = n; i < 2 * n; i++)rt[i][i] = 1;
	return power(rt, k);
}
```

Matrix class

```
struct matrix {
	using T = int;
	using row = vector<T>;
	vector<vector<T>> v;
	matrix() {

	}
	matrix(int n, int m, T val = 0) :
		v(n, row(m, val)) {
	}
	int size() const {
		return v.size();
	}
	int cols() const {
		return v[0].size();
	}
	matrix operator*(T a) const {
		matrix rt = *this;
		REP(i, rt.size())
			REP(j, rt.cols())
			rt.v[i][j] *= a;
		return rt;
	}
	friend matrix operator*(T a, const matrix& b) {
		return (b * a);
	}
	friend matrix operator+(const matrix& a, const matrix& b) {
		assert(a.size() == b.size() && a.cols() == b.cols());
		matrix rt(a.size(), a.cols());
		REP(i, rt.size())
			REP(j, rt.cols())
			rt.v[i][j] = a.v[i][j] + b.v[i][j];
		return rt;
	}
	friend matrix operator*(const matrix& a, const matrix& b) {
		assert(a.cols() == b.size());
		matrix rt(a.size(), b.cols());
		REP(i, rt.size())
			REP(k, a.cols())
		{
			if (a.v[i][k] == 0)
				continue;
			REP(j, rt.cols())
				rt.v[i][j] += a.v[i][k] * b.v[k][j];
		}
		return rt;
	}
};

matrix identity(int n) {
	matrix r(n, n);
	for (int i = 0; i < n; i++)
		r.v[i][i] = 1;
	return r;
}

matrix power(matrix a, long long y) {
	assert(y >= 0 && a.size() == a.cols());
	matrix rt = identity(a.size());
	while (y > 0) {
		if (y & 1)
			rt = rt * a;
		a = a * a;
```

```
            y >>= 1;
        }
        return rt;
}

matrix addIdentity(const matrix& a) {
        matrix rt = a;
        REP(i, a.size())
                rt.v[i][i]++;
        return rt;
}

matrix sumPower(const matrix& a, ll k) {
        if (k == 0)
                return matrix(sz(a), sz(a));
        if (k & 1)
                return a * addIdentity(sumPower(a, k - 1));
        return (sumPower(a, k >> 1) * addIdentity(power(a, k >> 1)));
}

matrix sumPowerV2(const matrix& a, ll k) {
        int n = sz(a);
        matrix rt(2 * n, 2 * n);
        REP(i, n)
                REP(j, n)
        {
                rt.v[i][j] = a.v[i][j];
                rt.v[i + n][j] = a.v[i][j];
        }
        for (int i = n; i < 2 * n; i++)
                rt.v[i][i] = 1;
        return power(rt, k);
}
```

Mod inverse

```
ll power(ll x, ll y, int mod) {
        if (y == 0)
                return 1;
        if (y == 1)
                return x % mod;
        ll r = power(x, y >> 1, mod);
        return (((r * r) % mod) * power(x, y & 1, mod)) % mod;
}

ll modInverse(ll b, ll mod) { // if mod is Prime
        return power(b, mod - 2, mod);
}

// Calulate Modular inverse // don't work correctly
ll modInv(ll a, ll m) {
        ll m0 = m, t, q;
        ll x0 = 0, x1 = 1;
        if (m == 1)
                return 0;
        while (a > 1) {
                q = a / m;
                t = m;
                m = a % m, a = t;
                t = x0;
                x0 = x1 - q * x0;
                x1 = t;
        }
        if (x1 < 0)
```

```
            x1 += m0;
    return x1;
}

template<typename T>
T inverse(T a, T m) { //tourist code
    T u = 0, v = 1;
    while (a != 0) {
        T t = m / a;
        m -= t * a;
        swap(a, m);
        u -= t * v;
        swap(u, v);
    }
    assert(m == 1);
    return u;
}
```

(a^n)%p=result , return n

```
// (a^n)%p=result,return minimum n
int getPower(int a, int result, int mod) {
    int sq = sqrt(mod);
    map<int, int> mp;
    ll r = 1;
    for (int i = 0; i < sq; i++) {
        if (mp.find(r) == mp.end())
            mp[r] = i;
        r = (r * a) % mod;
    }
    ll tmp = modInverse(r, mod);
    ll cur = result;
    for (int i = 0; i <= mod; i += sq) {
        if (mp.find(cur) != mp.end())
            return i + mp[cur];
        cur = (cur * tmp) % mod;//val/(a^sq)
    }
    return INF;
}

// Returns minimum x for which a ^ x % m = b % m.
// a,m not not coprime
int getPower(int a, int b, int m) {
    a %= m, b %= m;
    int k = 1, add = 0, g;
    while ((g = __gcd(a, m)) > 1) {
        if (b == k)
            return add;
        if (b % g)
            return -1;
        b /= g, m /= g, ++add;
        k = (k * 1ll * a / g) % m;
    }

    int n = sqrt(m) + 1;
    int an = 1;
    for (int i = 0; i < n; ++i)
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
```

```
		cur = (cur * 1ll * a) % m;
	}

	for (int p = 1, cur = k; p <= n; ++p) {
		cur = (cur * 1ll * an) % m;
		if (vals.count(cur)) {
			int ans = n * p - vals[cur] + add;
			return ans;
		}
	}
	return -1;
}
```

NCR pre calculation

```
const int N = 1e5 + 100;
const int mod = 1e9 + 7;
ll fact[N];
ll inv[N]; //mod inverse for i
ll invfact[N]; //mod inverse for i!
void factInverse() {
	fact[0] = inv[1] = fact[1] = invfact[0] = invfact[1] = 1;
	for (long long i = 2; i < N; i++) {
		fact[i] = (fact[i - 1] * i) % mod;
		inv[i] = mod - (inv[mod % i] * (mod / i) % mod);
		invfact[i] = (inv[i] * invfact[i - 1]) % mod;
	}
}

ll nCr(int n, int r) {
	if (r > n)
		return 0;
	return (((fact[n] * invfact[r]) % mod) * invfact[n - r]) % mod;
}
```

Primes

```
typedef vector<pair<ll, int>> primeFactors;

const int N = 1e8;
bool isPrime[N + 1];
vector<int> prime;
void sieve() {
	memset(isPrime, true, sizeof(isPrime));
	isPrime[0] = isPrime[1] = false;
	for (int i = 4; i <= N; i += 2)
		isPrime[i] = false;
	for (int i = 3; i * i <= N; i += 2)
		if (isPrime[i])
			for (int j = i * i; j <= N; j += i + i)
				isPrime[j] = false;
	for (int i = 1; i <= N; i++)
		if (isPrime[i])
			prime.push_back(i);
}

// generate prime divisors in n
// n = p1^x1 * p2^x2 .... pn^xn
// O(sqrt(n)) // max = 1e16
primeFactors prime_factors(ll n) {
	primeFactors p;
	int idx = 0;
	while (!(n <= N && isPrime[n]) && idx < prime.size()
```

```
			&& 1LL * prime[idx] * prime[idx] <= n) {
			int cnt = 0;
			while (n % prime[idx] == 0)
				n /= prime[idx], cnt++;
			if (cnt)
				p.push_back({ prime[idx], cnt });
			idx++;
		}
		if (n > 1)
			p.push_back({ n, 1 });
		return p;
}

// return number of Divisors(n) using prime factorization
ll numOfDivisors(primeFactors mp) {
		ll cnt = 1;
		for (auto it : mp)
			cnt *= (it.second + 1);
		return cnt;
}

// return sum of Divisors(n) using prime factorization
ll sumOfDivisors(primeFactors mp) {
		ll sum = 1;
		for (auto it : mp)
			sum *= sumPower(it.first, it.second);
		return sum;
}
ll phi_function(ll n) {
		double result = n;
		primeFactors pf = prime_factors(n);
		for (auto& it : pf) {
			ll p = it.first;
			result -= (result / p);
		}
		return result;
}
```

Summations

```
#define numOfDigit(x) 1+(int)(floor(log10(x)))
#define numOfBits(x) 1+(int)(floor(log2(x)))
//return sum of sequence a, a+x , a+2x .... b
ll sumSequence(ll a, ll b, ll x) {
		a = ((a + x - 1) / x) * x;
		b = (b / x) * x;
		return (b + a) * (b - a + x) / (2 * x);
}
ll sumPower(ll x, ll y) { //x^0 + x^1 + x^2 ... x^y
		return (power(x, y + 1) - 1) / (x - 1);
}
// return sum of divisors for all number from 1 to n
//O(n)
ll sumRangeDivisors(int n) {
		ll ans = 0;
		for (int x = 1; x <= n; x++)
			ans += (n / x) * x;
		return ans;
}
// return sum of divisors for all number from 1 to n
// calc 1e9 in 42ms,can calc more but need big integer
ll sumRangeDivisors(ll x) {
		ll ans = 0, left = 1, right;
		for (; left <= x; left = right + 1) {
```

```
			right = x / (x / left);
			ans += (x / left) * (left + right) * (right - left + 1) / 2;
		}
		return ans;
}
```

Misc

Bitmask

```
bool getBit(ll num, int ind) {
	return ((num >> ind) & 1);
}
ll setBit(ll num, int ind, bool val) {
	return val ? (num | (1LL << ind)) : (num & ~(1LL << ind));
}
ll flipBit(ll num, int ind) {
	return (num ^ (1LL << ind));
}
ll leastBit(ll num) {
	return (num & -num);
}
template<class Int>
Int turnOnLastZero(Int num) {
	return num | num + 1;
}
template<class Int>
Int turnOnLastConsecutiveZeroes(Int num) {
	return num | num - 1;
}
template<class Int>
Int turnOffLastBit(Int num) {
	return num & num - 1;
}
template<class Int>
Int turnOffLastConsecutiveBits(Int num) {
	return num & num + 1;
}
//num%mod, mod is a power of 2
ll Mod(ll num, ll mod) {
	return (num & mod - 1);
}
bool isPowerOfTwo(ll num) {
	return (num & num - 1) == 0;
}
void genAllSubmask(int mask) {
	for (int subMask = mask;; subMask = (subMask - 1) & mask) {
		//code
		if (subMask == 0)
			break;
	}
}
/*
 * __builtin functions:
 *  __builtin_popcount -> used to count the number of one's
 *  __builtin_clz -> used to count the leading zeros of the integer
 *  __builtin_ctz -> used to count the trailing zeros of the integer
 *
 */
int LOG2(int x) { //floor(log2(x))
	return 31 - __builtin_clz(x);
}
```

```
int LOG2(long long x) { //floor(log2(x))
	return 63 - __builtin_clzll(x);
}
```

coordinateCompress

```
void coordinateCompress(vector<int>& axes, vector<int>& iToV,
	map<int, int> vToI, int start = 2, int step = 2) {
	for (auto& it : axes)
		vToI[it] = 0;
	iToV.resize(start + step * vToI.size());
	int idx = 0;
	for (auto& it : vToI) {
		it.second = start + step * idx;
		iToV[it.second] = it.first;
		idx++;
	}
}
```

Random numbers

```
//write this line once in top
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count()* ((uint64_t) new char | 1));

// use this instead of rand()
long long rnd = uniform_int_distribution<long long>(low, high)(rng);
```

Custom hash

```
struct custom_hash {
	static uint64_t splitmix64(uint64_t x) {
		x += 0x9e3779b97f4a7c15;
		x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
		x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
		return x ^ (x >> 31);
	}
	// for pair
	size_t operator()(pair<uint64_t, uint64_t> x) const {
		static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
		return splitmix64(x.first + FIXED_RANDOM) ^ (splitmix64(x.second + FIXED_RANDOM) >> 1);
	}
	// for single element
	size_t operator()(uint64_t x) const {
		static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
		return splitmix64(x + FIXED_RANDOM);
	}

};
```

Max histogram area

```
int maxHistogramArea(vector<int> v) {
	stack<int> st;
	int maxArea = 0, area = 0;
	int i = 0;
	while (i < sz(v)) {
		if (st.empty() || v[st.top()] <= v[i])
			st.push(i++);
		else {
			int top = st.top(); st.pop();
			if (st.empty())
				area = v[top] * i;
			else
				area = v[top] * (i - st.top() - 1);
			maxArea = max(maxArea, area);
		}
```

```
	}
	while (!st.empty()) {
		int top = st.top(); st.pop();
		if (st.empty())
			area = v[top] * i;
		else
			area = v[top] * (i - st.top() - 1);
		maxArea = max(maxArea, area);
	}
	return maxArea;
}
```

Sorting

```
long long cnt = 0;
vector<int> v, temp;

// e the first index not have in range array
// like end()
template<class T = less<int>>
void merge_sort(int s, int e, T cmp = less<int>()) {
	if (s + 1 >= e)
		return;
	int m = s + (e - s >> 1);
	merge_sort(s, m, cmp);
	merge_sort(m, e, cmp);
	for (int i = s; i < e; i++)
		temp[i] = v[i];
	int i = s, j = m, k = s;
	while (i < m && j < e)
		if (cmp(temp[i], temp[j]))
			v[k++] = temp[i++];
		else
			v[k++] = temp[j++], cnt += j - k;
	while (i < m)
		v[k++] = temp[i++];
	while (j < e)
		v[k++] = temp[j++];
}

// O(n*log(n)/log(base))
// O(n + base) memory
void radix_sort(vector<int>& v, int base) {
	vector<int> tmp(v.size());
	for (int it = 0, p = 1; it < 10; it++, p *= base) {
		vector<int> frq(base);
		for (auto& it : v)
			frq[(it / p) % base]++;
		for (int i = 1; i < base; i++)
			frq[i] += frq[i - 1];
		for (int i = v.size() - 1; i >= 0; i--)
			tmp[--frq[(v[i] / p) % base]] = v[i];
		v = tmp;
	}
}

void quick_sort(int s, int e) {
	if (s >= e)
		return;
	int j = rand() % (e - s + 1) + s;
	swap(v[s], v[j]);
	j = s;
	int pivot = v[s];
	for (int i = s + 1; i <= e; i++)
```

```
            if (v[i] <= pivot)
                swap(v[i], v[++j]);
        swap(v[s], v[j]);
        quick_sort(s, j - 1);
        quick_sort(j + 1, e);
}
```

LIS binary Search

```
/*
 * without build
 * make upper_bound if can take equal elements
 */
int LIS(const vector<int>& v) {
        vector<int> lis(v.size());//put value less than zero if needed
        int l = 0;
        for (int i = 0; i < sz(v); i++) {
                int idx = lower_bound(lis.begin(), lis.begin() + l, v[i]) - lis.begin();
                if (idx == l)
                        l++;
                lis[idx] = v[i];
        }
        return l;
}

void LIS_binarySearch(vector<int> v) {
        int n = v.size();
        vector<int> last(n), prev(n, -1);
        int length = 0;
        auto BS = [&](int val) {
                int st = 1, ed = length, md, rt = length;
                while (st <= ed) {
                        md = st + ed >> 1;
                        if (v[last[md]] >= val)
                                ed = md - 1, rt = md;
                        else
                                st = md + 1;
                }
                return rt;
        };
        for (int i = 1; i < n; i++) {
                if (v[i] < v[last[0]])
                        last[0] = i;
                else if (v[i] > v[last[length]]) {
                        prev[i] = last[length];
                        last[++length] = i;
                }
                else {
                        int index = BS(v[i]);
                        prev[i] = last[index - 1];
                        last[index] = i;
                }
        }
        cout << length + 1 << "\n";
        vector<int> out;
        for (int i = last[length]; i >= 0; i = prev[i])
                out.push_back(v[i]);
        reverse(out.begin(), out.end());
        for (auto it : out)
                cout << it << endl;
}
```

Mo algorithm

```
int sqrtN; //use a constent value
struct query {
	int l, r, qIdx, block;
	query(int l, int r, int qIdx) :
		l(l), r(r), qIdx(qIdx), block(l / sqrtN) {
	}
	bool operator <(const query& o) const {
		if (block != o.block)
			return block < o.block;
		return (block % 2 == 0 ? r < o.r : r > o.r);
	}
};

int curL, curR, ans;
vector<query> q;
void add(int index);
void remove(int index);
void solve(int l, int r) {
	while (curL > l)
		add(--curL);
	while (curR < r)
		add(++curR);
	while (curL < l)
		remove(curL++);
	while (curR > r)
		remove(curR--);
}

vector<int> MO(int n) {
	vector<int> rt(q.size());
	ans = curL = curR = 0;
	add(0);
	sort(q.begin(), q.end());
	for (auto it : q) {
		solve(it.l, it.r);
		rt[it.qIdx] = ans;
	}
	return rt;
}
```

floyd cycle detection algorithm

```
template<class IntFunction>
pair<int, int> find_cycle_floyd(IntFunction f, int x0) {
	int tortoise = f(x0), hare = f(f(x0));
	while (tortoise != hare) {
		tortoise = f(tortoise);
		hare = f(f(hare));
	}
	int start = 0;
	tortoise = x0;
	while (tortoise != hare) {
		tortoise = f(tortoise);
		hare = f(hare);
		start++;
	}
	int length = 1;
	hare = f(tortoise);
	while (tortoise != hare) {
		hare = f(hare);
		length++;
	}
	return make_pair(start, length);
```