**University of Cambridge – Slope Party**

**//----Convex Hull, give c-clockwise, with bottomleft first. nlogn**

#define REMOVE\_REDUNDANT

typedef double T;

const T EPS = 1e-7;

struct PT {

T x, y;

PT() {}

PT(T x, T y) : x(x), y(y) {}

bool operator<(const PT &rhs) const { return make\_pair(y,x) < make\_pair(rhs.y,rhs.x); }

bool operator==(const PT &rhs) const { return make\_pair(y,x) == make\_pair(rhs.y,rhs.x); }

};

T cross(PT p, PT q) { return p.x\*q.y-p.y\*q.x; }

T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE\_REDUNDANT

bool between(const PT &a, const PT &b, const PT &c) {

return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)\*(c.x-b.x) <= 0 && (a.y-b.y)\*(c.y-b.y) <= 0);

}

#endif

void ConvexHull(vector<PT> &pts) {

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

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

vector<PT> up, dn;

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

while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop\_back();

while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop\_back();

up.push\_back(pts[i]);

dn.push\_back(pts[i]);

}

pts = dn;

for (int i = (int) up.size() - 2; i >= 1; i--) pts.push\_back(up[i]);

#ifdef REMOVE\_REDUNDANT

if (pts.size() <= 2) return;

dn.clear();

dn.push\_back(pts[0]);

dn.push\_back(pts[1]);

for (int i = 2; i < pts.size(); i++) {

if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop\_back();

dn.push\_back(pts[i]);

}

if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {

dn[0] = dn.back();

dn.pop\_back();

}

pts = dn;

#endif

}

**//----Modular Arithmetic**

typedef vector<int> VI;

typedef pair<int, int> PII;

inline int mod(int a, int b) {

return ((a%b) + b) % b;

}

int gcd(int a, int b) {

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

return a;

}

int lcm(int a, int b) {

return a / gcd(a, b)\*b;

}

int powermod(int a, int b, int m)

{

int ret = 1;

while (b)

{

if (b & 1) ret = mod(ret\*a, m);

a = mod(a\*a, m);

b >>= 1;

}

return ret;

}

**// returns g = gcd(a, b), finds x, y such that d = ax + by**

int extended\_euclid(int a, int b, int &x, int &y) {

int xx = y = 0;

int yy = x = 1;

while (b) {

int q = a / b;

int t = b; b = a%b; a = t;

t = xx; xx = x - q\*xx; x = t;

t = yy; yy = y - q\*yy; y = t;

}

return a;

}

**// finds all solutions to ax = b (mod n)**

VI modular\_linear\_equation\_solver(int a, int b, int n) {

int x, y;

VI ret;

int g = extended\_euclid(a, n, x, y);

if (!(b%g)) {

x = mod(x\*(b / g), n);

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

ret.push\_back(mod(x + i\*(n / g), n));

}

return ret;

}

**//find modular inverse, -1 if does not exist**

int mod\_inverse(int a, int n) {

int x, y;

int g = extended\_euclid(a, n, x, y);

if (g > 1) return -1;

return mod(x, n);

}

**// find z such that z % m1 = r1, z % m2 = r2 mod lcm(m1,m2).**

PII chinese\_remainder\_theorem(int m1, int r1, int m2, int r2) {

int s, t;

int g = extended\_euclid(m1, m2, s, t);

if (r1%g != r2%g) return make\_pair(0, -1);

return make\_pair(mod(s\*r2\*m1 + t\*r1\*m2, m1\*m2) / g, m1\*m2 / g);

}

**// CMT for the genral case**

PII chinese\_remainder\_theorem(const VI &m, const VI &r) {

PII ret = make\_pair(r[0], m[0]);

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

ret = chinese\_remainder\_theorem(ret.second, ret.first, m[i], r[i]);

if (ret.second == -1) break;

}

return ret;

}

**// determines if there are x and y such that ax + by = c**

bool linear\_diophantine(int a, int b, int c, int &x, int &y) {

if (!a && !b)

{

if (c) return false;

x = 0; y = 0;

return true;

}

if (!a)

{

if (c % b) return false;

x = 0; y = c / b;

return true;

}

if (!b)

{

if (c % a) return false;

x = c / a; y = 0;

return true;

}

int g = gcd(a, b);

if (c % g) return false;

x = c / g \* mod\_inverse(a / g, b / g);

y = (c - a\*x) / b;

return true;

}

**//----fft**

struct cpx

{

cpx(){}

cpx(double aa):a(aa),b(0){}

cpx(double aa, double bb):a(aa),b(bb){}

double a;

double b;

double modsq(void) const

{

return a \* a + b \* b;

}

cpx bar(void) const

{

return cpx(a, -b);

}

};

cpx operator +(cpx a, cpx b)

{

return cpx(a.a + b.a, a.b + b.b);

}

cpx operator \*(cpx a, cpx b)

{

return cpx(a.a \* b.a - a.b \* b.b, a.a \* b.b + a.b \* b.a);

}

cpx operator /(cpx a, cpx b)

{

cpx r = a \* b.bar();

return cpx(r.a / b.modsq(), r.b / b.modsq());

}

cpx EXP(double theta)

{

return cpx(cos(theta),sin(theta));

}

const double two\_pi = 4 \* acos(0);

// in: input array

// out: output array

// step: {SET TO 1} (used internally)

// size: length of the input/output {MUST BE A POWER OF 2}

// dir: either plus or minus one (direction of the FFT)

// RESULT: out[k] = \sum\_{j=0}^{size - 1} in[j] \* exp(dir \* 2pi \* i \* j \* k / size)

void FFT(cpx \*in, cpx \*out, int step, int size, int dir)

{

if(size < 1) return;

if(size == 1)

{

out[0] = in[0];

return;

}

FFT(in, out, step \* 2, size / 2, dir);

FFT(in + step, out + size / 2, step \* 2, size / 2, dir);

for(int i = 0 ; i < size / 2 ; i++)

{

cpx even = out[i];

cpx odd = out[i + size / 2];

out[i] = even + EXP(dir \* two\_pi \* i / size) \* odd;

out[i + size / 2] = even + EXP(dir \* two\_pi \* (i + size / 2) / size) \* odd;

}

}

// Usage:

// f[0...N-1] and g[0..N-1] are numbers

// Want to compute the convolution h, defined by

// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).

// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.

// Let F[0...N-1] be FFT(f), and similarly, define G and H.

// The convolution theorem says H[n] = F[n]G[n] (element-wise product).

// To compute h[] in O(N log N) time, do the following:

// 1. Compute F and G (pass dir = 1 as the argument).

// 2. Get H by element-wise multiplying F and G.

// 3. Get h by taking the inverse FFT (use dir = -1 as the argument)

// and \*dividing by N\*. DO NOT FORGET THIS SCALING FACTOR.

int main(void)

{

printf("If rows come in identical pairs, then everything works.\n");

cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};

cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};

cpx A[8];

cpx B[8];

FFT(a, A, 1, 8, 1);

FFT(b, B, 1, 8, 1);

for(int i = 0 ; i < 8 ; i++)

{

printf("%7.2lf%7.2lf", A[i].a, A[i].b);

}

printf("\n");

for(int i = 0 ; i < 8 ; i++)

{

cpx Ai(0,0);

for(int j = 0 ; j < 8 ; j++)

{

Ai = Ai + a[j] \* EXP(j \* i \* two\_pi / 8);

}

printf("%7.2lf%7.2lf", Ai.a, Ai.b);

}

printf("\n");

cpx AB[8];

for(int i = 0 ; i < 8 ; i++)

AB[i] = A[i] \* B[i];

cpx aconvb[8];

FFT(AB, aconvb, 1, 8, -1);

for(int i = 0 ; i < 8 ; i++)

aconvb[i] = aconvb[i] / 8;

for(int i = 0 ; i < 8 ; i++)

{

printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);

}

printf("\n");

for(int i = 0 ; i < 8 ; i++)

{

cpx aconvbi(0,0);

for(int j = 0 ; j < 8 ; j++)

{

aconvbi = aconvbi + a[j] \* b[(8 + i - j) % 8];

}

printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);

}

printf("\n");

return 0;

}

**//Gaussian Elimination**

// INPUT: a[][] = an nxn matrix

// b[][] = an nxm matrix

// OUTPUT: X = an nxm matrix (stored in b[][])

// A^{-1} = an nxn matrix (stored in a[][])

// returns determinant of a[][]

const double EPS = 1e-10;

typedef vector<int> VI;

typedef double T;

typedef vector<T> VT;

typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {

const int n = a.size();

const int m = b[0].size();

VI irow(n), icol(n), ipiv(n);

T det = 1;

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

int pj = -1, pk = -1;

for (int j = 0; j < n; j++) if (!ipiv[j])

for (int k = 0; k < n; k++) if (!ipiv[k])

if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }

if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }

ipiv[pk]++;

swap(a[pj], a[pk]);

swap(b[pj], b[pk]);

if (pj != pk) det \*= -1;

irow[i] = pj;

icol[i] = pk;

T c = 1.0 / a[pk][pk];

det \*= a[pk][pk];

a[pk][pk] = 1.0;

for (int p = 0; p < n; p++) a[pk][p] \*= c;

for (int p = 0; p < m; p++) b[pk][p] \*= c;

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

c = a[p][pk];

a[p][pk] = 0;

for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] \* c;

for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] \* c;

}

}

for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {

for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);

}

return det;

}

**//Geomtery**

double INF = 1e100;

double EPS = 1e-12;

struct PT {

double x, y;

PT() {}

PT(double x, double y) : x(x), y(y) {}

PT(const PT &p) : x(p.x), y(p.y) {}

PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }

PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }

PT operator \* (double c) const { return PT(x\*c, y\*c ); }

PT operator / (double c) const { return PT(x/c, y/c ); }

};

double dot(PT p, PT q) { return p.x\*q.x+p.y\*q.y; }

double dist2(PT p, PT q) { return dot(p-q,p-q); }

double cross(PT p, PT q) { return p.x\*q.y-p.y\*q.x; }

ostream &operator<<(ostream &os, const PT &p) {

os << "(" << p.x << "," << p.y << ")";

}

// rotate a point CCW or CW around the origin

PT RotateCCW90(PT p) { return PT(-p.y,p.x); }

PT RotateCW90(PT p) { return PT(p.y,-p.x); }

PT RotateCCW(PT p, double t) {

return PT(p.x\*cos(t)-p.y\*sin(t), p.x\*sin(t)+p.y\*cos(t));

}

// project point c onto line through a and b

// assuming a != b

PT ProjectPointLine(PT a, PT b, PT c) {

return a + (b-a)\*dot(c-a, b-a)/dot(b-a, b-a);

}

// project point c onto line segment through a and b

PT ProjectPointSegment(PT a, PT b, PT c) {

double r = dot(b-a,b-a);

if (fabs(r) < EPS) return a;

r = dot(c-a, b-a)/r;

if (r < 0) return a;

if (r > 1) return b;

return a + (b-a)\*r;

}

// compute distance from c to segment between a and b

double DistancePointSegment(PT a, PT b, PT c) {

return sqrt(dist2(c, ProjectPointSegment(a, b, c)));

}

// compute distance between point (x,y,z) and plane ax+by+cz=d

double DistancePointPlane(double x, double y, double z,

double a, double b, double c, double d)

{

return fabs(a\*x+b\*y+c\*z-d)/sqrt(a\*a+b\*b+c\*c);

}

// determine if lines from a to b and c to d are parallel or collinear

bool LinesParallel(PT a, PT b, PT c, PT d) {

return fabs(cross(b-a, c-d)) < EPS;

}

bool LinesCollinear(PT a, PT b, PT c, PT d) {

return LinesParallel(a, b, c, d)

&& fabs(cross(a-b, a-c)) < EPS

&& fabs(cross(c-d, c-a)) < EPS;

}

// determine if line segment from a to b intersects with

// line segment from c to d

bool SegmentsIntersect(PT a, PT b, PT c, PT d) {

if (LinesCollinear(a, b, c, d)) {

if (dist2(a, c) < EPS || dist2(a, d) < EPS ||

dist2(b, c) < EPS || dist2(b, d) < EPS) return true;

if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)

return false;

return true;

}

if (cross(d-a, b-a) \* cross(c-a, b-a) > 0) return false;

if (cross(a-c, d-c) \* cross(b-c, d-c) > 0) return false;

return true;

}

// compute intersection of line passing through a and b

// with line passing through c and d, assuming that unique

// intersection exists; for segment intersection, check if

// segments intersect first

PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {

b=b-a; d=c-d; c=c-a;

assert(dot(b, b) > EPS && dot(d, d) > EPS);

return a + b\*cross(c, d)/cross(b, d);

}

// compute center of circle given three points

PT ComputeCircleCenter(PT a, PT b, PT c) {

b=(a+b)/2;

c=(a+c)/2;

return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));

}

/\*determine if point is in a possibly non-convex polygon. Returns 1

for strictly interior points, 0 for strictly exterior points, and 0

or 1 for the remaining points. it is possible to convert this into

an exact test using integer arithmetic by taking care of the division

appropriately (making sure to deal with signs properly) and then by

writing exact tests for checking point on polygon boundary\*/

bool PointInPolygon(const vector<PT> &p, PT q) {

bool c = 0;

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

int j = (i+1)%p.size();

if ((p[i].y <= q.y && q.y < p[j].y ||

p[j].y <= q.y && q.y < p[i].y) &&

q.x < p[i].x + (p[j].x - p[i].x) \* (q.y - p[i].y) / (p[j].y - p[i].y))

c = !c;

}

return c;

}

// determine if point is on the boundary of a polygon

bool PointOnPolygon(const vector<PT> &p, PT q) {

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

if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)

return true;

return false;

}

// compute intersection of line through points a and b with

// circle centered at c with radius r > 0

vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {

vector<PT> ret;

b = b-a;

a = a-c;

double A = dot(b, b);

double B = dot(a, b);

double C = dot(a, a) - r\*r;

double D = B\*B - A\*C;

if (D < -EPS) return ret;

ret.push\_back(c+a+b\*(-B+sqrt(D+EPS))/A);

if (D > EPS)

ret.push\_back(c+a+b\*(-B-sqrt(D))/A);

return ret;

}

// compute intersection of circle centered at a with radius r

// with circle centered at b with radius R

vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {

vector<PT> ret;

double d = sqrt(dist2(a, b));

if (d > r+R || d+min(r, R) < max(r, R)) return ret;

double x = (d\*d-R\*R+r\*r)/(2\*d);

double y = sqrt(r\*r-x\*x);

PT v = (b-a)/d;

ret.push\_back(a+v\*x + RotateCCW90(v)\*y);

if (y > 0)

ret.push\_back(a+v\*x - RotateCCW90(v)\*y);

return ret;

}

// computes the area or centroid of a (possibly nonconvex) polygon, assuming

//coordinates listed in a clockwise or counterclockwise fashion.

double ComputeSignedArea(const vector<PT> &p) {

double area = 0;

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

int j = (i+1) % p.size();

area += p[i].x\*p[j].y - p[j].x\*p[i].y;

}

return area / 2.0;

}

double ComputeArea(const vector<PT> &p) {

return fabs(ComputeSignedArea(p));

}

PT ComputeCentroid(const vector<PT> &p) {

PT c(0,0);

double scale = 6.0 \* ComputeSignedArea(p);

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

int j = (i+1) % p.size();

c = c + (p[i]+p[j])\*(p[i].x\*p[j].y - p[j].x\*p[i].y);

}

return c / scale;

}

// tests if a given polygon (in CW or CCW order) is simple

bool IsSimple(const vector<PT> &p) {

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

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

int j = (i+1) % p.size();

int l = (k+1) % p.size();

if (i == l || j == k) continue;

if (SegmentsIntersect(p[i], p[j], p[k], p[l]))

return false;

}

}

return true;

}

//Distance from x,y,z to ax+by+cz+d=0

double PointPlaneDist(double x,double y,double z,

double a,double b,double c,double d)

{

return fabs(a\*x+b\*y+c\*z+d)/sqrt(a\*a+b\*b+c\*c);

}

//Distance between parallel planes ax+by+cz+d1=0 and +d2=0

double PlanePlaneDist(double a,double b,double c,

double d1,double d2)

{

return fabs(d1-d2)/sqrt(a\*a+b\*b+c\*c);

}

//Squared distance from px,py,pz to line x1,y1,z1-x2,y2,z2.

//type: 0=line 1=segment 2=ray (first is endpoint)

double PointLineDistSq(double x1,double y1,double z1,

double x2,double y2,double z2,double px,double py,double pz,

int type)

{

double pd2 = (x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2)+(z1-z2)\*(z1-z2);

double x,y,z;

if (fabs(pd2)<EPS){x=x1;y=y1;z=z1;}

else {

double u=((px-x1)\*(x2-x1)+(py-y1)\*(y2-y1)+(pz-z1)\*(z2-z1)/pd2;

x=x1+u\*(x2-x1);y=y1+u\*(y2-y1);z=z1+u\*(z2-z1);

if ((type!=0) && (u<0) {x=x1;y=y1;z=z1;}

if ((type==1) && (u>1.0){x=x2;y=y2;z=z2;}

}

return (x-px)\*(x-px)+(y-px)\*(y-py)+(z-pz)\*(z-pz);

}

**//Miller Rabin**

#define EPS 1e-7

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)

{

LL ret=0, c=a;

while(b)

{

if(b&1) ret=(ret+c)%m;

b>>=1; c=(c+c)%m;

}

return ret;

}

LL ModularExponentiation(LL a, LL n, LL m)

{

LL ret=1, c=a;

while(n)

{

if(n&1) ret=ModularMultiplication(ret, c, m);

n>>=1; c=ModularMultiplication(c, c, m);

}

return ret;

}

bool Witness(LL a, LL n)

{

LL u=n-1;

int t=0;

while(!(u&1)){u>>=1; t++;}

LL x0=ModularExponentiation(a, u, n), x1;

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

{

x1=ModularMultiplication(x0, x0, n);

if(x1==1 && x0!=1 && x0!=n-1) return true;

x0=x1;

}

if(x0!=1) return true;

return false;

}

LL Random(LL n)

{

LL ret=rand(); ret\*=32768;

ret+=rand(); ret\*=32768;

ret+=rand(); ret\*=32768;

ret+=rand();

return ret%n;

}

bool IsPrimeFast(LL n, int TRIAL)

{

while(TRIAL--)

{

LL a=Random(n-2)+1;

if(Witness(a, n)) return false;

}

return true;

}

**// RREF** INPUT: a[][] = an nxm matrix

// OUTPUT: rref[][] = an nxm matrix (stored in a[][])

// returns rank of a[][]

const double EPSILON = 1e-10;

typedef double T;

typedef vector<T> VT;

typedef vector<VT> VVT;

int rref(VVT &a) {

int n = a.size();

int m = a[0].size();

int r = 0;

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

int j = r;

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

if (fabs(a[i][c]) > fabs(a[j][c])) j = i;

if (fabs(a[j][c]) < EPSILON) continue;

swap(a[j], a[r]);

T s = 1.0 / a[r][c];

for (int j = 0; j < m; j++) a[r][j] \*= s;

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

T t = a[i][c];

for (int j = 0; j < m; j++) a[i][j] -= t \* a[r][j];

}

r++;

}

return r;

}

**// Simplex**

// maximize c^T x

// subject to Ax <= b

// x >= 0

// INPUT: A -- an m x n matrix

// b -- an m-dimensional vector

// c -- an n-dimensional vector

// x -- a vector where the optimal solution will be stored

// OUTPUT: value of the optimal solution (infinity if unbounded

// above, nan if infeasible)

// To use this code, create an LPSolver object with A, b, and c as

// arguments. Then, call Solve(x).

typedef long double DOUBLE;

typedef vector<DOUBLE> VD;

typedef vector<VD> VVD;

typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {

int m, n;

VI B, N;

VVD D;

LPSolver(const VVD &A, const VD &b, const VD &c) :

m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {

for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];

for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }

for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }

N[n] = -1; D[m + 1][n] = 1;

}

void Pivot(int r, int s) {

double inv = 1.0 / D[r][s];

for (int i = 0; i < m + 2; i++) if (i != r)

for (int j = 0; j < n + 2; j++) if (j != s)

D[i][j] -= D[r][j] \* D[i][s] \* inv;

for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] \*= inv;

for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] \*= -inv;

D[r][s] = inv;

swap(B[r], N[s]);

}

bool Simplex(int phase) {

int x = phase == 1 ? m + 1 : m;

while (true) {

int s = -1;

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

if (phase == 2 && N[j] == -1) continue;

if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;

}

if (D[x][s] > -EPS) return true;

int r = -1;

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

if (D[i][s] < EPS) continue;

if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||

(D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r = i;

}

if (r == -1) return false;

Pivot(r, s);

}

}

DOUBLE Solve(VD &x) {

int r = 0;

for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;

if (D[r][n + 1] < -EPS) {

Pivot(r, n);

if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric\_limits<DOUBLE>::infinity();

for (int i = 0; i < m; i++) if (B[i] == -1) {

int s = -1;

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

if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;

Pivot(i, s);

}

}

if (!Simplex(2)) return numeric\_limits<DOUBLE>::infinity();

x = VD(n);

for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];

return D[m][n + 1];

}

};

**//Delaunay. Quintic. Does not handle degenerate cases**

// INPUT: x[] = x-coordinates

// y[] = y-coordinates

// OUTPUT: triples = a vector containing m triples of indices

// corresponding to triangle vertices

typedef double T;

struct triple {

int i, j, k;

triple() {}

triple(int i, int j, int k) : i(i), j(j), k(k) {}

};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {

int n = x.size();

vector<T> z(n);

vector<triple> ret;

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

z[i] = x[i] \* x[i] + y[i] \* y[i];

for (int i = 0; i < n-2; i++) {

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

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

if (j == k) continue;

double xn = (y[j]-y[i])\*(z[k]-z[i]) - (y[k]-y[i])\*(z[j]-z[i]);

double yn = (x[k]-x[i])\*(z[j]-z[i]) - (x[j]-x[i])\*(z[k]-z[i]);

double zn = (x[j]-x[i])\*(y[k]-y[i]) - (x[k]-x[i])\*(y[j]-y[i]);

bool flag = zn < 0;

for (int m = 0; flag && m < n; m++)

flag = flag && ((x[m]-x[i])\*xn +

(y[m]-y[i])\*yn +

(z[m]-z[i])\*zn <= 0);

if (flag) ret.push\_back(triple(i, j, k));

}

}

}

return ret;

}

无源汇上下界可行流：

建图模型： 以前写的最大流默认的下界为0，而这里的下界却不为0，所以我们要进行再构造让每条边的下

界为0，这样做是为了方便处理。对于每根管子有一个上界容量up和一个下界容量low，我们让这根管子的

容量下界变为0，上界为up-low。可是这样做了的话流量就不守恒了，为了再次满足流量守恒，即每个节点

"入流=出流”，我们增设一个超级源点st和一个超级终点sd。我们开设一个数组du[]来记录每个节点的流量情况。

du[i]=in[i]（i节点所有入流下界之和）-out[i]（i节点所有出流下界之和）。

当du[i]大于0的时候，st到i连一条流量为du[i]的边。

当du[i]小于0的时候，i到sd连一条流量为-du[i]的边。

最后对（st，sd）求一次最大流即可，当所有附加边全部满流时（即maxflow==所有du[]>0之和)，有可行解。

有源汇上下界最大流：

如果从s到t有一个流量为a的可行流，那么从t到s连一条弧，下界为a，则这个图有一个无源汇的可行流。

如果从s到t的最大流量为amax，那么从t到s连下界为a'>amax的弧时，改造后的图没有可行流。

因此，二分答案amax，每次判断是否存在可行流，然后找amax的最大值即可。

有源汇上下界最小流：

和最大流类似，不过现在从t到s连一条弧，上界为a。同样二分答案找到最小的amax即可。

**// MATH FORMULAE**

n^(n-2) spanning trees of complete graph (n) vetices

derangement der(n) =(n-1)(der(n-1)+der(n-2)) tends to 1-e^-1

d1>=d2>=d3 .. dn can be the "degree sequence" of a simple graph iff sum(di) is even and for all k, sum\_i=1->k\_(di) <= k(k-1)+sum\_i=k+1->n(min(d\_i,k)) holds

V-E+F = 2 V(no of vertices) E(no of edge) F(no of faces)

number of pieces into which a circle is divided if n points on its circumference are joined by chords with no three internally concurrent: g(n) = nC4 + nC2+1

Let I be the number of integer points in the polygon, A be the

area of the polygon, and b be the number of integer points on the boundary, then A=i+ b/2 −1.

no of spanning tree of complete bipartite graph is m^(n-1)\*n^(m-1)

**//VIM Settings**

colorscheme desert

set tabstop=4

set t\_Co=256

imnnoremap j gj

nnoremap k gk

ap fd <Esc>

set number

set wrap

set ignorecase

set smartcase

set gdefault

set incsearch

set showmatch

set hlsearch

// INPUT: start, w[i][j] = cost of edge from i to j

// OUTPUT: dist[i] = min weight path from start to i

// prev[i] = previous node on the best path from the

// start node

typedef double T;

typedef vector<T> VT;

typedef vector<VT> VVT;

typedef vector<int> VI;

typedef vector<VI> VVI;

bool BellmanFord (const VVT &w, VT &dist, VI &prev, int start){

int n = w.size();

prev = VI(n, -1);

dist = VT(n, 1000000000);

dist[start] = 0;

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

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

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

if (dist[j] > dist[i] + w[i][j]){

if (k == n-1) return false;

dist[j] = dist[i] + w[i][j];

prev[j] = i;

}}}}

return true;}

// INPUT: start, w[i][j] = cost of edge from i to j

// OUTPUT: dist[i] = min weight path from start to i

// prev[i] = previous node on the best path from the

// start node

void Dijkstra (const VVT &w, VT &dist, VI &prev, int start){

int n = w.size();

VI found (n);

prev = VI(n, -1);

dist = VT(n, 1000000000);

dist[start] = 0;

while (start != -1){

found[start] = true;

int best = -1;

for (int k = 0; k < n; k++) if (!found[k]){

if (dist[k] > dist[start] + w[start][k]){

dist[k] = dist[start] + w[start][k];

prev[k] = start;

}

if (best == -1 || dist[k] < dist[best]) best = k;

}

start = best;}}

// INPUT: w[i][j] = weight of edge from i to j

// OUTPUT: w[i][j] = shortest path from i to j

// prev[i][j] = node before j on the best path starting at i

bool FloydWarshall (VVT &w, VVI &prev){

int n = w.size();

prev = VVI (n, VI(n, -1));

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

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

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

if (w[i][j] > w[i][k] + w[k][j]){

w[i][j] = w[i][k] + w[k][j];

prev[i][j] = k;}}}}

// check for negative weight cycles

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

if (w[i][i] < 0) return false;

return true;

}

**// AdjList dinic**

// INPUT:

// - graph, constructed using AddEdge()

// - source and sink

//

// OUTPUT:

// - maximum flow value

// - To obtain actual flow values, look at edges with capacity > 0

// (zero capacity edges are residual edges).

struct Edge {

int u, v;

LL cap, flow;

Edge() {}

Edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}

};

struct Dinic {

int N;

vector<Edge> E;

vector<vector<int>> g;

vector<int> d, pt;

Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

void AddEdge(int u, int v, LL cap) {

if (u != v) {

E.emplace\_back(Edge(u, v, cap));

g[u].emplace\_back(E.size() - 1);

E.emplace\_back(Edge(v, u, 0));

g[v].emplace\_back(E.size() - 1);}}

bool BFS(int S, int T) {

queue<int> q({S});

fill(d.begin(), d.end(), N + 1);

d[S] = 0;

while(!q.empty()) {

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

if (u == T) break;

for (int k: g[u]) {

Edge &e = E[k];

if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {

d[e.v] = d[e.u] + 1;

q.emplace(e.v);}}}

return d[T] != N + 1;

}

LL DFS(int u, int T, LL flow = -1) {

if (u == T || flow == 0) return flow;

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

Edge &e = E[g[u][i]];

Edge &oe = E[g[u][i]^1];

if (d[e.v] == d[e.u] + 1) {

LL amt = e.cap - e.flow;

if (flow != -1 && amt > flow) amt = flow;

if (LL pushed = DFS(e.v, T, amt)) {

e.flow += pushed;

oe.flow -= pushed;

return pushed;}}}

return 0;

}

LL MaxFlow(int S, int T) {

LL total = 0;

while (BFS(S, T)) {

fill(pt.begin(), pt.end(), 0);

while (LL flow = DFS(S, T))

total += flow;

}

return total;

}

};

**// Eulerian Path: use every edge exactly once**

struct Edge;

typedef list<Edge>::iterator iter;

struct Edge

{

int next\_vertex;

iter reverse\_edge;

Edge(int next\_vertex)

:next\_vertex(next\_vertex) { }

};

const int max\_vertices = ;

int num\_vertices;

list<Edge> adj[max\_vertices]; // adjacency list

vector<int> path;

void find\_path(int v)

{

while(adj[v].size() > 0)

{

int vn = adj[v].front().next\_vertex;

adj[vn].erase(adj[v].front().reverse\_edge);

adj[v].pop\_front();

find\_path(vn);

}

path.push\_back(v);

}

void add\_edge(int a, int b)

{

adj[a].push\_front(Edge(b));

iter ita = adj[a].begin();

adj[b].push\_front(Edge(a));

iter itb = adj[b].begin();

ita->reverse\_edge = itb;

itb->reverse\_edge = ita;

}

**// Implementation of Dijkstra's algorithm using adjacency lists**

**// and priority queue for efficiency.**

void dijkstra(){

dist[E] = 0; // INF = 1B to avoid overflow

priority\_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, E)); while (!pq.empty()) { // main loop

ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex

int d = front.first, u = front.second;

if (d > dist[u]) continue; // this is a very important check

for (int j = 0; j < (int)adj[u].size(); j++) {

ii v = adj[u][j]; // all outgoing edges from u

if (dist[u] + v.second < dist[v.first]) {

dist[v.first] = dist[u] + v.second; // relax operation

pq.push(ii(dist[v.first], v.first));}}}}

**// LCA**

const int max\_nodes, log\_max\_nodes;

int num\_nodes, log\_num\_nodes, root;

vector<int> children[max\_nodes]; // children[i] contains the children of node i

int A[max\_nodes][log\_max\_nodes+1]; // A[i][j] is the 2^j-th ancestor of node i, or -1 if that ancestor does not exist

int L[max\_nodes]; // L[i] is the distance between node i and the root

// floor of the binary logarithm of n

int lb(unsigned int n) {

if(n==0)

return -1;

int p = 0;

if (n >= 1<<16) { n >>= 16; p += 16; }

if (n >= 1<< 8) { n >>= 8; p += 8; }

if (n >= 1<< 4) { n >>= 4; p += 4; }

if (n >= 1<< 2) { n >>= 2; p += 2; }

if (n >= 1<< 1) { p += 1; }

return p;

}

void DFS(int i, int l) {

L[i] = l;

for(int j = 0; j < children[i].size(); j++)

DFS(children[i][j], l+1);

}

int LCA(int p, int q) {

// ensure node p is at least as deep as node q

if(L[p] < L[q])

swap(p, q);

// "binary search" for the ancestor of node p situated on the same level as q

for(int i = log\_num\_nodes; i >= 0; i--)

if(L[p] - (1<<i) >= L[q])

p = A[p][i];

if(p == q)

return p;

// "binary search" for the LCA

for(int i = log\_num\_nodes; i >= 0; i--)

if(A[p][i] != -1 && A[p][i] != A[q][i]) {

p = A[p][i];

q = A[q][i];

}

return A[p][0];

}

int main(int argc,char\* argv[]) {

// read num\_nodes, the total number of nodes

log\_num\_nodes=lb(num\_nodes);

for(int i = 0; i < num\_nodes; i++) {

int p;

// read p, the parent of node i or -1 if node i is the root

A[i][0] = p;

if(p != -1)

children[p].push\_back(i);

else

root = i;

}

// precompute A using dynamic programming

for(int j = 1; j <= log\_num\_nodes; j++)

for(int i = 0; i < num\_nodes; i++)

if(A[i][j-1] != -1)

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

else

A[i][j] = -1;

// precompute L

DFS(root, 0);

return 0;

}

// INPUT: w[i][j] = edge between row node i and column node j

// OUTPUT: mr[i] = assignment for row node i, -1 if unassigned

// mc[j] = assignment for column node j, -1 if unassigned

// function returns number of matches made

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {

for (int j = 0; j < w[i].size(); j++) {

if (w[i][j] && !seen[j]) {

seen[j] = true;

if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {

mr[i] = j;

mc[j] = i;

return true;}}}

return false;

}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {

mr = VI(w.size(), -1);

mc = VI(w[0].size(), -1);

int ct = 0;

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

VI seen(w[0].size());

if (FindMatch(i, w, mr, mc, seen)) ct++;

}

return ct;

}

**// Dinic Adj Matrix**

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

// OUTPUT:

// - maximum flow value

// - To obtain the actual flow, look at positive values only.

struct MaxFlow {

int N;

VVI cap, flow;

VI dad, Q;

MaxFlow(int N) :

N(N), cap(N, VI(N)), flow(N, VI(N)), dad(N), Q(N) {}

void AddEdge(int from, int to, int cap) {

this->cap[from][to] += cap;

}

int BlockingFlow(int s, int t) {

fill(dad.begin(), dad.end(), -1);

dad[s] = -2;

int head = 0, tail = 0;

Q[tail++] = s;

while (head < tail) {

int x = Q[head++];

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

if (dad[i] == -1 && cap[x][i] - flow[x][i] > 0) {

dad[i] = x;

Q[tail++] = i;}}}

if (dad[t] == -1) return 0;

int totflow = 0;

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

if (dad[i] == -1) continue;

int amt = cap[i][t] - flow[i][t];

for (int j = i; amt && j != s; j = dad[j])

amt = min(amt, cap[dad[j]][j] - flow[dad[j]][j]);

if (amt == 0) continue;

flow[i][t] += amt;

flow[t][i] -= amt;

for (int j = i; j != s; j = dad[j]) {

flow[dad[j]][j] += amt;

flow[j][dad[j]] -= amt;

}

totflow += amt;

}

return totflow;

}

int GetMaxFlow(int source, int sink) {

int totflow = 0;

while (int flow = BlockingFlow(source, sink))

totflow += flow;

return totflow;}};

// cost[i][j] = cost for pairing left node i with right node j

// Lmate[i] = index of right node that left node i pairs with

// Rmate[j] = index of left node that right node j pairs with

// The values in cost[i][j] may be positive or negative. To perform

// maximization, simply negate the cost[][] matrix.

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {

int n = int(cost.size());

// construct dual feasible solution

VD u(n);

VD v(n);

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

u[i] = cost[i][0];

for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);

}

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

v[j] = cost[0][j] - u[0];

for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);

}

// construct primal solution satisfying complementary slackness

Lmate = VI(n, -1);

Rmate = VI(n, -1);

int mated = 0;

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

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

if (Rmate[j] != -1) continue;

if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {

Lmate[i] = j;

Rmate[j] = i;

mated++;

break;}}}

VD dist(n);

VI dad(n);

VI seen(n);

// repeat until primal solution is feasible

while (mated < n) {

// find an unmatched left node

int s = 0;

while (Lmate[s] != -1) s++;

// initialize Dijkstra

fill(dad.begin(), dad.end(), -1);

fill(seen.begin(), seen.end(), 0);

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

dist[k] = cost[s][k] - u[s] - v[k];

int j = 0;

while (true) {

// find closest

j = -1;

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

if (seen[k]) continue;

if (j == -1 || dist[k] < dist[j]) j = k;

}

seen[j] = 1;

// termination condition

if (Rmate[j] == -1) break;

// relax neighbors

const int i = Rmate[j];

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

if (seen[k]) continue;

const double new\_dist = dist[j] + cost[i][k] - u[i] - v[k];

if (dist[k] > new\_dist) {

dist[k] = new\_dist;

dad[k] = j;

}

}

}

// update dual variables

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

if (k == j || !seen[k]) continue;

const int i = Rmate[k];

v[k] += dist[k] - dist[j];

u[i] -= dist[k] - dist[j];

}

u[s] += dist[j];

// augment along path

while (dad[j] >= 0) {

const int d = dad[j];

Rmate[j] = Rmate[d];

Lmate[Rmate[j]] = j;

j = d;

}

Rmate[j] = s;

Lmate[s] = j;

mated++;

}

double value = 0;

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

value += cost[i][Lmate[i]];

return value;

//**MinCostMaxFlow**

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

// OUTPUT:

// - (maximum flow value, minimum cost value)

// - To obtain the actual flow, look at positive values only.

const L INF = numeric\_limits<L>::max() / 4;

struct MinCostMaxFlow {

int N;

VVL cap, flow, cost;

VI found;

VL dist, pi, width;

VPII dad;

MinCostMaxFlow(int N) :

N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),

found(N), dist(N), pi(N), width(N), dad(N) {}

void AddEdge(int from, int to, L cap, L cost) {

this->cap[from][to] = cap;

this->cost[from][to] = cost;

}

void Relax(int s, int k, L cap, L cost, int dir) {

L val = dist[s] + pi[s] - pi[k] + cost;

if (cap && val < dist[k]) {

dist[k] = val;

dad[k] = make\_pair(s, dir);

width[k] = min(cap, width[s]);}

}

L Dijkstra(int s, int t) {

fill(found.begin(), found.end(), false);

fill(dist.begin(), dist.end(), INF);

fill(width.begin(), width.end(), 0);

dist[s] = 0;

width[s] = INF;

while (s != -1) {

int best = -1;

found[s] = true;

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

if (found[k]) continue;

Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);

Relax(s, k, flow[k][s], -cost[k][s], -1);

if (best == -1 || dist[k] < dist[best]) best = k;

}

s = best;

}

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

pi[k] = min(pi[k] + dist[k], INF);

return width[t];

}

pair<L, L> GetMaxFlow(int s, int t) {

L totflow = 0, totcost = 0;

while (L amt = Dijkstra(s, t)) {

totflow += amt;

for (int x = t; x != s; x = dad[x].first) {

if (dad[x].second == 1) {

flow[dad[x].first][x] += amt;

totcost += amt \* cost[dad[x].first][x];

} else {

flow[x][dad[x].first] -= amt;

totcost -= amt \* cost[x][dad[x].first];}}}

return make\_pair(totflow, totcost);}};

**// Min cut adj Mat**

// INPUT:

// - graph, constructed using AddEdge()

//

// OUTPUT:

// - (min cut value, nodes in half of min cut)

pair<int, VI> GetMinCut(VVI &weights) {

int N = weights.size();

VI used(N), cut, best\_cut;

int best\_weight = -1;

for (int phase = N-1; phase >= 0; phase--) {

VI w = weights[0];

VI added = used;

int prev, last = 0;

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

prev = last;

last = -1;

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

if (!added[j] && (last == -1 || w[j] > w[last])) last = j;

if (i == phase-1) {

for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];

for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];

used[last] = true;

cut.push\_back(last);

if (best\_weight == -1 || w[last] < best\_weight) {

best\_cut = cut;

best\_weight = w[last];

}

} else {

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

w[j] += weights[last][j];

added[last] = true;}}}

return make\_pair(best\_weight, best\_cut);

}

**// PUSH RELABEL**

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

// OUTPUT:

// - maximum flow value

// - To obtain the actual flow values, look at all edges with

// capacity > 0 (zero capacity edges are residual edges).

struct Edge {

int from, to, cap, flow, index;

Edge(int from, int to, int cap, int flow, int index) :

from(from), to(to), cap(cap), flow(flow), index(index) {}

};

struct PushRelabel {

int N;

vector<vector<Edge> > G;

vector<LL> excess;

vector<int> dist, active, count;

queue<int> Q;

PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2\*N) {}

void AddEdge(int from, int to, int cap) {

G[from].push\_back(Edge(from, to, cap, 0, G[to].size()));

if (from == to) G[from].back().index++;

G[to].push\_back(Edge(to, from, 0, 0, G[from].size() - 1));

}

void Enqueue(int v) {

if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }

}

void Push(Edge &e) {

int amt = int(min(excess[e.from], LL(e.cap - e.flow)));

if (dist[e.from] <= dist[e.to] || amt == 0) return;

e.flow += amt;

G[e.to][e.index].flow -= amt;

excess[e.to] += amt;

excess[e.from] -= amt;

Enqueue(e.to);

}

void Gap(int k) {

for (int v = 0; v < N; v++) {

if (dist[v] < k) continue;

count[dist[v]]--;

dist[v] = max(dist[v], N+1);

count[dist[v]]++;

Enqueue(v);}}

void Relabel(int v) {

count[dist[v]]--;

dist[v] = 2\*N;

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

if (G[v][i].cap - G[v][i].flow > 0)

dist[v] = min(dist[v], dist[G[v][i].to] + 1);

count[dist[v]]++;

Enqueue(v);

}

void Discharge(int v) {

for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);

if (excess[v] > 0) {

if (count[dist[v]] == 1)

Gap(dist[v]);

else

Relabel(v);}}

LL GetMaxFlow(int s, int t) {

count[0] = N-1;

count[N] = 1;

dist[s] = N;

active[s] = active[t] = true;

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

excess[s] += G[s][i].cap;

Push(G[s][i]);

}

while (!Q.empty()) {

int v = Q.front();

Q.pop();

active[v] = false;

Discharge(v);

}

LL totflow = 0;

for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;

return totflow;}};

//**Topological Sort**

// Running time: O(|V|^2). If you use adjacency lists (vector<map<int> >),

// the running time is reduced to O(|E|).

// INPUT: w[i][j] = 1 if i should come before j, 0 otherwise

// OUTPUT: a permutation of 0,...,n-1 (stored in a vector)

// which represents an ordering of the nodes which

// is consistent with w

// If no ordering is possible, false is returned.

bool TopologicalSort (const VVI &w, VI &order){

int n = w.size();

VI parents (n);

queue<int> q;

order.clear();

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

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

if (w[j][i]) parents[i]++;

if (parents[i] == 0) q.push (i);

}

while (q.size() > 0){

int i = q.front();

q.pop();

order.push\_back (i);

for (int j = 0; j < n; j++) if (w[i][j]){

parents[j]--;

if (parents[j] == 0) q.push (j);

}

}

return (order.size() == n);

}

/\*

Uses **Kruskal**'s Algorithm to calculate the weight of the minimum spanning

forest (union of minimum spanning trees of each connected component) of

a possibly disjoint graph, given in the form of a matrix of edge weights

(-1 if no edge exists). Returns the weight of the minimum spanning

forest (also calculates the actual edges - stored in T). Note: uses a

disjoint-set data structure with amortized (effectively) constant time per

union/find. Runs in O(E\*log(E)) time.

\*/

typedef int T;

struct edge

{

int u, v;

T d;

};

struct edgeCmp {

int operator()(const edge& a, const edge& b) { return a.d > b.d; }

};

int find(vector <int>& C, int x) { return (C[x] == x) ? x : C[x] = find(C, C[x]); }

T Kruskal(vector <vector <T> >& w)

{

int n = w.size();

T weight = 0;

vector <int> C(n), R(n);

for(int i=0; i<n; i++) { C[i] = i; R[i] = 0; }

vector <edge> T;

priority\_queue <edge, vector <edge>, edgeCmp> E;

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

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

if(w[i][j] >= 0) {

edge e;

e.u = i; e.v = j; e.d = w[i][j];

E.push(e);

}

while(T.size() < n-1 && !E.empty()) {

edge cur = E.top(); E.pop();

int uc = find(C, cur.u), vc = find(C, cur.v);

if(uc != vc)

{

T.push\_back(cur); weight += cur.d;

if(R[uc] > R[vc]) C[vc] = uc;

else if(R[vc] > R[uc]) C[uc] = vc;

else { C[vc] = uc; R[uc]++; }}}

return weight;

}

**// articulation point / bridge**

void articulationPointAndBridge(int u) {

dfs\_low[u] = dfs\_num[u] = dfsNumberCounter++; // dfs\_low[u] <= dfs\_num[u]

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

if (dfs\_num[v.first] == UNVISITED) {

dfs\_parent[v.first] = u;

if (u == dfsRoot) rootChildren++;

articulationPointAndBridge(v.first);

if (dfs\_low[v.first] >= dfs\_num[u]) articulation\_vertex[u] = true;

if (dfs\_low[v.first] > dfs\_num[u])

printf(" Edge (%d, %d) is a bridge\n", u, v.first);

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v.first]); // update dfs\_low[u] }

else if (v.first != dfs\_parent[u]) // a back edge and not direct cycle

dfs\_low[u] = min(dfs\_low[u], dfs\_num[v.first]); // update dfs\_low[u]

}}}

**// SCC**

vi dfs\_num, dfs\_low, S, visited; // global variables

void tarjanSCC(int u) {

dfs\_low[u] = dfs\_num[u] = dfsNumberCounter++; // dfs\_low[u] <= dfs\_num[u]

S.push\_back(u); // stores u in a vector based on order of visitation

visited[u] = 1;

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

if (dfs\_num[v.first] == UNVISITED)

tarjanSCC(v.first);

if (visited[v.first]) // condition for update

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v.first]); }

if (dfs\_low[u] == dfs\_num[u]) { // if this is a root (start) of an SCC

printf("SCC %d:", ++numSCC); // this part is done after recursion

while (1) {

int v = S.back(); S.pop\_back(); visited[v] = 0; printf(" %d", v);

if (u == v) break; }

printf("\n"); }}

Data Structures

==

**Disjoint Sets**

--

```C++

class UnionFind {

private:

vi p, rank, setSize;int numSets;

public:

UnionFind(int N) {

setSize.assign(N, 1); numSets = N; rank.assign(N, 0);

p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }

int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }

bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }

void unionSet(int i, int j) {

if (!isSameSet(i, j)) { numSets--;

int x = findSet(i), y = findSet(j);

// rank is used to keep the tree short

if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }

else { p[x] = y; setSize[y] += setSize[x];

if (rank[x] == rank[y]) rank[y]++; } } }

int numDisjointSets() { return numSets; }

int sizeOfSet(int i) { return setSize[findSet(i)]; }};

```

Segment Tree

--

**For dynamic range minimum queries**

```C++

class SegmentTree { // the segment tree is stored like a heap array

private: vi st, A; // recall that vi is: typedef vector<int> vi;

int n;

int left (int p) { return p << 1; } // same as binary heap operations

int right(int p) { return (p << 1) + 1; }

void build(int p, int L, int R) { // O(n log n)

if (L == R) // as L == R, either one is fine

st[p] = L; // store the index

else { // recursively compute the values

build(left(p) , L , (L + R) / 2);

build(right(p), (L + R) / 2 + 1, R );

int p1 = st[left(p)], p2 = st[right(p)];

st[p] = (A[p1] <= A[p2]) ? p1 : p2;

} }

int rmq(int p, int L, int R, int i, int j) { // O(log n)

if (i > R || j < L) return -1; // current segment outside query range

if (L >= i && R <= j) return st[p]; // inside query range

// compute the min position in the left and right part of the interval

int p1 = rmq(left(p) , L , (L+R) / 2, i, j);

int p2 = rmq(right(p), (L+R) / 2 + 1, R , i, j);

if (p1 == -1) return p2; // if we try to access segment outside query

if (p2 == -1) return p1; // same as above

return (A[p1] <= A[p2]) ? p1 : p2; } // as as in build routine

int update\_point(int p, int L, int R, int idx, int new\_value) {

// this update code is still preliminary, i == j

// must be able to update range in the future!

int i = idx, j = idx;

// if the current interval does not intersect

// the update interval, return this st node value!

if (i > R || j < L)

return st[p];

// if the current interval is included in the update range,

// update that st[node]

if (L == i && R == j) {

A[i] = new\_value; // update the underlying array

return st[p] = L; // this index

}

// compute the minimum pition in the

// left and right part of the interval

int p1, p2;

p1 = update\_point(left(p) , L , (L + R) / 2, idx, new\_value);

p2 = update\_point(right(p), (L + R) / 2 + 1, R , idx, new\_value);

// return the pition where the overall minimum is

return st[p] = (A[p1] <= A[p2]) ? p1 : p2;

}

public:

SegmentTree(const vi &\_A) {

A = \_A; n = (int)A.size(); // copy content for local usage

st.assign(4 \* n, 0); // create large enough vector of zeroes

build(1, 0, n - 1); // recursive build

}

int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); } // overloading

int update\_point(int idx, int new\_value) {

return update\_point(1, 0, n - 1, idx, new\_value); }

};

int main() {

int arr[] = { 18, 17, 13, 19, 15, 11, 20 }; // the original array

vi A(arr, arr + 7); // copy the contents to a vector

SegmentTree st(A);

printf(" idx 0, 1, 2, 3, 4, 5, 6\n");

printf(" A is {18,17,13,19,15, 11,20}\n");

printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // answer = index 2

printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // answer = index 5

printf("RMQ(3, 4) = %d\n", st.rmq(3, 4)); // answer = index 4

printf("RMQ(0, 0) = %d\n", st.rmq(0, 0)); // answer = index 0

printf("RMQ(0, 1) = %d\n", st.rmq(0, 1)); // answer = index 1

printf("RMQ(0, 6) = %d\n", st.rmq(0, 6)); // answer = index 5

printf(" idx 0, 1, 2, 3, 4, 5, 6\n");

printf("Now, modify A into {18,17,13,19,15,100,20}\n");

st.update\_point(5, 100); // update A[5] from 11 to 100

printf("These values do not change\n");

printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // 2

printf("RMQ(3, 4) = %d\n", st.rmq(3, 4)); // 4

printf("RMQ(0, 0) = %d\n", st.rmq(0, 0)); // 0

printf("RMQ(0, 1) = %d\n", st.rmq(0, 1)); // 1

printf("These values change\n");

printf("RMQ(0, 6) = %d\n", st.rmq(0, 6)); // 5->2

printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // 5->4

printf("RMQ(4, 5) = %d\n", st.rmq(4, 5)); // 5->4

return 0;

}

```

**Binary Indexed Tree**

--

For dynamic cumulative frequency table and range sum queries

```C++

#define LSOne(S) (S & (-S))

class FenwickTree {

private:

vi ft;

public:

FenwickTree() {}

// initialization: n + 1 zeroes, ignore index 0

FenwickTree(int n) { ft.assign(n + 1, 0); }

int rsq(int b) { // returns RSQ(1, b)

int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];

return sum; }

int rsq(int a, int b) { // returns RSQ(a, b)

return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }

// adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)

void adjust(int k, int v) { // note: n = ft.size() - 1

for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }

};

int main() { // idx 0 1 2 3 4 5 6 7 8 9 10, no index 0!

FenwickTree ft(10); // ft = {-,0,0,0,0,0,0,0, 0,0,0}

ft.adjust(2, 1); // ft = {-,0,1,0,1,0,0,0, 1,0,0}, idx 2,4,8 => +1

ft.adjust(4, 1); // ft = {-,0,1,0,2,0,0,0, 2,0,0}, idx 4,8 => +1

ft.adjust(5, 2); // ft = {-,0,1,0,2,2,2,0, 4,0,0}, idx 5,6,8 => +2

ft.adjust(6, 3); // ft = {-,0,1,0,2,2,5,0, 7,0,0}, idx 6,8 => +3

ft.adjust(7, 2); // ft = {-,0,1,0,2,2,5,2, 9,0,0}, idx 7,8 => +2

ft.adjust(8, 1); // ft = {-,0,1,0,2,2,5,2,10,0,0}, idx 8 => +1

ft.adjust(9, 1); // ft = {-,0,1,0,2,2,5,2,10,1,1}, idx 9,10 => +1

printf("%d\n", ft.rsq(1, 1)); // 0 => ft[1] = 0

printf("%d\n", ft.rsq(1, 2)); // 1 => ft[2] = 1

printf("%d\n", ft.rsq(1, 6)); // 7 => ft[6] + ft[4] = 5 + 2 = 7

printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11

printf("%d\n", ft.rsq(3, 6)); // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1

ft.adjust(5, 2); // update demo

printf("%d\n", ft.rsq(1, 10)); // now 13

} // return 0;

```

**Lazy-segment Tree**

---

```C++

#include <cctype>

#include <cmath>

#include <cstdlib>

const int N = 1.1e6 + 10, INF = 0x3f3f3f3f, MOD = 1e9 + 7;

string str;

struct Node {

int l, r;

int sum;

Node() {}

Node(int l, int r): l(l), r(r) {}

int len() {

return r - l + 1;

}

void set(int v) {

if(v == -1) return;

if(v == 2) sum = len() - sum;

else sum = len() \* v;

}

} dat[N << 2];

int tag[N << 2];

void pushUp(int rt) {

dat[rt].sum = dat[rt << 1].sum + dat[rt << 1 | 1].sum;

}

void combineTag(int fa, int& son) {

if(fa == 2) {

if(son == -1) son = 2;

else if(son == 2) son = -1;

else son ^= 1; // switch 0, 1

} else son = fa; //set 0, 1

}

void pushDown(int rt) {

if(tag[rt] == -1) return;

int ls = rt << 1, rs = ls | 1;

dat[ls].set(tag[rt]);

dat[rs].set(tag[rt]);

combineTag(tag[rt], tag[ls]);

combineTag(tag[rt], tag[rs]);

tag[rt] = -1;

}

void build(int l, int r, int rt) {

dat[rt] = Node(l, r);

tag[rt] = -1;

if(l == r) {

dat[rt].sum = str[l] - '0';

return;

}

int m = l + r >> 1;

build(l, m, rt << 1);

build(m + 1, r, rt << 1 | 1);

pushUp(rt);

}

void update(int L, int R, int v, int rt) {

if(L <= dat[rt].l && dat[rt].r <= R) {

dat[rt].set(v);

combineTag(v, tag[rt]);

return;

}

pushDown(rt);

int m = dat[rt].l + dat[rt].r >> 1;

if(L <= m) update(L, R, v, rt << 1);

if(R > m) update(L, R, v, rt << 1 | 1);

pushUp(rt);

}

int query(int L, int R, int rt) {

if(L <= dat[rt].l && dat[rt].r <= R) return dat[rt].sum;

pushDown(rt);

int m = dat[rt].l + dat[rt].r >> 1;

int ret = 0;

if(L <= m) ret += query(L, R, rt << 1);

if(R > m) ret += query(L, R, rt << 1 | 1);

return ret;

}

int main() {

build(0, str.size() - 1, 1); //init flag to -1

if(\*op == 'F') update(a, b, 1, 1); //set change flag to 1

else if(\*op == 'E') update(a, b, 0, 1); //set change flag to 0

else if(\*op == 'I') update(a, b, 2, 1); // set flip flag

else printf("Q%d: %d\n", ++qs, query(a, b, 1));

}

```

KD-Tree

---

```C++

#include <limits>

#include <cstdlib>

// number type for coordinates, and its maximum value

typedef long long ntype;

const ntype sentry = numeric\_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D

struct point {

ntype x, y;

point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}

};

bool operator==(const point &a, const point &b)

{return a.x == b.x && a.y == b.y;}

// sorts points on x-coordinate

bool on\_x(const point &a, const point &b)

{return a.x < b.x;}

// sorts points on y-coordinate

bool on\_y(const point &a, const point &b)

{return a.y < b.y;}

// squared distance between points

ntype pdist2(const point &a, const point &b)

{ntype dx = a.x-b.x, dy = a.y-b.y;return dx\*dx + dy\*dy;}

**// bounding box for a set of points**

struct bbox

{

ntype x0, x1, y0, y1;

bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

// computes bounding box from a bunch of points

void compute(const vector<point> &v) {

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

x0 = min(x0, v[i].x); x1 = max(x1, v[i].x);

y0 = min(y0, v[i].y); y1 = max(y1, v[i].y);

}

}

// squared distance between a point and this bbox, 0 if inside

ntype distance(const point &p) {

if (p.x < x0) {

if (p.y < y0) return pdist2(point(x0, y0), p);

else if (p.y > y1) return pdist2(point(x0, y1), p);

else return pdist2(point(x0, p.y), p);

}

else if (p.x > x1) {

if (p.y < y0) return pdist2(point(x1, y0), p);

else if (p.y > y1) return pdist2(point(x1, y1), p);

else return pdist2(point(x1, p.y), p);

}

else {

if (p.y < y0) return pdist2(point(p.x, y0), p);

else if (p.y > y1) return pdist2(point(p.x, y1), p);

else return 0;

}

}

};

**// stores a single node of the kd-tree, either internal or leaf**

struct kdnode

{

bool leaf; // true if this is a leaf node (has one point)

point pt; // the single point of this is a leaf

bbox bound; // bounding box for set of points in children

kdnode \*first, \*second; // two children of this kd-node

kdnode() : leaf(false), first(0), second(0) {}

~kdnode() { if (first) delete first; if (second) delete second; }

// intersect a point with this node (returns squared distance)

ntype intersect(const point &p) {

return bound.distance(p);

}

// recursively builds a kd-tree from a given cloud of points

void construct(vector<point> &vp)

{

// compute bounding box for points at this node

bound.compute(vp);

// if we're down to one point, then we're a leaf node

if (vp.size() == 1) {

leaf = true;

pt = vp[0];

}

else {

// split on x if the bbox is wider than high (not best heuristic...)

if (bound.x1-bound.x0 >= bound.y1-bound.y0)

sort(vp.begin(), vp.end(), on\_x);

// otherwise split on y-coordinate

else

sort(vp.begin(), vp.end(), on\_y);

// divide by taking half the array for each child

// (not best performance if many duplicates in the middle)

int half = vp.size()/2;

vector<point> vl(vp.begin(), vp.begin()+half);

vector<point> vr(vp.begin()+half, vp.end());

first = new kdnode(); first->construct(vl);

second = new kdnode(); second->construct(vr);

}

}

};

**// simple kd-tree class to hold the tree and handle queries**

struct kdtree

{

kdnode \*root;

// constructs a kd-tree from a points (copied here, as it sorts them)

kdtree(const vector<point> &vp) {

vector<point> v(vp.begin(), vp.end());

root = new kdnode();

root->construct(v);

}

~kdtree() { delete root; }

// recursive search method returns squared distance to nearest point

ntype search(kdnode \*node, const point &p)

{

if (node->leaf) {

// commented special case tells a point not to find itself

// if (p == node->pt) return sentry;

// else

return pdist2(p, node->pt);

}

ntype bfirst = node->first->intersect(p);

ntype bsecond = node->second->intersect(p);

// choose the side with the closest bounding box to search first

// (note that the other side is also searched if needed)

if (bfirst < bsecond) {

ntype best = search(node->first, p);

if (bsecond < best)

best = min(best, search(node->second, p));

return best;

}

else {

ntype best = search(node->second, p);

if (bfirst < best)

best = min(best, search(node->first, p));

return best;

}

}

// squared distance to the nearest

ntype nearest(const point &p) {

return search(root, p);

}

};

int main()

{

// generate some random points for a kd-tree

vector<point> vp;

for (int i = 0; i < 100000; ++i) {

vp.push\_back(point(rand()%100000, rand()%100000));

}

kdtree tree(vp);

// query some points

for (int i = 0; i < 10; ++i) {

point q(rand()%100000, rand()%100000);

cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"

<< " is " << tree.nearest(q) << endl;

}

return 0;

}

```

**Lowest Common ancestor**

---

```C++

const int max\_nodes, log\_max\_nodes;

int num\_nodes, log\_num\_nodes, root;

vector<int> children[max\_nodes]; // children[i] contains the children of node i

int A[max\_nodes][log\_max\_nodes+1]; // A[i][j] is the 2^j-th ancestor of node i, or -1 if that ancestor does not exist

int L[max\_nodes]; // L[i] is the distance between node i and the root

// floor of the binary logarithm of n

int lb(unsigned int n)

{

if(n==0)return -1;

int p = 0;

if (n >= 1<<16) { n >>= 16; p += 16; }

if (n >= 1<< 8) { n >>= 8; p += 8; }

if (n >= 1<< 4) { n >>= 4; p += 4; }

if (n >= 1<< 2) { n >>= 2; p += 2; }

if (n >= 1<< 1) { p += 1; }

return p;

}

void DFS(int i, int l)

{

L[i] = l;

for(int j = 0; j < children[i].size(); j++)

DFS(children[i][j], l+1);

}

int LCA(int p, int q)

{

// ensure node p is at least as deep as node q

if(L[p] < L[q])

swap(p, q);

// "binary search" for the ancestor of node p situated on the same level as q

for(int i = log\_num\_nodes; i >= 0; i--)

if(L[p] - (1<<i) >= L[q])

p = A[p][i];

if(p == q)

return p;

// "binary search" for the LCA

for(int i = log\_num\_nodes; i >= 0; i--)

if(A[p][i] != -1 && A[p][i] != A[q][i])

{

p = A[p][i];

q = A[q][i];

}

return A[p][0];

}

int main(int argc,char\* argv[])

{

// read num\_nodes, the total number of nodes

log\_num\_nodes=lb(num\_nodes);

for(int i = 0; i < num\_nodes; i++)

{

int p;

// read p, the parent of node i or -1 if node i is the root

A[i][0] = p;

if(p != -1)

children[p].push\_back(i);

else

root = i;

}

// precompute A using dynamic programming

for(int j = 1; j <= log\_num\_nodes; j++)

for(int i = 0; i < num\_nodes; i++)

if(A[i][j-1] != -1)

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

else

A[i][j] = -1;

// precompute L

DFS(root, 0);

return 0;

}

```

**Dates**

---

```C++

string dayOfWeek[] = {"Mon","Tue"...};

// converts Gregorian date to integer (Julian day number)

int dateToInt (int m, int d, int y){

return

1461 \* (y + 4800 + (m - 14) / 12) / 4 +

367 \* (m - 2 - (m - 14) / 12 \* 12) / 12 -

3 \* ((y + 4900 + (m - 14) / 12) / 100) / 4 +

d - 32075;

}

// converts integer (Julian day number) to Gregorian date: month/day/year

void intToDate (int jd, int &m, int &d, int &y){

int x, n, i, j;

x = jd + 68569; n = 4 \* x / 146097;

x -= (146097 \* n + 3) / 4; i = (4000 \* (x + 1)) / 1461001;

x -= 1461 \* i / 4 - 31; j = 80 \* x / 2447;

d = x - 2447 \* j / 80;

x = j / 11;

m = j + 2 - 12 \* x;

y = 100 \* (n - 49) + i + x;

}

// converts integer (Julian day number) to day of week

string intToDay (int jd){ return dayOfWeek[jd % 7];}

```

Sparse table

---

```C++

#include <cmath>

#define MAX\_N 1000 // adjust this value as needed

#define LOG\_TWO\_N 10 // 2^10 > 1000, adjust this value as needed\

class RMQ { // Range Minimum Query

private:

int \_A[MAX\_N], SpT[MAX\_N][LOG\_TWO\_N];

public:

RMQ(int n, int A[]) { // constructor as well as pre-processing routine

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

\_A[i] = A[i];

SpT[i][0] = i; // RMQ of sub array starting at index i + length 2^0=1

}

// the two nested loops below have overall time complexity = O(n log n)

for (int j = 1; (1<<j) <= n; j++) // for each j s.t. 2^j <= n, O(log n)

for (int i = 0; i + (1<<j) - 1 < n; i++) // for each valid i, O(n)

if (\_A[SpT[i][j-1]] < \_A[SpT[i+(1<<(j-1))][j-1]]) // RMQ

SpT[i][j] = SpT[i][j-1]; // start at index i of length 2^(j-1)

else // start at index i+2^(j-1) of length 2^(j-1)

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

}

int query(int i, int j) {

int k = (int)floor(log((double)j-i+1) / log(2.0)); // 2^k <= (j-i+1)

if (\_A[SpT[i][k]] <= \_A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];

else return SpT[j-(1<<k)+1][k];

} };

int main() {

// same example as in chapter 2: segment tree

int n = 7, A[] = {18, 17, 13, 19, 15, 11, 20};

RMQ rmq(n, A);

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

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

printf("RMQ(%d, %d) = %d\n", i, j, rmq.query(i, j));

return 0;

}

```

**String operation**

==

Basic C++ library functions

--

1. cin.getline(cin, S) in \<string\>

2. n = S.find(str) | (str, startingPos) | (char) returning n = -1 if not found or index (from 0)

3. std::sort (myvector.begin(), myvector.begin()+4);

4. const char\* P = s.c\_str();

**KMP**

--

```C++

#define MAX\_N 100010

char T[MAX\_N], P[MAX\_N];

int b[MAX\_N], n, m;

void kmpPreprocess() {

int i = 0, j = -1; b[0] = -1;

while (i < m) {

while (j >= 0 && P[i] != P[j]) j = b[j];

i++; j++;

b[i] = j;

}

}

void kmpSearch() {

int i = 0, j = 0;

while (i < n) {

while (j >= 0 && T[i] != P[j]) j = b[j];

i++; j++;

if (j == m) {

//found at index i - j

j = b[j];

}

}

}

```

**String Alignment**

--

```C++

#include <algorithm>

#include <cstdio>

#include <cstring>

using namespace std;

int main() {

char A[20] = "ACAATCC", B[20] = "AGCATGC";

int n = (int)strlen(A), m = (int)strlen(B);

int i, j, table[20][20]; // Needleman Wunsnch's algorithm

memset(table, 0, sizeof table);

// insert/delete = -1 point

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

table[i][0] = i \* -1;

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

table[0][j] = j \* -1;

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

for (j = 1; j <= m; j++) {

// match = 2 points, mismatch = -1 point

table[i][j] = table[i - 1][j - 1] + (A[i - 1] == B[j - 1] ? 2 : -1); // cost for match or mismatches

// insert/delete = -1 point

table[i][j] = max(table[i][j], table[i - 1][j] - 1); // delete

table[i][j] = max(table[i][j], table[i][j - 1] - 1); // insert

}

printf("DP table:\n");

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

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

printf("%3d", table[i][j]);

printf("\n");

}

printf("Maximum Alignment Score: %d\n", table[n][m]);

return 0;

}

```

Longest Common Subsequence

--

Change the weight of mismatch to - infinity, cost of delete and insert to 0 and cost of match to 1.

**Suffix Array**

--

```C++

#include <algorithm>

#include <cstdio>

#include <cstring>

using namespace std;

typedef pair<int, int> ii;

#define MAX\_N 100010 // second approach: O(n log n)

char T[MAX\_N]; // the input string, up to 100K characters

int n; // the length of input string

int RA[MAX\_N], tempRA[MAX\_N]; // rank array and temporary rank array

int SA[MAX\_N], tempSA[MAX\_N]; // suffix array and temporary suffix array

int c[MAX\_N]; // for counting/radix sort

char P[MAX\_N]; // the pattern string (for string matching)

int m; // the length of pattern string

int Phi[MAX\_N]; // for computing longest common prefix

int PLCP[MAX\_N];

int LCP[MAX\_N]; // LCP[i] stores the LCP between previous suffix T+SA[i-1]

// and current suffix T+SA[i]

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; } // compare

void countingSort(int k) { // O(n)

int i, sum, maxi = max(300, n); // up to 255 ASCII chars or length of n

memset(c, 0, sizeof c); // clear frequency table

for (i = 0; i < n; i++) // count the frequency of each integer rank

c[i + k < n ? RA[i + k] : 0]++;

for (i = sum = 0; i < maxi; i++) {

int t = c[i]; c[i] = sum; sum += t;

}

for (i = 0; i < n; i++) // shuffle the suffix array if necessary

tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];

for (i = 0; i < n; i++) // update the suffix array SA

SA[i] = tempSA[i];

}

void constructSA() { // this version can go up to 100000 characters

int i, k, r;

for (i = 0; i < n; i++) RA[i] = T[i]; // initial rankings

for (i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}

for (k = 1; k < n; k <<= 1) { // repeat sorting process log n times

countingSort(k); // actually radix sort: sort based on the second item

countingSort(0); // then (stable) sort based on the first item

tempRA[SA[0]] = r = 0; // re-ranking; start from rank r = 0

for (i = 1; i < n; i++) // compare adjacent suffixes

tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r

(RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;

for (i = 0; i < n; i++) // update the rank array RA

RA[i] = tempRA[i];

if (RA[SA[n-1]] == n-1) break; // nice optimization trick

} }

void computeLCP() {

int i, L;

Phi[SA[0]] = -1; // default value

for (i = 1; i < n; i++) // compute Phi in O(n)

Phi[SA[i]] = SA[i-1]; // remember which suffix is behind this suffix

for (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)

if (Phi[i] == -1) { PLCP[i] = 0; continue; } // special case

while (T[i + L] == T[Phi[i] + L]) L++; // L increased max n times

PLCP[i] = L;

L = max(L-1, 0); // L decreased max n times

}

for (i = 0; i < n; i++) // compute LCP in O(n)

LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the correct position

}

ii stringMatching() { // string matching in O(m log n)

int lo = 0, hi = n-1, mid = lo; // valid matching = [0..n-1]

while (lo < hi) { // find lower bound

mid = (lo + hi) / 2; // this is round down

int res = strncmp(T + SA[mid], P, m); // try to find P in suffix 'mid'

if (res >= 0) hi = mid; // prune upper half (notice the >= sign)

else lo = mid + 1; // prune lower half including mid

} // observe `=' in "res >= 0" above

if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1); // if not found

ii ans; ans.first = lo;

lo = 0; hi = n - 1; mid = lo;

while (lo < hi) { // if lower bound is found, find upper bound

mid = (lo + hi) / 2;

int res = strncmp(T + SA[mid], P, m);

if (res > 0) hi = mid; // prune upper half

else lo = mid + 1; // prune lower half including mid

} // (notice the selected branch when res == 0)

if (strncmp(T + SA[hi], P, m) != 0) hi--; // special case

ans.second = hi;

return ans;

} // return lower/upperbound as first/second item of the pair, respectively

ii LRS() { // returns a pair (the LRS length and its index)

int i, idx = 0, maxLCP = -1;

for (i = 1; i < n; i++) // O(n), start from i = 1

if (LCP[i] > maxLCP)

maxLCP = LCP[i], idx = i;

return ii(maxLCP, idx);

}

int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }

ii LCS() { // returns a pair (the LCS length and its index)

int i, idx = 0, maxLCP = -1;

for (i = 1; i < n; i++) // O(n), start from i = 1

if (owner(SA[i]) != owner(SA[i-1]) && LCP[i] > maxLCP)

maxLCP = LCP[i], idx = i;

return ii(maxLCP, idx);

}

int main() {

strcpy(T, "GATAGACA");

n = (int)strlen(T);

T[n++] = '$';

// if '\n' is read, uncomment the next line

//T[n-1] = '$'; T[n] = 0;

constructSA(); // O(n log n)

for (int i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T + SA[i]);

computeLCP(); // O(n)

// Longest Repeated Substring demo

ii ans = LRS(); // find the LRS of the first input string

char lrsans[MAX\_N];

strncpy(lrsans, T + SA[ans.second], ans.first);

printf("\nThe LRS is '%s' with length = %d\n\n", lrsans, ans.first);

// stringMatching demo

//printf("\nNow, enter a string P below, we will try to find P in T:\n");

strcpy(P, "A");

m = (int)strlen(P);

// if '\n' is read, uncomment the next line

//P[m-1] = 0; m--;

ii pos = stringMatching();

if (pos.first != -1 && pos.second != -1) {

printf("%s is found SA[%d..%d] of %s\n", P, pos.first, pos.second, T);

printf("They are:\n");

for (int i = pos.first; i <= pos.second; i++)

printf(" %s\n", T + SA[i]);

} else printf("%s is not found in %s\n", P, T);

// Longest Common Substring demo

// T already has '$' at the back

strcpy(P, "CATA");

m = (int)strlen(P);

// if '\n' is read, uncomment the next line

//P[m-1] = 0; m--;

strcat(T, P); // append P

strcat(T, "#"); // add '$' at the back

n = (int)strlen(T); // update n

// reconstruct SA of the combined strings

constructSA(); // O(n log n)

computeLCP(); // O(n)

//printf("\nThe LCP information of 'T+P' = '%s':\n", T);

//printf("i\tSA[i]\tLCP[i]\tOwner\tSuffix\n");

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

// printf("%2d\t%2d\t%2d\t%2d\t%s\n", i, SA[i], LCP[i], owner(SA[i]), T + SA[i]);

ans = LCS(); // find the longest common substring between T and P

char lcsans[MAX\_N];

strncpy(lcsans, T + SA[ans.second], ans.first);

printf("\nThe LCS is '%s' with length = %d\n", lcsans, ans.first);

return 0;

}

```

**Comments**

**Comments**