Universidade Federal de Goiás
Team Reference Material

2018 South America/Brazil Regional Contest

Contents

Augmenting Path

```
//Augmenting Path for MCBM
#include <bits/stdc++.h>
using namespace std;

const int N = 1000;
vector<int> g[N];

int augment(int v, vector<int> &match, vector<bool> &visited) {
      if (visited[v]) return 0;
      visited[v] = true;

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

            if (match[u] == -1 || augment(u, match, visited)) {
                  match[u] = v;
                  return 1;
            }
      }

      return 0;
}

int matching(int left_sz, int right_sz) {
      vector<int> match(left_sz + right_sz, -1);
      vector<bool> visited;

      int mcmb = 0;

      for (int i = 0; i < left_sz; i++) {
            visited.assign(left_sz, false);
            mcmb += augment(i, match, visited);
      }

      return mcmb;
}

int main() {
      ios::sync_with_stdio(false);

      return 0;
}
```

Combinations

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

const int N = 120;
int comb[N][N];
int combination(int n, int m) {
   return comb[n][m];
}

void pre_compute(int n) {
   assert(n < N-1);
   for (int i = 0; i <= n; i++) {
      comb[i][0] = comb[i][i] = 1;
      for (int j = 0; j < i; j++) {
         comb[i][j] = comb[i-1][j-1] + comb[i-1][j];
      }
   }
}

int main() {
   ios::sync_with_stdio(false);

   return 0;
}
```

Debug message

```
//Lib - Debug message for CF
#include <bits/stdc++.h>
using namespace std;

template<typename T>
void debug(T value) {
#ifndef ONLINE_JUDGE
   cout << value << endl;
#endif
}

template <typename T, typename... Args>
void debug(T value, Args... args) {
#ifndef ONLINE_JUDGE
   cout << value;
   debug(args...);
#endif
}

int main() {
   debug("this␣message␣do␣not␣will␣appear␣in␣CF.");
   return 0;
}
```

Euler Tour

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

const int N = 100 * 1000 + 10;
vector<int> g[N];
int sz[N], ini[N], cnt = 0;
int64_t path[N];

int dfs(int v, int p) {
   ini[v] = cnt;
   path[cnt] = v;
   cnt++;
   sz[v] = 1;

   for (const auto &u : g[v]) {
      if (u == p) continue;

      sz[v] += dfs(u, v);
   }

   return sz[v];
}

int main() {
   int n;
   cin >> n;

   for (int i = 0, x, y; i < n-1; i++) {
      cin >> x >> y;
      g[x].push_back(y);
      g[y].push_back(x);
   }

   dfs(0, -1);

   for (int i = 0; i < cnt; i++) {
      cout << path[i] << " ";
   }
   cout << endl;

   int v = 1;
   // subarvore de v
   for (int i = ini[v]; i < ini[v]+sz[v]; i++) {
      cout << path[i] << " ";
   }
   cout << endl;

   return 0;
}
```

Extended Euclids

```
// Euclids Algorithm and Modular Multiplicative Inverse
#include <bits/stdc++.h>
using namespace std;

pair<int, int> extendedEuclids(int a, int b) {
   if (b == 0) {
      return {1, 0};
   } else {
      auto p = extendedEuclids(b, a % b);
      return {p.second,
         p.first - floor((double)a / b) * p.second};
   }
}

int inverseMod(int a, int n) {
   auto p = extendedEuclids(a, n);
   return (p.first % n + n) % n;
}

int main() {
   ios::sync_with_stdio(false);

   return 0;
}
```

Fenwick 2D

```
//Lib - Fenwick Tree 2D
#include <bits/stdc++.h>
using namespace std;

const int N = 1010;
int ft[N][N];

int last(int v) {
      return v & -v;
}

void update(int x, int y, int val) {
      for (int i = x; i < N; i += last(i)) {
            for (int j = y; j < N; j += last(j)) {
                  ft[i][j] += val;
            }
      }
}
```

```
int query(int x, int y) {
      int s = 0;
      for (int i = x; i > 0; i -= last(i)) {
            for (int j = y; j > 0; j -= last(j)) {
                  s += ft[i][j];
            }
      }
      return s;
}

int sum(int x1, int y1, int x2, int y2) {
      return query(x2, y2) - query(x2, y1 - 1)
            - query(x1 - 1, y2) + query(x1 - 1, y1 - 1);
}

int main() {
   ios::sync_with_stdio(false);
   return 0;
}
```

Fenwick 2D (Xor)

```
//Lib - Fenwick Tree 2D XOR
#include <bits/stdc++.h>
using namespace std;

const int N = 1010;
int64_t dp[4][N][N];

int parity(int x, int y) {
      int res = 0;
      if (x % 2) res++;
      if (y % 2) res += 2;
      return res;
}

int64_t query(int x, int y) {
      int64_t res = 0;
      x++;
      y++;
      int whichSquare = parity(x, y);
      for (int i = x; i > 0; i -= (i & (-i))) {
            for (int j = y; j > 0; j -= (j & (-j))) {
                  res ^= dp[whichSquare][i][j];
            }
      }
      return res;
}

void update(int x, int y, int64_t val) {
      x++;
      y++;
      int whichSquare = parity(x, y);
      for (int i = x; i <= N; i += (i & (-i))) {
            for (int j = y; j <= N; j += (j & (-j))) {
                  dp[whichSquare][i][j] ^= val;
            }
      }
}

int query2d(int x1, int y1, int x2, int y2) {
   int r = query(x2, y2);
   r ^= query(x2, y1 - 1);
   r ^= query(x1 - 1, y2);
   r ^= query(x1 - 1, y1 - 1);
   return r;
}

void update2d(int x1, int y1, int x2, int y2, int val) {
   update(x1, y1, val);
   update(x1, y2 + 1, val);
   update(x2 + 1, y1, val);
   update(x2 + 1, y2 + 1, val);
}

int main() {
   ios::sync_with_stdio(false);

   int n, m, p;
   int a, b, c, d;
   cin >> n >> m;
   for (int i = 0; i < m; i++) {
      cin >> p >> a >> b >> c >> d;

      update2d(a, b, c, d, p);
   }

   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         cout << query2d(i, j, i, j);
         if (j < n)
            cout << " ";
      }
      cout << endl;
   }
   return 0;
}
```

Fenwick

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

const int N = 2*1e5+10;

int bit[N];

// execute the query [1, x]
int query(int x) {
	int s = 0;

	while (x) {
		s += bit[x];
		x -= (x & -x);
	}

	return s;
}

// execute the update [1,x]
void update(int x, int v) {
	while (x <= N) {
		bit[x] += v;
		x += (x & -x);
	}
}

int main() {
	ios::sync_with_stdio(false);

	return 0;
}
```

Gaussian Elimination

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

#define MAX_N 100

struct AugmentedMatrix {
	double mat[MAX_N][MAX_N + 1];
};
struct ColumnVector {
	double vec[MAX_N];
};

ColumnVector elimination(int n, AugmentedMatrix aug) {
	ColumnVector x;

	for (int j = 0, lgst; j < n-1; j++) {
		// find the largest value of column j
		lgst = j;
		for (int i = j + 1; i < n; i++) {
			if (fabs(aug.mat[i][j]) > fabs(aug.mat[lgst][j]))
				lgst = i;
		}

		// change to first row of elimination the row
		// with largest value
		for (int k = j; k < n; k++) {
			swap(aug.mat[j][k], aug.mat[lgst][k]);
		}

		// elimination phase
		for (int i = j + 1; i < n; i++) {
			for (int k = n; k >= j; k--) {
				aug.mat[i][k] -= aug.mat[i][j] * aug.mat[j][k] / aug.mat[j][j];
			}
		}
	}

	for (int j = n-1; j >= 0; j--) {
		double t = 0.0;
		for (int k = j+1; k < n; k++)
			t += aug.mat[j][k] * x.vec[k]; //back substitution
		x.vec[j] = (aug.mat[j][n] - t) / aug.mat[j][j];
	}

	return x;
}

int main() {
	return 0;
}
```

Generate all combinations

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

void gen_comb(vector<int> a, int r) {
   assert(r <= (int) a.size());

   vector<bool> v(a.size());
   fill(v.begin(), v.begin() + r, true);

   do {
      for (size_t i = 0; i < v.size(); i++) {
         if (v[i]) {
            cout << a[i] << " ";
         }
      }
      cout << "\n";
```

```
   } while (prev_permutation(v.begin(), v.end()));
}

int main() {
   const int N = 50, R = 4;
   vector<int> a(N);
   for (int i = 0; i < N; i++) {
      a[i] = rand();
   }

   gen_comb(a, R);
   return 0;
}
```

Geometry

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

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant; alternative #define PI (2.0 * acos(0.0))

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

double RAD_to_DEG(double r) {
   return r * 180.0 / PI;
}

// struct point_i { int x, y; }; // basic raw form, minimalist mode
struct point_i {
   int x, y; // whenever possible, work with point_i
   point_i() {
      x = y = 0; // default constructor
   }
   point_i(int _x, int _y) : x(_x), y(_y) {}
}; // user-defined

struct point {
   double x, y; // only used if more precision is needed
   point() {
      x = y = 0.0; // default constructor
   }
   point(double _x, double _y) : x(_x), y(_y) {} // user-defined
   bool operator < (point other) const { // override less than operator
      if (fabs(x - other.x) > EPS) // useful for sorting
         return x < other.x; // first criteria , by x-coordinate
      return y < other.y;
   } // second criteria, by y-coordinate
   // use EPS (1e-9) when testing equality of two floating points
   bool operator == (point other) const {
      return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS));
   }
};

double dist(point p1, point p2) { // Euclidean distance
   // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
   return hypot(p1.x - p2.x, p1.y - p2.y);
} // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
   double rad = DEG_to_RAD(theta); // multiply theta with PI / 180.0
   return point(p.x * cos(rad) - p.y * sin(rad),
             p.x * sin(rad) + p.y * cos(rad));
}

struct line {
   double a, b, c;
}; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
   if (fabs(p1.x - p2.x) < EPS) { // vertical line is fine
      l.a = 1.0;
      l.b = 0.0;
      l.c = -p1.x; // default values
   } else {
      l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
      l.b = 1.0; // IMPORTANT: we fix the value of b to 1.0
      l.c = -(double)(l.a * p1.x) - p1.y;
   }
}

// not needed since we will use the more robust form: ax + by + c = 0 (see above)
struct line2 {
   double m, c;
}; // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
   if (abs(p1.x - p2.x) < EPS) { // special case: vertical line
      l.m = INF; // l contains m = INF and c = x_value
      l.c = p1.x; // to denote vertical line x = x_value
      return 0; // we need this return variable to differentiate result
   }
   else {
      l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
      l.c = p1.y - l.m * p1.x;
      return 1; // l contains m and c of the line equation y = mx + c
   }
}

bool areParallel(line l1, line l2) { // check coefficients a & b
   return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
}

bool areSame(line l1, line l2) { // also check coefficient c
   return areParallel(l1,l2) && (fabs(l1.c - l2.c) < EPS);
}

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
```

```
    if (areParallel(l1, l2)) return false; // no intersection
    // solve system of 2 linear algebraic equations with 2 unknowns
    p.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 (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
    else p.y = -(l2.a * p.x + l2.c);
    return true;
}

struct vec {
    double x, y; // name: 'vec' is different from STL vector
    vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(point a, point b) { // convert 2 points to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}

vec scale(vec v, double s) { // nonnegative s = [<1 .. 1 .. >1]
    return vec(v.x * s, v.y * s);
} // shorter.same.longer

point translate(point p, vec v) { // translate p according to v
    return point(p.x + v.x, p.y + v.y);
}

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
    l.a = -m; // always -m
    l.b = 1; // always 1
    l.c = -((l.a * p.x) + (l.b * p.y));
} // compute this

void closestPoint(line l, point p, point &ans) {
    line perpendicular; // perpendicular to l and pass through p
    if (fabs(l.b) < EPS) { // special case 1: vertical line
        ans.x = -(l.c);
        ans.y = p.y;
        return;
    }

    if (fabs(l.a) < EPS) { // special case 2: horizontal line
        ans.x = p.x;
        ans.y = -(l.c);
        return;
    }

    pointSlopeToLine(p, 1 / l.a, perpendicular); // normal line
    // intersect line l with this perpendicular line
    // the intersection point is the closest point
    areIntersect(l, perpendicular, ans);
}

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
    point b;
    closestPoint(l, p, b); // similar to distToLine
    vec v = toVec(p, b); // create a vector
    ans = translate(translate(p, v), v);
} // translate p twice

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

double norm_sq(vec v) {
    return v.x * v.x + v.y * v.y;
}

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
    // formula: c = a + u * ab
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u)); // translate a to c
    return dist(p, c);
} // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) {
        c = point(a.x, a.y); // closer to a
        return dist(p, a);
    } // Euclidean distance between p and a
    if (u > 1.0) {
        c = point(b.x, b.y); // closer to b
        return dist(p, b);
    } // Euclidean distance between p and b
    return distToLine(p, a, b, c);
} // run distToLine as above

double angle(point a, point o, point b) { // returns angle aob in rad
    vec oa = toVec(o, a), ob = toVec(o, b);
    return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

double cross(vec a, vec b) {
    return a.x * b.y - a.y * b.x;
}

//// another variant
//int area2(point p, point q, point r) { // returns 'twice' the area of this triangle A-B-c
// return p.x * q.y - p.y * q.x +
// q.x * r.y - q.y * r.x +
// r.x * p.y - r.y * p.x;
//}

// note: to accept collinear points, we have to change the '> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
    return cross(toVec(p, q), toVec(p, r)) > 0;
}

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
    return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}

// circles
```

```
int insideCircle(point_i p, point_i c, int r) { // all integer version
   int dx = p.x - c.x, dy = p.y - c.y;
   int Euc = dx * dx + dy * dy, rSq = r * r; // all integer
   return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
} //inside/border/outside

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.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
   c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
   return true;
} // to get the other center, reverse p1 and p2

double perimeter(double ab, double bc, double ca) {
   return ab + bc + ca;
}

double perimeter(point a, point b, point c) {
   return dist(a, b) + dist(b, c) + dist(c, a);
}

double area(double ab, double bc, double ca) {
   // Heron's formula, split sqrt(a * b) into sqrt(a) * sqrt(b); in implementation
   double s = 0.5 * perimeter(ab, bc, ca);
   return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca);
}

double area(point a, point b, point c) {
   return area(dist(a, b), dist(b, c), dist(c, a));
}

double rInCircle(double ab, double bc, double ca) {
   return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca));
}

double rInCircle(point a, point b, point c) {
   return rInCircle(dist(a, b), dist(b, c), dist(c, a));
}

// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
   r = rInCircle(p1, p2, p3);
   if (fabs(r) < EPS) return 0; // no inCircle center

   line l1, l2; // compute these two angle bisectors
   double ratio = dist(p1, p2) / dist(p1, p3);
   point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
   pointsToLine(p1, p, l1);

   ratio = dist(p2, p1) / dist(p2, p3);
   p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
   pointsToLine(p2, p, l2);

   areIntersect(l1, l2, ctr); // get their intersection point
   return 1;
}

double rCircumCircle(double ab, double bc, double ca) {
   return ab * bc * ca / (4.0 * area(ab, bc, ca));
}

double rCircumCircle(point a, point b, point c) {
   return rCircumCircle(dist(a, b), dist(b, c), dist(c, a));
}

// assumption: the required points/lines functions have been written
// returns 1 if there is a circumCenter center, returns 0 otherwise
// if this function returns 1, ctr will be the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double &r) {
   double a = p2.x - p1.x, b = p2.y - p1.y;
   double c = p3.x - p1.x, d = p3.y - p1.y;
   double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
   double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
   double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
   if (fabs(g) < EPS) return 0;

   ctr.x = (d*e - b*f) / g;
   ctr.y = (a*f - c*e) / g;
   r = dist(p1, ctr); // r = distance from center to 1 of the 3 points
   return 1;
}

// returns true if point d is inside the circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c, point d) {
return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) +
      (c.y - d.y) * (c.y - d.y)) +
(a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y))
      * (c.x - d.x) +
((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.x - d.x)
      * (c.y - d.y) -
((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.y - d.y)
      * (c.x - d.x) -
(a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y)
      * (c.y - d.y)) -
(a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y))
      * (c.y - d.y) > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b, double c) {
   return (a + b > c) && (a + c > b) && (b + c > a);
}

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
   double result = 0.0;
   for (int i = 0; i < (int)P.size()-1; i++) // remember that P[0] = P[n-1]
      result += dist(P[i], P[i+1]);
   return result;
}

// returns the area, which is half the determinant
double area(const vector<point> &P) {
   double result = 0.0, x1, y1, x2, y2;
   for (int i = 0; i < (int)P.size()-1; i++) {
      x1 = P[i].x;
      x2 = P[i+1].x;
      y1 = P[i].y;
```

```
        y2 = P[i+1].y;
        result += (x1 * y2 - x2 * y1);
    }
    return fabs(result) / 2.0;
}

// returns true if we always make the same turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
    int sz = (int)P.size();
    if (sz <= 3) return false; // a point/sz=2 or a line/sz=3 is not convex
    bool isLeft = ccw(P[0], P[1], P[2]); // remember one result
    for (int i = 1; i < sz-1; i++) // then compare with the others
        if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
            return false; // different sign -> this polygon is concave
    return true;
} // this polygon is convex

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
    if ((int)P.size() == 0) return false;
    double sum = 0; // assume the first vertex is equal to the last vertex
    for (int i = 0; i < (int)P.size()-1; i++) {
        if (ccw(pt, P[i], P[i+1]))
            sum += angle(P[i], pt, P[i+1]); // left turn/ccw
        else sum -= angle(P[i], pt, P[i+1]);
    } // right turn/cw
    return fabs(fabs(sum) - 2*PI) < EPS;
}

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
    double a = B.y - A.y;
    double b = A.x - B.x;
    double c = B.x * A.y - A.x * B.y;
    double u = fabs(a * p.x + b * p.y + c);
    double v = fabs(a * q.x + b * q.y + c);
    return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v));
}

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
    vector<point> P;
    for (int i = 0; i < (int)Q.size(); i++) {
        double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
        if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
        if (left1 > -EPS) P.push_back(Q[i]); // Q[i] is on the left of ab
        if (left1 * left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line ab
            P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
    }
    if (!P.empty() && !(P.back() == P.front()))
        P.push_back(P.front()); // make P's first point = P's last point
    return P;
}

point pivot;
bool angleCmp(point a, point b) { // angle-sorting function
    if (collinear(pivot, a, b)) // special case
        return dist(pivot, a) < dist(pivot, b); // check which one is closer
    double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
    double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
    return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
} // compare two angles

vector<point> CH(vector<point> P) { // the content of P may be reshuffled
    int i, j, n = (int)P.size();
    if (n <= 3) {
        if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
        return P; // special case, the CH is P itself
    }

    // first, find P0 = point with lowest Y and if tie: rightmost X
    int P0 = 0;
    for (i = 1; i < n; i++)
        if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
            P0 = i;

    point temp = P[0];
    P[0] = P[P0];
    P[P0] = temp; // swap P[P0] with P[0]

    // second, sort points by angle w.r.t. pivot P0
    pivot = P[0]; // use this global variable as reference
    sort(++P.begin(), P.end(), angleCmp); // we do not sort P[0]

    // third, the ccw tests
    vector<point> S;
    S.push_back(P[n-1]);
    S.push_back(P[0]);
    S.push_back(P[1]); // initial S
    i = 2; // then, we check the rest
    while (i < n) { // note: N must be >= 3 for this method to work
        j = (int)S.size()-1;
        if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]); // left turn, accept
        else S.pop_back();
    } // or pop the top of S until we have a left turn
    return S;
} // return the result

int main() {
    return 0;
}
```

Getchar unlocked

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

bool readChar(char &c) {
        c = getchar_unlocked();
        return c != EOF;
}

inline void writeChar(char c) {
        putchar_unlocked(c);
```

```
}

template<typename T>
bool readInt( T &n ) {
   n = 0;
   register bool neg = false;
   register char c = getchar_unlocked();
   if( c == EOF) { n = -1; return false; }
   while (!('0' <= c && c <= '9')) {
      if( c == '-' ) neg = true;
      c = getchar_unlocked();
   }
   while ('0' <= c && c <= '9') {
      n = n * 10 + c - '0';
      c = getchar_unlocked();
   }
   n = (neg ? (-n) : (n));
   return true;
}

template<typename T>
inline void writeInt(T n){
   register int idx = 20;
   if( n < 0 ) putchar_unlocked('-');
   n = abs(n);
   char out[21];
   out[20] = ' ';
   do{
      idx--;
      out[idx] = n % 10 + '0';
      n/= 10;
   }while(n);
   do{ putchar_unlocked(out[idx++]); } while (out[idx] != ' ');
}

int main() {
      ios::sync_with_stdio(false);

      int x;
      while (readInt(x)) {
            writeInt(x);
      }
      return 0;
}
```

Kruskal

```
// UVA - Bond - 11354
// similar to problem "Caminhoes"
// Kruskal and queries response inside kruskal
#include <bits/stdc++.h>
using namespace std;

namespace UF {
      const int N = 100 * 1000 + 10;
      int parent[N];
      int sz[N];
      void init(int size) {
         assert(size < N);
            for (int i = 0; i < size; i++) {
                  parent[i] = i;
                  sz[i] = 1;
      }
      }

      int find(int x) {
            if (parent[x] == x) return x;
            else return parent[x] = find(parent[x]);
      }

      void join(int x, int y) {
            x = find(x);
            y = find(y);

            if (x == y) return;

      if (sz[x] < sz[y]) {
         parent[x] = y;
         sz[y] += sz[x];
      } else {
         parent[y] = x;
            sz[x] += sz[y];
      }
      }
}

struct edge {
   int x, y;
   int64_t w;
   bool operator< (const edge &other) {
      return this->w < other.w || (this->w == other.w && this->x < other.x);
   }
};

struct query {
   int x, y, ans = -1;
};

const int N = 100 * 1000 + 10;
edge edges[N];
query queries[N];
vector<int> comp_queries[N];

int main() {
      ios::sync_with_stdio(false);
      cin.tie(nullptr);

   int n, m, x, y, q;
   int64_t w;
   bool first = true;

   while (cin >> n >> m) {
      if (!first) {
         cout << "\n";
      }
```

```
      first = false;

      for (int i = 0; i < m; i++) {
         cin >> edges[i].x >> edges[i].y >> edges[i].w;
      }

      for (int i = 1; i <= n; i++) {
         comp_queries[i].clear();
      }

      cin >> q;

      for (int i = 0; i < q; i++) {
         cin >> x >> y;
         queries[i].x = x;
         queries[i].y = y;
         queries[i].ans = -1;

         comp_queries[x].push_back(i);
         comp_queries[y].push_back(i);
      }

      sort(edges, edges+m);

      UF::init(n+1);

      for (int i = 0; i < m; i++) {
         x = edges[i].x;
         y = edges[i].y;
         w = edges[i].w;

         if (UF::find(x) != UF::find(y)) {
            int small = UF::find(x),
               large = UF::find(y);

            if (UF::sz[small] >= UF::sz[large]) {
               swap(small, large);
            }

            // note: should choose small and large in the same
            // manner that union-find do

            UF::join(x, y);

            for (const auto &id : comp_queries[small]) {
               if (queries[id].ans == -1 &&
                     UF::find(queries[id].x) == UF::find(queries[id].y)) {
                  queries[id].ans = w;
               } else {
                  comp_queries[large].push_back(id);
               }
            }

            // store queries in large will maintain
            // access to queries, because UF::find return the large

            comp_queries[small].clear();
         }
      }

      for (int i = 0; i < q; i++) {
         cout << queries[i].ans << "\n";
      }
   }

      return 0;
}
```

LCA

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

const int N = 100100, L = 20;
vector<pair<int, int64_t>> g[N];
int64_t dist[N];
int parent[N][L], lvl[N];

void dfs(int v, int p) {
      parent[v][0] = p;
      lvl[v] = lvl[p] + 1;

      int64_t w;
      for (int j = 0, u; j < (int)g[v].size(); j++) {
            u = g[v][j].first;
            w = g[v][j].second;

            if (u == p) continue;

            dist[u] = dist[v] + w;
            dfs(u, v);
      }
}

void pre_lca(int n) {
      for (int i = 1; i < L; i++) {
            for (int v = 0; v < n; v++) {
                  parent[v][i] = parent[parent[v][i-1]][i-1];
            }
      }
}

int lca(int a, int b) {
      if (lvl[a] < lvl[b]) {
            swap(a, b);
      }

      int d = lvl[a] - lvl[b];

      for (int i = L-1; i >= 0; i--) {
            if (d & (1 << i)) {
                  a = parent[a][i];
            }
      }
```

```
    if (a == b) return a;

    for (int i = L-1; i >= 0; i--) {
        if (parent[a][i] != parent[b][i]) {
            a = parent[a][i];
            b = parent[b][i];
        }
    }

    return parent[a][0];
}

int main() {
    ios::sync_with_stdio(false);

    int n;
    cin >> n;
    dfs(0, 0);
    pre_lca(n);

    return 0;
}
```

LIS

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

int main() {
    ios::sync_with_stdio(false);
    const int N = 1000;
  int n;
  int64_t a[N];

  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }

    set<int64_t> st;
    auto it = st.begin();
  for (int i = 0; i < n; i++) {
        if (st.find(a[i]) != st.end()) continue;

        st.insert(a[i]);
        it = st.find(a[i]);
        if (next(it) != st.end())
            st.erase(next(it));
    }

    cout << st.size() << endl;

    return 0;
}
```

LIS (Print elements)

```
// LIS algorithm with function to print the sequence
// UVA - What Goes Up - 481
#include <bits/stdc++.h>
using namespace std;

namespace LIS {
  const int N = 100 * 1000 + 10;
  int parent[N], l[N], l_id[N], l_end;
  void lis(int a[], const int n) {
    l_end = 0;
    for (int i = 0; i < n; i++) {
      int pos = lower_bound(l, l+l_end, a[i]) - l;

      l[pos] = a[i];
      l_id[pos] = i;
      parent[i] = pos ? l_id[pos-1] : -1;

      if (pos == l_end) {
        l_end++;
      }
    }
  }

  vector<int> get_lis(int a[]) {
    stack<int> st;
    for (int x = l_id[l_end-1]; x != -1; x = parent[x]) {
      st.push(a[x]);
    }

    vector<int> ans;
    while (!st.empty()) {
      ans.push_back(st.top());
      st.pop();
    }
    return ans;
  }
}

int main() {
  const int N = 100 * 1000 + 10;
  int x, n = 0;
  int a[N];

  while (cin >> x) {
    a[n++] = x;
  }

  LIS::lis(a, n);
```

```
   vector<int> ans = LIS::get_lis(a);
   cout << ans.size() << "\n-\n";
   for (const auto &x : ans) {
      cout << x << "\n";
   }

   return 0;
}
```

LIS with Fenwick

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

const int M = 310;
int ft[M];

void update(int x, int v) {
   x += 3;
   while (x < M) {
      ft[x] = max(ft[x], v);
      x += (x & -x);
   }
}

int query(int x) {
   x += 3;
   int ans = 0;
   while (x > 0) {
      ans = max(ans, ft[x]);
      x -= (x & -x);
   }
   return ans;
}

int dp[1000100];

int main() {
   const int N = 110;
   int n;
   int64_t a[N];

   cin >> n;
   for (int i = 0; i < n; i++) {
      cin >> a[i];
   }

   int lis = 1;

   for (int i = 0; i < n; i++) {
      dp[i] = query(a[i%n]-1) + 1;
      update(a[i%n], dp[i]);
      lis = max(lis, dp[i]);
   }

   cout << lis << endl;
   return 0;
}
```

Matrix Power

```
// Codeforces - Gym 101845 - Univ. Nacional de Colombia PC
// Apple Trees
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 5;

struct matrix {
   int64_t m[MAXN][MAXN];
};

matrix mult(matrix a, matrix b, int64_t mod) {
   matrix ans;
   for (int i = 0; i < MAXN; i++) {
      for (int j = 0; j < MAXN; j++) {
         ans.m[i][j] = 0;
         for (int k = 0; k < MAXN; k++) {
            ans.m[i][j] += (a.m[i][k] * b.m[k][j]) % mod;
         }
         ans.m[i][j] %= mod;
      }
   }

   return ans;
}

matrix power(matrix base, int64_t exp, int64_t mod) {
   matrix ans;
   for (int i = 0; i < MAXN; i++) {
      for (int j = 0; j < MAXN; j++) {
         ans.m[i][j] = (i == j) ? 1 : 0;
         base.m[i][j] %= mod;
      }
   }

   while (exp) {
      if (exp & 1) {
         ans = mult(ans, base, mod);
      }

      base = mult(base, base, mod);
```

```
      exp >>= 1;
   }

   return ans;
}

int main() {
   const int64_t mod = 1000000007;
   int64_t n;

   cin >> n;

   if (n < 10) {
      cout << "1\n";
      return 0;
   }

   matrix a = {{{16, 9, 4, 1, 0},
             {1, 0, 0, 0, 0},
             {0, 1, 0, 0, 0},
             {0, 0, 1, 0, 0},
             {0, 0, 0, 1, 0} }};

   a = power(a, n/10, mod);

   int64_t ans = 0;
   for (int i = 0; i < 4; i++) {
      ans += a.m[i][0];
      ans %= mod;
   }

   if (n % 10 < 5) {
      ans += a.m[4][0];
      ans %= mod;
   }

   cout << ans << "\n";
   return 0;
}
```

Max Flow

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

const int INF = numeric_limits<int>::max();
const int64_t LINF = 1000 * 1000 * 1000LL * 1000 * 1000 * 1000LL;
const int V = 110; // number of vertex

// augment - walk in augmented path and update flow
// adj - adj matrix, p - parent of vertex i, t - sink
int64_t augment(int64_t adj[][V], int p[], int t) {
      int u = t;
      int64_t minimum = LINF;

      // find minimum flow in augmented path
      while (p[u] != -1) {
            minimum = min(adj[p[u]][u], minimum);
            u = p[u];
      }

      // walk in augment path updating flow
      u = t;
      while (p[u] != -1) {
            adj[p[u]][u] -= minimum;
            adj[u][p[u]] += minimum;
            u = p[u];
      }

      return minimum; // return minimum flow in augmented path
}

// s - source, t - sink, n - number of vertex
int64_t edmonds_karp(int64_t adj[][V], const int s, const int t, const int n) {

      int64_t mf = 0, // max flow answer
                  f = 1;
      int64_t dist[V];
      int p[V];
      int u;

      while (f > 0) {
            f = 0;
            for (int i = 0; i < n; i++)
               dist[i] = LINF, p[i] = -1;
            dist[s] = 0;

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

            while (!q.empty()) {
                  u = q.front();
                  q.pop();

                  // stop if reach sink t
                  if (u == t) break;

                  for (int v = 0; v < n; v++) {
                        if (adj[u][v] > 0 && dist[v] == LINF) {
                              dist[v] = dist[u] + 1;
                              q.push(v);
                              p[v] = u;
                        }
                  }
            }

            // verify if bfs stop when reach sink t
            if (u == t) {

                  // find minimum flow in augmented path
                  f = augment(adj, p, t);

                  // update max flow of network
                  mf += f;
            }
      }
```

```
    return mf;
}

int main() {
    ios::sync_with_stdio(false);

    int n, m, k;
    int a, b, source, sink;
    int64_t adj[V][V], c;

    k = 1;
    while (cin >> n, n) {
        // cleaning adj matrix
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                adj[i][j] = 0;

        cin >> source >> sink >> m;
        source--;
        sink--;

        for (int i = 0; i < m; i++) {
            cin >> a >> b >> c;
            a--;
            b--;
            adj[a][b] += c;
            adj[b][a] += c;
        }

        c = edmonds_karp(adj, source, sink, n);
        cout << "Network " << k << "\nThe bandwidth is " << c << ".\n\n";
        k++;
    }

    return 0;
}
```

Min Cost Max Flow

```
// Min cost max flow
// UVA 10594 - Data Flow

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

const int INF = numeric_limits<int>::max();
const int64_t LINF = 1000LL * 1000 * 1000 * 1000 * 1000 * 1000LL;
const int N = 150; // number of vertex
const int M = 4*5002; // number of edges

struct edge {
   // v - from vertex
   // u - to vertex
   int v, u, next;
   int64_t cap, cost;
} edges[M];

int first[N], edgenum = 0;

// initialize algorithm structs
void init(int sz = N) {
   for (int i = 0; i < sz; i++) {
      first[i] = -1;
   }
   edgenum = 0;
}

// add a directed edge v -> u and residual edge v <- u
void add_edge(int v, int u, int64_t cap, int64_t cost) {
   edge &e = edges[edgenum];
   e.v = v;
   e.u = u;
   e.cap = cap;
   e.cost = cost;
   e.next = first[v];
   first[v] = edgenum;
   edgenum++;

   // residual edge
   edge &e2 = edges[edgenum];
   e2.v = u;
   e2.u = v;
   e2.cap = 0;
   e2.cost = -cost;
   e2.next = first[u];
   first[u] = edgenum;
   edgenum++;
}

int64_t dist[N];
int in_queue[N], p[N];

// augment - walk in augmented path and update flow
// u - sink or final of path
int64_t augment(int u) {
      int64_t minimum = LINF;

      // find minimum flow in augmented path
   for (int k = p[u]; k != -1; k = p[edges[k].v]) {
      minimum = min(edges[k].cap, minimum);
      }

      // walk in augment path updating flow
   for (int k = p[u]; k != -1; k = p[edges[k].v]) {
      // forward edge
      edges[k].cap -= minimum;
      // residual edge
      edges[k^1].cap += minimum;
      }

      return minimum; // return minimum flow in augmented path
}

// s - source, t - sink, n - number of vertex [0, ..., n-1]
pair<int64_t, int64_t> mcmf(const int s, const int t, const int n) {
```

```
    int64_t mf = 0, // max flow answer
          min_cost = 0, // min cost answer
                f = 1; // current min cost

    while (f > 0) {
          f = 0;
          for (int i = 0; i < n; i++) {
             dist[i] = LINF;
             p[i] = -1;
             in_queue[i] = 0;
    }

          dist[s] = 0;

          queue<int> q;
          q.push(s);
    in_queue[s] = 1;
          while (!q.empty()) {
                int u = q.front();
                q.pop();
                in_queue[u] = 0;

       for (int k = first[u], v; k != -1; k = edges[k].next) {
          v = edges[k].u;
                      if (edges[k].cap > 0 && dist[v] > dist[u] + edges[k].cost) {
                            dist[v] = dist[u] + edges[k].cost;
                            p[v] = k;
                            if (in_queue[v] == 0) {
                               q.push(v);
                               in_queue[v] = 1;
                            }
                      }
                }

          }

          // verify if bfs stop when reach sink t
    if (dist[t] != LINF) {
                // find minimum flow in augmented path
                f = augment(t);

                // update max flow of network
                min_cost += f * dist[t];
                mf += f;
          }
          }

          return {mf, min_cost};
}


int main() {
   int a[M], b[M];
   int64_t c[M];
      int n, m;
      int64_t d, k;
      int source, sink;

      while (cin >> n >> m) {
      init(n+10);

            for (int i = 0; i < m; i++) {
               cin >> a[i] >> b[i] >> c[i];
            }

      cin >> d >> k;

            for (int i = 0; i < m; i++) {
               add_edge(a[i], b[i], k, c[i]);
               add_edge(b[i], a[i], k, c[i]);
      }

      source = 0;
      sink = n;

      add_edge(source, 1, d, 0);

            auto p = mcmf(source, sink, n+1);

            if (p.first != d) {
               printf("Impossible.\n");
      } else {
         printf("%jd\n", p.second);
      }
      }

      return 0;
}
```

Mo's Algorithm

```
// Mo's Algorithm
// Problem H - Wine Production
#include <bits/stdc++.h>
using namespace std;

// Variables, that hold current "state" of computation
long long current_answer;

// Array to store answers (because the order we achieve them is messed up)
const int N = 100500;
long long answers[N];
int BLOCK_SIZE;
int arr[N];

// We will represent each query as three numbers: L, R, idx. Idx is
// the position (in original order) of this query.
pair< pair<int, int>, int> queries[N];

unordered_map<int, int > cnt, caras;
map<int, int> ok;

// Essential part of Mo's algorithm: comparator, which we will
// use with std::sort. It is a function, which must return True
// if query x must come earlier than query y, and False otherwise.
```

```
inline bool mo_cmp(const pair< pair<int, int>, int> &x,
      const pair< pair<int, int>, int> &y) {
   int block_x = x.first.first / BLOCK_SIZE;
   int block_y = y.first.first / BLOCK_SIZE;
   if(block_x != block_y)
      return block_x < block_y;
   return x.first.second < y.first.second;
}

// When adding a number, we first nullify it's effect on current
// answer, then update cnt array, then account for it's effect again.
inline void add(int pos) {
   cnt[arr[pos]]++;
   int q = cnt[arr[pos]];
   caras[q]++;

   if (caras[q] >= q) {
      ok[q]++;
   }
   if(!ok.empty()){
      current_answer = (*(ok.end()--)).first;
   }
   else current_answer = 1;
}

// Removing is much like adding.
inline void remove(int pos) {
   int q = cnt[arr[pos]];
   cnt[arr[pos]]--;
   caras[q]--;

   if (ok.count(q)){
      ok[q]--;
      if (ok[q] == 0) {
         ok.erase(q);
      }
   }
   if(!ok.empty()){
      current_answer = (*(ok.end()--)).first;
   }
   else current_answer = 1;
}

int main() {
   cin.sync_with_stdio(false);
   cin.tie(NULL);

   int n, q;
   cin >> n >> q;
   BLOCK_SIZE = static_cast<int>(sqrt(n));

   for(int i = 0; i < n; i++)
      cin >> arr[i];

   for(int i = 0; i < q; i++) {
      cin >> queries[i].first.first >> queries[i].first.second;
      queries[i].first.first--;
      queries[i].first.second--;

      queries[i].second = i;
   }

   // Sort queries using Mo's special comparator we defined.
   sort(queries, queries + q, mo_cmp);

   // Set up current segment [mo_left, mo_right].
   int mo_left = 0, mo_right = -1;

   for(int i = 0; i < q; i++) {
      // [left, right] is what query we must answer now.
      int left = queries[i].first.first;
      int right = queries[i].first.second;

      // Usual part of applying Mo's algorithm: moving mo_left
      // and mo_right.
      while(mo_right < right) {
         mo_right++;
         add(mo_right);
      }
      while(mo_right > right) {
         remove(mo_right);
         mo_right--;
      }

      while(mo_left < left) {
         remove(mo_left);
         mo_left++;
      }
      while(mo_left > left) {
         mo_left--;
         add(mo_left);
      }

      // Store the answer into required position.
      answers[queries[i].second] = current_answer;
   }

   // We output answers *after* we process all queries.
   for(int i = 0; i < q; i++)
      cout << answers[i] << "\n";
   return 0;
}
```

Random Numbers

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <vector>

using namespace std;

const int N = 3000000;

double average_distance(const vector<int> &permutation) {
```

```
   double distance_sum = 0;

   for (int i = 0; i < N; i++)
      distance_sum += abs(permutation[i] - i);

   return distance_sum / N;
}

int main() {
   mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
   vector<int> permutation(N);

   for (int i = 0; i < N; i++)
      permutation[i] = i;

   shuffle(permutation.begin(), permutation.end(), rng);
   cout << average_distance(permutation) << '\n';

   for (int i = 0; i < N; i++)
      permutation[i] = i;

   for (int i = 1; i < N; i++)
      swap(permutation[i], permutation[uniform_int_distribution<int>(0, i)(rng)]);

   cout << average_distance(permutation) << '\n';
}
```

Sieve

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

const int M = 10000010;
bool notprime[M];
void sieve(vector<int> &primes, int n) {
   assert(n < M);
   notprime[0] = notprime[1] = true;
   for (int64_t i = 2; i <= n; i++) {
      if (!notprime[i]) {
         primes.push_back(i);
         for (int64_t j = i * i; j <= n; j += i) {
            notprime[j] = true;
         }
      }
   }
}

bool isPrime(vector<int> &primes, int64_t n) {
   if (n < M) return !notprime[n];

   for (const auto &p : primes) {
      if (n % p == 0)
         return false;
   }

   return true;
}

int main() {
   int n;
   vector<int> p;

   cin >> n;

   sieve(p, n);

   return 0;
}
```

Suffix Automata

```
// Suffix Automaton
// LCS problem
#include <bits/stdc++.h>
using namespace std;

struct SuffixAutomaton {
   vector<map<char, int>> edges; // edges[i] : edges of state i
   vector<int> link; // suffix link of state i
   vector<int> lenght; // lenght of longest string in class of state i
   int last, sz;
   vector<bool> term;

   SuffixAutomaton() {
      // add the initial node
      edges.push_back(map<char, int>());
      link.push_back(-1); // suffix link of first state is dummy
      lenght.push_back(0);
      term.push_back(false);
      last = 0; // initiate with index of first state
      sz = 0; // lenght of longest suffix added to automaton
   }

   SuffixAutomaton(string s) : SuffixAutomaton() {
      for (const auto &ch : s) {
         extend(ch);
      }
      find_terminals();
   }

   void extend(char ch) {
      // create new state for new equivalence (end-points) class
      edges.push_back(map<char, int>());
      sz++;
      lenght.push_back(sz);
```

```
        link.push_back(0);
        term.push_back(false);

        int r = edges.size() - 1;
        int p = last;

        while (p >= 0 && edges[p].find(ch) == edges[p].end()) {
            edges[p][ch] = r;
            p = link[p];
        }

        if (p != -1) {
            int q = edges[p][ch];
            if (lenght[p] + 1 == lenght[q]) {
                link[r] = q;
            } else {
                edges.push_back(edges[q]);
                lenght.push_back(lenght[p] + 1);
                link.push_back(link[q]);
                term.push_back(false);
                int qq = edges.size() - 1;
                link[q] = qq;
                link[r] = qq;

                while (p >= 0 && edges[p][ch] == q) {
                    edges[p][ch] = qq;
                    p = link[p];
                }
            }
        }

        last = r;
    }

    void find_terminals() {
        term = vector<bool>(edges.size(), false);
        int p = last;
        while (p >= 0) {
            term[p] = true;
            p = link[p];
        }
    }
};

int64_t lcs(SuffixAutomaton &sa, string t) {
    int st = 0, len = 0, best = 0, best_i = 0;

    for (int i = 0, sz = t.size(); i < sz; i++) {
        char ch = t[i];

        while (st != 0 && sa.edges[st].count(ch) == 0) {
            st = sa.link[st];
            len = sa.lenght[st];
        }

        if (sa.edges[st].count(ch) > 0) {
            st = sa.edges[st][ch];
            len++;
        }

        if (best < len) {
            best = len;
            best_i = i;
        }
    }

    cout << t.substr(best_i-best+1, best) << endl;

    return best;
}

void dfs(SuffixAutomaton &sa, vector<int64_t> &occur, vector<int64_t> &words, int st) {
    if (occur[st] > 0)
        return;

    int64_t occ = 0, wrd = 0;

    if (sa.term[st]) {
        occ++;
        wrd++;
    }

    for (const auto &p : sa.edges[st]) {
        dfs(sa, occur, words, p.second);
        occ += occur[p.second];
        wrd += words[p.second] + occur[p.second];
    }

    occur[st] = occ;
    words[st] = wrd;
}

string find_kth_substr(SuffixAutomaton &sa, int k) {
    vector<int64_t> occur(sa.edges.size()), words(sa.edges.size());
    dfs(sa, occur, words, 0);

    // find kth substring

    int st = 0;
    string t = "";

    int64_t prev_k = k;
    while (k > 0) {
        int64_t acc = 0, tmp;

        for (const auto &p : sa.edges[st]) {
            tmp = acc;
            acc += words[p.second];
            if (acc >= k) {
                st = p.second;
                k -= tmp + occur[p.second];
                t += p.first;
                break;
            }
        }

        if (k == prev_k) {
            t = "No such line.";
            break;
        }

        prev_k = k;
    }

    return t;
}
```

```
int main() {
   ios::sync_with_stdio(false);
   string s, t;

   cin >> s >> t;

   SuffixAutomaton am(s);
   cout << lcs(am, t) << "\n";

   // find kth substring in lexicographical order
   cout << find_kth_substr(am, 1) << endl;
   return 0;
}
```

Applications

Problem: Find whether a given string w is a substring of s.

Solution: Simply run the automaton.

```
SuffixAutomaton a(s);
bool fail = false;
int n = 0;
for(int i=0;i<w.size();i++) {
  if(a.edges[n].find(w[i]) == a.edges[n].end()) {
    fail = true;
    break;
  }
  n = a.edges[n][w[i]];
}
if(!fail) cout << w << " is a substring of " << s << "\n";
```

Problem: Find whether a given string w is a suffix of s.

Solution: Construct the list of terminal states, run the automaton as above and check in the end if the n is among the terminal states.

Let's now look at the dp problems.

Problem: Count the number of distinct substrings in s.

Solution: The number of distinct substrings is the number of different paths in the automaton. These can be calculated recursively by calculating for each node the number of different paths starting from that node. The number of different paths starting from a node is the sum of the corresponding numbers of its direct successors, plus 1 corresponding to the path that does not leave the node.

Problem: Count the number of times a given word w occurs in s.

Solution: Similar to the previous problem. Let p be the node in the automaton that we end up while running it for w. This time the number of times a given word occurs is the number of paths starting from p and ending in a terminal node, so one can calculate recursively the number of paths from each node ending in a terminal node.

Problem: Find where a given word w occurs for the first time in s.

Solution: This is equivalent to calculating the longest path in the automaton after reaching the node p (defined as in the previous solution).

Finally let's consider the following problem where the suffix links come handy.

Problem: Find all the positions where a given word w occurs in s.

Solution: Prepend the string with some symbol '$' that does not occur in the string and construct the suffix automaton. Let's then add to each node of the suffix automaton its children in the suffix tree:

```
children=vector<vector<int>>(link.size());
for(int i=0;i<link.size();i++) {
  if(link[i] >= 0) children[link[i]].push_back(i);
```

```
}
```

Now find the node p corresponding to the node w as has been done in the previous problems. We can then dfs through the subtree of the suffix tree rooted at p by using the children vector. Once we reach a leaf, we know that we have found a prefix of s that ends in w, and the length of the leaf can be used to calculate the position of w. All of the dfs branches correspond to different prefixes, so no unnecessary work is done and the complexity is $O(|s| + |w| + sizeofoutput)$.

Union Find

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

// versao com namespace
namespace UF {
      const int N = 100 * 1000 + 10;
      int parent[N];
      int sz[N];
      void init(int size) {
         assert(size < N);
            for (int i = 0; i < size; i++) {
                  parent[i] = i;
                  sz[i] = 1;
      }
      }
      int find(int x) {
            if (parent[x] == x) return x;
            else return parent[x] = find(parent[x]);
      }
      void join(int x, int y) {
            x = find(x);
            y = find(y);

            if (x == y) return;

      if (sz[x] < sz[y]) {
         parent[x] = y;
         sz[y] += sz[x];
      } else {
         parent[y] = x;
         sz[x] += sz[y];
      }
      }
}

// versao usando um struct
const int N = 100 * 1000 + 10;
struct union_find {
   int parent[N];
   int sz[N];

   union_find() {
   }

   void init(int size) {
      assert(size < N);
      for (int i = 0; i < size; i++) {
         parent[i] = i;
         sz[i] = 1;
      }
   }

   int find(int x) {
      if (parent[x] == x)
         return x;
      else
         return parent[x] = find(parent[x]);
   }

   void join(int x, int y) {
         x = find(x);
         y = find(y);

         if (x == y) return;

      if (sz[x] < sz[y]) {
         parent[x] = y;
         sz[y] += sz[x];
      } else {
         parent[y] = x;
         sz[x] += sz[y];
      }
   }
};

int main() {
      ios::sync_with_stdio(false);

      int n;
      cin >> n;

   UF::init(n);

   union_find uf;
   uf.init(n);

      return 0;
}
```