Cada implementação de algoritmo é feita de maneira separada das demais. Desta forma, funções secundárias, declarações de macros, estruturas auxiliares, etc., serão redeclaradas e reescritas para cada algoritmo sempre que necessário.

**Conteúdo**

[2D Geometry 6](#__RefHeading___Toc26732_2932636602)

[Angle between three points (90 degress or PI/2 rads) 6](#__RefHeading___Toc26734_2932636602)

[Angle between three points (180 degrees or PI rads) 7](#__RefHeading___Toc26736_2932636602)

[Area of an Ellipse 8](#__RefHeading___Toc26738_2932636602)

[Area of triangle - Heron's formula 9](#__RefHeading___Toc26740_2932636602)

[Area of a polygon 10](#__RefHeading___Toc26742_2932636602)

[Area of a regular polygon 11](#__RefHeading___Toc26744_2932636602)

[Area of a Triangle - 3 points 12](#__RefHeading___Toc26746_2932636602)

[Area of the union of Rectangles 13](#__RefHeading___Toc26748_2932636602)

[CCW Orientation analysis 14](#__RefHeading___Toc26750_2932636602)

[Centroid of a Polygon 16](#__RefHeading___Toc26752_2932636602)

[Parameter of circle from 3 points 17](#__RefHeading___Toc26754_2932636602)

[Circle tangents 19](#__RefHeading___Toc26756_2932636602)

[Closest pair of points 20](#__RefHeading___Toc26758_2932636602)

[Closest Point on Infinite Line 22](#__RefHeading___Toc26760_2932636602)

[Closest Point on a Line Segment 23](#__RefHeading___Toc26762_2932636602)

[Convex Hull (minimizing version) 25](#__RefHeading___Toc26764_2932636602)

[Distance between two points in 2D 28](#__RefHeading___Toc26766_2932636602)

[Distance from a point to an infinte line 29](#__RefHeading___Toc26768_2932636602)

[Circle Intersection Area 30](#__RefHeading___Toc26770_2932636602)

[Circle-Line Intersection 32](#__RefHeading___Toc26772_2932636602)

[Circle Intersection Points 34](#__RefHeading___Toc26774_2932636602)

[Circle Intersection Type 36](#__RefHeading___Toc26776_2932636602)

[Infinite Line Intersection 38](#__RefHeading___Toc26778_2932636602)

[Line Segment Intersection 39](#__RefHeading___Toc26780_2932636602)

[Area of intersection between two general polygons 41](#__RefHeading___Toc26782_2932636602)

[Laser Reflections 46](#__RefHeading___Toc26784_2932636602)

[Lattice-polygons - Pick's Theorem 49](#__RefHeading___Toc26786_2932636602)

[Largest rectangle not enclosing points 50](#__RefHeading___Toc26788_2932636602)

[Polygon Midpoints to Verticies 52](#__RefHeading___Toc26790_2932636602)

[Minimum bounding circle 54](#__RefHeading___Toc26792_2932636602)

[Point in convex polygon 56](#__RefHeading___Toc26794_2932636602)

[Polygon Similarity & Congruency 57](#__RefHeading___Toc26796_2932636602)

[Generalized Equation of a line given two points 60](#__RefHeading___Toc26798_2932636602)

[Point in Polygon Test 61](#__RefHeading___Toc26800_2932636602)

[Point Left/Right/In Line test 62](#__RefHeading___Toc26802_2932636602)

[Rectangle in Rectangle Test 64](#__RefHeading___Toc26804_2932636602)

[Reflect a point across a line 66](#__RefHeading___Toc26806_2932636602)

[Rotation of a point around a point 67](#__RefHeading___Toc26808_2932636602)

[Soddy Circle Radius 69](#__RefHeading___Toc26810_2932636602)

[Triangulation of a simple 2D polygon 69](#__RefHeading___Toc26812_2932636602)

[3D Geometry 73](#__RefHeading___Toc26814_2932636602)

[Area of a convex polygon in a plane given in 3D 73](#__RefHeading___Toc26816_2932636602)

[Geometry Primitives 74](#__RefHeading___Toc26818_2932636602)

[Closest Point in Line 76](#__RefHeading___Toc26820_2932636602)

[Closest Point in Line Segment 76](#__RefHeading___Toc26822_2932636602)

[Closest point to plane represented by vetor normal and a other point 76](#__RefHeading___Toc26824_2932636602)

[Closest point to plane represented by three others points 76](#__RefHeading___Toc26826_2932636602)

[Find the intersection points of a infinite line and a sphere 77](#__RefHeading___Toc26828_2932636602)

[Translate a 3D problem with planes in a 2d problem 77](#__RefHeading___Toc26830_2932636602)

[Given two lines in 3D space, find distance of closest approach 78](#__RefHeading___Toc26832_2932636602)

[Given two lines in 3D space, find distance of closest approach and returns the points of each line 78](#__RefHeading___Toc26834_2932636602)

[Is the point p on the infinite line ab? 78](#__RefHeading___Toc26836_2932636602)

[Is the point p on the segment ab? 78](#__RefHeading___Toc26838_2932636602)

[Given a plane and a line ab, determine if the two intersect, and if so, find the single point of intersection 79](#__RefHeading___Toc26840_2932636602)

[Great Circle 79](#__RefHeading___Toc26842_2932636602)

[Sphere from 4 Points 80](#__RefHeading___Toc26844_2932636602)

[Solves the determinant of a n by n matrix recursively 81](#__RefHeading___Toc26846_2932636602)

[Sphere from four points 81](#__RefHeading___Toc26848_2932636602)

[Arithmetic 84](#__RefHeading___Toc26850_2932636602)

[Discrete Logarithm solver 84](#__RefHeading___Toc26852_2932636602)

[Fast Exponentition 85](#__RefHeading___Toc26854_2932636602)

[Fast Exponentiation mod m 86](#__RefHeading___Toc26856_2932636602)

[Simpson's Rule for Numerical Intergration 87](#__RefHeading___Toc26858_2932636602)

[Cubic equation solver 88](#__RefHeading___Toc26860_2932636602)

[Quadratic equation solver 90](#__RefHeading___Toc26862_2932636602)

[Coupons Problem 91](#__RefHeading___Toc26864_2932636602)

[Combinatorics 94](#__RefHeading___Toc26866_2932636602)

[Binomial coefficient 94](#__RefHeading___Toc26868_2932636602)

[Digit Occurence count 95](#__RefHeading___Toc26870_2932636602)

[Digits in N! (N factorial) 96](#__RefHeading___Toc26872_2932636602)

[Josephus Ring Survivor 97](#__RefHeading___Toc26874_2932636602)

[Necklace/Bracelet Enumeration 98](#__RefHeading___Toc26876_2932636602)

[Permutation index on distinct characters 100](#__RefHeading___Toc26878_2932636602)

[Dynamic Programming 102](#__RefHeading___Toc26880_2932636602)

[Longest Ascending Subsequence – O(nlogn) 102](#__RefHeading___Toc26882_2932636602)

[Longest ascending subsequence - O(N^2) 104](#__RefHeading___Toc26884_2932636602)

[Edit Distance (with path recovery) 106](#__RefHeading___Toc26886_2932636602)

[Integer Parititoning 109](#__RefHeading___Toc26888_2932636602)

[Longest Common Subsequence 110](#__RefHeading___Toc26890_2932636602)

[Maximum Submatrix Sum 112](#__RefHeading___Toc26892_2932636602)

[Generators 115](#__RefHeading___Toc26894_2932636602)

[Catalan Numbers 115](#__RefHeading___Toc26896_2932636602)

[Binary Strings generator - (ordered by cardinality) 116](#__RefHeading___Toc26898_2932636602)

[K-Subset functions 117](#__RefHeading___Toc26900_2932636602)

[Pythagorean Triples 123](#__RefHeading___Toc26902_2932636602)

[Graph Theory 125](#__RefHeading___Toc26904_2932636602)

[Articulation Points in a Graph (Adj List version) 125](#__RefHeading___Toc26906_2932636602)

[Bellman-Ford Algorithm - negative weigth cycles 127](#__RefHeading___Toc26908_2932636602)

[Maximum Bipartite Matching 129](#__RefHeading___Toc26910_2932636602)

[Articulation Points & Bridges (adj list) 131](#__RefHeading___Toc26912_2932636602)

[Eulerian Graphs 134](#__RefHeading___Toc26914_2932636602)

[Directed Eulerian Graphs 138](#__RefHeading___Toc26916_2932636602)

[Floyd's Algorithm - All pairs shortest path 142](#__RefHeading___Toc26918_2932636602)

[Maximum Flow in a directed graph 143](#__RefHeading___Toc26920_2932636602)

[Maximum Flow with Lower bounds on edges 147](#__RefHeading___Toc26922_2932636602)

[Maximum Flow in a directed graph 151](#__RefHeading___Toc26924_2932636602)

[Minimum disjoint-vertex path cover on an acyclic directed graph 154](#__RefHeading___Toc26926_2932636602)

[Minimum Spanning Tree (version 2) 157](#__RefHeading___Toc26928_2932636602)

[Chinese Postman Problem 159](#__RefHeading___Toc26930_2932636602)

[K-cycle in a directed tournament graph 161](#__RefHeading___Toc26932_2932636602)

[Stable Marriage 164](#__RefHeading___Toc26934_2932636602)

[Strongly Connected Components 166](#__RefHeading___Toc26936_2932636602)

[Topological Sort 168](#__RefHeading___Toc26938_2932636602)

[Tree Isomorphism by certificates 170](#__RefHeading___Toc26940_2932636602)

[Maximum Weighted Bipartite Matching Combinatorics: Assignment Problem 172](#__RefHeading___Toc26942_2932636602)

[Miscellaneous 176](#__RefHeading___Toc26944_2932636602)

[Bit Count 176](#__RefHeading___Toc26946_2932636602)

[Roman Numerals 177](#__RefHeading___Toc26948_2932636602)

[Rubik's Cube simulator 178](#__RefHeading___Toc26950_2932636602)

[Square index rotation/flipping 181](#__RefHeading___Toc26952_2932636602)

[String Reverse 182](#__RefHeading___Toc26954_2932636602)

[Uniquely Decipherable Codes 183](#__RefHeading___Toc26956_2932636602)

[Uniquely Decipherable Codes Test 185](#__RefHeading___Toc26958_2932636602)

[Number Theory 187](#__RefHeading___Toc26960_2932636602)

[GCD of binary numbers (Algorithm B) 187](#__RefHeading___Toc26962_2932636602)

[Euler Phi function 188](#__RefHeading___Toc26964_2932636602)

[Farey Sequence Generator 189](#__RefHeading___Toc26968_2932636602)

[Parsing 192](#__RefHeading___Toc26974_2932636602)

[Evaluating Arithmetic Expressions 192](#__RefHeading___Toc26976_2932636602)

[Converting infix to postfix 197](#__RefHeading___Toc26978_2932636602)

[Arithmetic Expression Parsing Template 198](#__RefHeading___Toc26980_2932636602)

[Search 201](#__RefHeading___Toc26982_2932636602)

[Binary Search Template 201](#__RefHeading___Toc26984_2932636602)

[(r-g)-Cage search template 203](#__RefHeading___Toc26986_2932636602)

[Graph coloring template 206](#__RefHeading___Toc26988_2932636602)

[Golden section Search 207](#__RefHeading___Toc26990_2932636602)

[Hamming Code search template / (Independent set) 209](#__RefHeading___Toc26992_2932636602)

[Orthogonal-Latin Squares 211](#__RefHeading___Toc26994_2932636602)

[N non-attacking Queens 213](#__RefHeading___Toc26996_2932636602)

[Suffix array 214](#__RefHeading___Toc26998_2932636602)

[Self Avoiding walk template 218](#__RefHeading___Toc27000_2932636602)

[Word Search 219](#__RefHeading___Toc27002_2932636602)

############################################

# 2D Geometry

############################################

*/\* 2D-Geometry:*

## Angle between three points (90 degress or PI/2 rads)

*=================================================================*

*Description: Given 3 points A, B, C, it returns the magnitude of*

*angle ABC in radians.*

*Will only return the angle 0 <= theta <= PI/2.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Nov 14, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Uses vectors to calculate the angle:*

*a . b = |a| \* |b| \* cos(theta) -->*

*theta = acos(a . b / (|a| \* |b|))*

*\*/*

#include <stdio.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

**double** angle2d(Point a, Point b, Point c){

**double** dx1 = a.x - b.x, dy1 = a.y - b.y;

**double** dx2 = c.x - b.x, dy2 = c.y - b.y;

**double** dot = dx1 \* dx2 + dy1 \* dy2;

**double** l1 = sqrt(SQR(dx1)+SQR(dy1));

**double** l2 = sqrt(SQR(dx2)+SQR(dy2));

**return** acos(dot / (l1\*l2));

}

**int** main(){

Point a, b, c;

printf("Enter A, B and C:\n");

**while**(scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y) == 6){

printf("Angle ABC in radians = %f\n", angle2d(a, b, c));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Angle between three points (180 degrees or PI rads)

*=================================================================*

*Description: Given 3 points A, B, C, it returns the magnitude of*

*angle ABC in radians.*

*Will only return the angle 0 <= theta <= PI.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Scott Crosswhite*

*Date: October 19, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Uses the Cosine Law, c^2 = a^2 + b^2 - 2ab cos C.*

*Presumes that d(A,B) and d(B,C) are non-zero.*

*\*/*

#include <iostream>

#include <cstdio>

#include <cmath>

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

**double** dist\_2d (Point a, Point b) {

**return** sqrt(SQR(a.x-b.x)+SQR(a.y-b.y));

}

**double** angle2d180 (Point A, Point B, Point C) {

**double** a = dist\_2d(A,B), b = dist\_2d(B,C), c = dist\_2d(A,C);

**return** acos((SQR(c)-SQR(a)-SQR(b)) / (-2\*a\*b));

}

**int** main () {

Point a, b, c;

printf("Enter A, B, C:\n");

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y) == 6) {

printf("Angle ABC in radians = %f\n", angle2d180(a,b,c));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Area of an Ellipse

*=================================================================*

*Description: Given the quadratic form of an ellipse:*

*ax^2 + bxy + cy^2 = 1*

*returns the area of the ellipse*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 7, 2002*

*References: http://mathworld.wolfram.com/Ellipse.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

**double** PI;

**double** area\_gen\_ellipse(**double** a, **double** b, **double** c){

**return** 2\*PI / (sqrt(4\*a\*c-b\*b));

}

**int** main(){

**double** a, b, c;

PI = acos(-1);

**while**(scanf("%lf %lf %lf", &a, &b, &c) == 3){

printf("Area of Ellipse is %f.\n", area\_gen\_ellipse(a,b,c));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Area of triangle - Heron's formula

*=================================================================*

*Description: Given the three lengths of a triangle, returns*

*the area of the triangle.*

*Returns -1 if triangle does not exist*

*Complexity: O(1) (square root)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee, Ashley Zinyk*

*Date: Sept 7, 2002*

*References: http://mathworld.wolfram.com/Area.html*

*-----------------------------------------------------------------*

*Reliability: 1 (Sept 2002)*

*Notes: If you deal with needle-shaped triangles, you might have*

*problems with numerical inaccuracy. In that case, try*

*area\_heron2, which is more stable but hard to type in.*

*\*/*

#include <stdio.h>

#include <math.h>

*/\**

*double area\_heron2(double a, double b, double c) {*

*double t;*

*if (a < b) { t=a; a=b; b=t; }*

*if (a < c) { t=a; a=c; c=t; }*

*if (b < c) { t=b; b=c; c=t; }*

*if ((c-(a-b)) < 0.0) return -1;*

*return sqrt((a+(b+c))\*(c-(a-b))\*(c+(a-b))\*(a+(b-c)))/4.0;*

*} \*/*

**double** area\_heron(**double** a, **double** b, **double** c){

**double** s = (a+b+c)/2.0;

**if**(a > s || b > s || c > s) **return** -1;

**return** sqrt(s\*(s-a)\*(s-b)\*(s-c));

}

**int** main(){

**double** a, b, c;

**while**(scanf("%lf %lf %lf", &a, &b, &c) == 3){

printf("Area of triangle = %f\n", area\_heron(a,b,c));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Area of a polygon

*=================================================================*

*Description: Given a polygon represented as an array of N points,*

*returns the signed area. The result is positive if*

*the orientation is counterclockwise, and negative*

*otherwise.*

*Complexity: O(N) N being the number of vertices of the polygon*

*-----------------------------------------------------------------*

*Author: Howard Cheng, Gilbert Lee*

*Date: Sept 2002*

*References: www.exaflop.org/docs/cgafaq/cga2.html*

*-----------------------------------------------------------------*

*Reliability: 0 (Sept 2002)*

*Notes: Remember to run fabs() on the result if a positive*

*area is desired.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

**typedef** **struct** {

**double** x, y;

} Point;

**double** area\_poly(Point \*p, **int** n){

**double** sum = 0;

**int** i, j;

**for**(i = n-1, j = 0; j < n; i = j++)

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

p[i].y \* p[j].x;

**return** sum/2.0;

}

**int** main(){

Point \*poly;

**int** i, n;

**while**(scanf("%d", &n) == 1 && n > 0){

poly = (Point \*) malloc(n \* **sizeof**(Point));

**for**(i = 0; i < n; i++) scanf("%lf %lf", &poly[i].x, &poly[i].y);

printf("Signed Area = %f\n", area\_poly(poly, n));

free(poly);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Area of a regular polygon

*=================================================================*

*Description: Returns the area of a regular n-gon with side*

*length s.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 7, 2002*

*References: http://mathworld.wolfram.com/Area.html*

*-----------------------------------------------------------------*

*Reliability: 0 (Sept 2002)*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

**double** PI;

**double** area\_regpoly(**int** n, **double** s){

**return** n < 3 ? 0 : n\*s\*s/tan(PI/n)/4;

}

**int** main(){

**int** n;

**double** s;

PI = acos(-1);

**while**(scanf("%d %lf", &n, &s) == 2)

printf("Regular %d-gon has area: %f\n", n, area\_regpoly(n,s));

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Area of a Triangle - 3 points

*=================================================================*

*Description: Given three vertices a triangle, returns the area*

*of the triangle.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 7, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

**typedef** **struct**{

**double** x, y;

} Point;

**double** area\_tri(Point a, Point b, Point c){

**double** area;

area = (b.x-a.x)\*(c.y-a.y)-

(b.y-a.y)\*(c.x-a.x);

**return** (fabs(area))/2;

}

**int** main(){

Point a, b, c;

**while**(scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y) == 6){

printf("Area of triangle = %f\n", area\_tri(a,b,c));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Area of the union of Rectangles

*=================================================================*

*Description: Given a set of N rectangles, determines the area of*

*it's union.*

*Complexity: O(N^2)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 8, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 successful use (Spain Problem 688) Sept 2002*

*Notes: Scan line technique - For each y-line, it determines*

*how much is inside the polygon, thus computing area*

*in rectangular blocks*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#define MAXN 200

**typedef** **struct**{

**double** minx, miny, maxx, maxy;

} Rect;

**typedef** **struct**{

**double** x, miny, maxy;

**char** m;

} Edge;

Rect r[MAXN];

**double** ys[2\*MAXN];

Edge e[2\*MAXN];

**int** cmp\_double(**double** \*a, **double** \*b){

**return** \*a < \*b ? -1 : \*a > \*b ? 1 : 0;

}

**int** cmp\_edge(Edge \*a, Edge \*b){

**return** a->x < b->x ? -1 : a->x > b->x ? 1 : 0;

}

**double** area\_unionrect(**int** n){

**int** flag, i, j;

**double** curr, sum, sx;

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

e[2\*i].miny = e[2\*i+1].miny = ys[2\*i] = r[i].miny;

e[2\*i].maxy = e[2\*i+1].maxy = ys[2\*i+1] = r[i].maxy;

e[2\*i].x = r[i].minx;

e[2\*i].m = 1;

e[2\*i+1].x = r[i].maxx;

e[2\*i+1].m = -1;

}

qsort(ys, 2\*n, **sizeof**(ys[0]), (**void** \*)cmp\_double);

qsort(e, 2\*n, **sizeof**(e[0]), (**void** \*)cmp\_edge);

sum = 0;

**for**(i = 0; i < 2\*n; i++){

**if**(i) sum += (ys[i]-ys[i-1])\*curr;

curr = flag = 0;

**for**(j = 0; j < 2\*n; j++){

**if**(e[j].miny <= ys[i] && ys[i] < e[j].maxy){

**if**(!flag) sx = e[j].x;

flag += e[j].m;

**if**(!flag) curr += e[j].x-sx;

}

}

}

**return** sum;

}

**int** main(){

**int** i, n;

**while**(scanf("%d", &n) == 1){

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

scanf("%lf %lf %lf %lf",

&r[i].minx, &r[i].miny, &r[i].maxx, &r[i].maxy);

}

printf("Area of union is %f\n", area\_unionrect(n));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## CCW Orientation analysis

*=================================================================*

*Description: Given three points a, b, c, it returns whether the*

*path from a to b to c is counterclockwise, clockwise*

*or undefined.*

*Undefined is returned if the 3 points are colinear,*

*and c is between a and b.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Howard Cheng, Scott Crosswhite, Gilbert Lee*

*Date: Nov 13, 2002*

*References: wilma.cs.brown.edu/courses/cs016/packet/node18.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Colinearity with respect to the line through a, b,*

*and c:*

*CCW for c in [b, +inf)*

*CNEITHER for c in [a,b)*

*CW for c in (-inf, a)*

*For true CW/CCW/Collinear testing, consider using*

*pt\_leftright.c*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

*/\* how close to call equal \*/*

#define EPS 1E-8

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

*/\* counterclockwise, clockwise, or undefined \*/*

**enum** {CCW, CW, CNEITHER};

**int** ccw(Point a, Point b, Point c)

{

**double** dx1 = b.x - a.x, dx2 = c.x - b.x;

**double** dy1 = b.y - a.y, dy2 = c.y - b.y;

**double** t1 = dx1 \* dy2;

**double** t2 = dx2 \* dy1;

**if**(fabs(t1 - t2) < EPS) {

**if**(dx1 \* dx2 < 0 || dy1 \* dy2 < 0) {

**if**(SQR(dx1)+SQR(dy1) >= SQR(dx2)+SQR(dy2)- EPS) **return** CNEITHER;

**return** CW;

}

**return** CCW;

}

**return** t1 > t2 ? CCW : CW;

}

**int** main(**void**)

{

Point a, b, c;

**int** a1, a2, a3, a4, a5, a6;

**int** res;

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y) == 6){

res = ccw(a,b,c);

**if** (res == CW) {

printf("CW\n");

} **else** **if** (res == CCW) {

printf("CCW\n");

} **else** **if** (res == CNEITHER) {

printf("CNEITHER\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Centroid of a Polygon

*=================================================================*

*Description: Given a polygon as a set of N points, this code finds*

*the centroid of the polygon.*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Nov 8, 2003*

*References: http://www.netcomuk.co.uk/~jenolive/centroid.html*

*www.exaflop.org/docs/cgafaq/cga2.html*

*see also area\_poly.c*

*-----------------------------------------------------------------*

*Reliability: 2 (Spain 10002)*

*(Center of Mass Nov 2003)*

*Notes: Points must be in clockwise or counterclockwise order.*

*The area of the polygon must be non-zero.*

*The polygon must not be self-intersecting (simple)*

*The polygon may or may not be convex.*

*\*/*

#include <stdio.h>

#include <math.h>

**typedef** **struct** {

**double** x, y;

} Point;

Point centroid(Point \*p, **int** n) {

**double** area, sum;

Point c;

**int** i, j;

c.x = c.y = sum = 0.0;

**for**(i = n-1, j = 0; j < n; i = j++) {

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

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

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

}

sum \*= 3.0;

c.x /= sum;

c.y /= sum;

**return** c;

}

**int** main() {

Point poly[100], c;

**int** i, n;

**while** (scanf("%d",&n)==1) {

**if** (n < 3) **break**;

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

scanf("%lf %lf",&poly[i].x, &poly[i].y);

c = centroid(poly,n);

printf("%0.3f %0.3f\n",c.x,c.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Parameter of circle from 3 points

*=================================================================*

*Description: This routine computes the parameters of a circle*

*(center and radius) from 3 points.*

*Returns non-zero if successful, zero if the three*

*points are collinear*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Howard Cheng*

*Date: Nov 13, 2002*

*References: www.exaflop.org/docs/cgafaq/*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

*/\* how close to call equal \*/*

#define EPS 1E-8

**typedef** **struct** {

**double** x, y;

} Point;

**int** circle(Point p1, Point p2, Point p3, Point \*center, **double** \*r)

{

**double** a,b,c,d,e,f,g;

a = p2.x - p1.x; b = p2.y - p1.y;

c = p3.x - p1.x; d = p3.y - p1.y;

e = a\*(p1.x + p2.x) + b\*(p1.y + p2.y);

f = c\*(p1.x + p3.x) + d\*(p1.y + p3.y);

g = 2.0\*(a\*(p3.y - p2.y) - b\*(p3.x - p2.x));

**if** (fabs(g) < EPS) **return** 0;

center->x = (d\*e - b\*f) / g;

center->y = (a\*f - c\*e) / g;

\*r = sqrt((p1.x-center->x)\*(p1.x-center->x) +

(p1.y-center->y)\*(p1.y-center->y));

**return** 1;

}

**int** main(**void**)

{

Point a, b, c, center;

**double** r;

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y) == 6) {

**if** (circle(a, b, c, &center, &r)) {

printf("center = (%5.3f, %5.3f)\n", center.x, center.y);

printf("radius = %5.3f\n", r);

printf("\n");

} **else** {

printf("colinear\n\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Circle tangents

*=================================================================*

*Description: Given a circle (defined by a center point and radius)*

*and a point strictly outside the circle, returns the*

*two points of tangency.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Nov 19, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Assumes a non-zero distance between p and c.*

*\*/*

#include <stdio.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

**double** dist2(Point a, Point b) {

**return** SQR(a.x-b.x) + SQR(a.y-b.y);

}

Point a, b;

**void** circ\_tangents(Point c, **double** r, Point p) {

**double** perp, para, tmp = dist2(p,c);

para = r\*r/tmp;

perp = r\*sqrt(tmp-r\*r)/tmp;

a.x = c.x + (p.x-c.x)\*para - (p.y-c.y)\*perp;

a.y = c.y + (p.y-c.y)\*para + (p.x-c.x)\*perp;

b.x = c.x + (p.x-c.x)\*para + (p.y-c.y)\*perp;

b.y = c.y + (p.y-c.y)\*para - (p.x-c.x)\*perp;

}

**int** main() {

Point c, p;

**double** rad;

**while** (scanf("%lf %lf %lf", &c.x, &c.y, &rad)==3) {

scanf("%lf %lf",&p.x,&p.y);

circ\_tangents(c, rad, p);

printf("%g,%g %g,%g\n",a.x,a.y,b.x,b.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Closest pair of points

*=================================================================*

*Description: Given a set of N points, finds two points A and B*

*such that the distance between them is no greater*

*than the distance between any other pair of points*

*in the set.*

*Complexity: O(N lg N lg N)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: December 10, 2002*

*References: www.csc.liv.ac.uk/~ped/teachadmin/algor/d\_and\_c.html*

*students.cec.wustl.edu/~cs241/handouts/closestpair.pdf*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 10245)*

*Notes: Call the function as shown in main().*

*The original array of points will be sorted.*

*The points are returned as globals. N must be >= 2.*

*- min stores the SQUARE of the minimum distance*

*The traditional solution to this problem is*

*O(N lg N), but it is about 80% more code and*

*200% more difficult to understand.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define SQR(a) ((a)\*(a))

**typedef** **struct** {

**double** x, y;

} Point;

Point mina, minb; */\* globals \*/*

**double** min;

**void** checkpair(Point a, Point b) {

**double** dist2 = SQR(b.x-a.x)+SQR(b.y-a.y);

**if** (dist2 < min) {

min = dist2;

mina = a; minb = b;

}

}

**int** xcmp(Point \*a, Point \*b) {

**if** (a->x < b->x) **return** -1;

**if** (a->x > b->x) **return** 1;

**if** (a->y < b->y) **return** -1;

**if** (a->y > b->y) **return** 1;

**return** 0;

}

*/\* copy and paste xcmp when typing this in \*/*

**int** ycmp(Point \*a, Point \*b) {

**if** (a->y < b->y) **return** -1;

**if** (a->y > b->y) **return** 1;

**if** (a->x < b->x) **return** -1;

**if** (a->x > b->x) **return** 1;

**return** 0;

}

**void** closest\_pair(Point \*p, **int** n) {

Point \*q;

**int** i, j, cnt;

**if** (n==2) checkpair(p[0],p[1]); */\* base cases \*/*

**if** (n<=2) **return**;

closest\_pair(p,n/2); */\* do each half \*/*

closest\_pair(&p[n/2],n-n/2);

q = malloc(n\***sizeof**(Point)); */\* do the middle \*/*

**for** (cnt = i = 0; i < n; i++)

**if** (SQR(p[i].x - p[n/2].x) < min) q[cnt++] = p[i];

qsort(q,cnt,**sizeof**(Point),(**void** \*)ycmp);

**for** (i = 0; i < cnt; i++) {

**if** (SQR(q[i].x - p[n/2].x) >= min) **break**;

**for** (j = i+1; j < cnt; j++) {

**if** (SQR(q[i].y - q[j].y) >= min) **break**;

checkpair(q[i],q[j]);

}

}

free(q);

}

**int** main() {

**int** n, i;

Point p[1000];

**while** (scanf("%d",&n)==1) {

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

scanf("%lf %lf", &p[i].x, &p[i].y);

qsort(p, n, **sizeof**(Point),(**void** \*) xcmp);

min = DBL\_MAX; */\* DBL\_MAX is in limits.h \*/*

*/\* You can also use a value guaranteed to be*

*greater than the greatest distance, squared \*/*

closest\_pair(p, n);

printf("(%lf,%lf) (%lf,%lf)\n", mina.x, mina.y, minb.x, minb.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Closest Point on Infinite Line

*=================================================================*

*Description: Given a point C, and an infinite line represented by*

*points A and B, returns the point P on the line*

*closest to C. (i.e. CP is orthogonal to AB)*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Nov 9, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Assumes that points A and B are not the same.*

*\*/*

#include <stdio.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**double** x, y;

} Point;

Point closest\_pt\_iline(Point a, Point b, Point c) {

Point p;

**double** dp;

b.x -= a.x;

b.y -= a.y;

dp = (b.x\*(c.x-a.x) + b.y\*(c.y-a.y)) / (SQR(b.x)+SQR(b.y));

p.x = b.x\*dp + a.x;

p.y = b.y\*dp + a.y;

**return** p;

}

**int** main() {

Point a, b, c, p;

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y) == 6){

p = closest\_pt\_iline(a,b,c);

printf("%g %g\n", p.x, p.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Closest Point on a Line Segment

*=================================================================*

*Description: Given the end points of a line segment, A and B, and*

*another point C, returns the point on the segment*

*closest to C.*

*If a line perpendicular to A,B intersects A, B then*

*intersection will be returned. Otherwise the closer*

*endpoint will be returned.*

*If the segment has a length of zero, an endpoint*

*will be returned.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Nov 9, 2002*

*References: 0*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 10263 - Railway)*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

#define EPS 1E-8

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

Point closest\_pt\_lineseg(Point a, Point b, Point c) {

Point p;

**double** dp;

b.x -= a.x;

b.y -= a.y;

**if** (fabs(b.x) < EPS && fabs(b.y) < EPS) **return** a;

dp = (b.x\*(c.x-a.x) + b.y\*(c.y-a.y))/(SQR(b.x)+SQR(b.y));

**if** (dp > 1) dp = 1;

**if** (dp < 0) dp = 0;

p.x = b.x\*dp + a.x;

p.y = b.y\*dp + a.y;

**return** p;

}

**int** main() {

Point a, b, c, d;

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y)==6) {

d = closest\_pt\_lineseg(a,b,c);

printf("%g %g\n", d.x, d.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Computational Geometry:*

## Convex Hull (minimizing version)

*=================================================================*

*Description:*

*Given a list of n (n >= 1) points in an array, this routine*

*returns the vertices of the convex hull in counterclockwise*

*order. It also returns the number of vertices in the convex*

*hull.*

*The convex hull is the smallest convex polygon that encompasses*

*or borders all of the points.*

*Complexity: O(n lg n) -> O(qsort) + O(n)*

*-----------------------------------------------------------------*

*Author: Jason Klaus*

*Date: Oct 25, 2002*

*References: Based on Graham's Scan, pg.949-955,*

*Introduction to Algorithms (2nd Edition),*

*Cormen, Leiserson, Rivest, Stein*

*-----------------------------------------------------------------*

*Reliability: 2 Spain successes with older version.*

*0 successes with current version.*

*Notes:*

*- The hull contains a minimum number of points; all co-linear*

*points and non-distinct points are eliminated from the hull.*

*The reference algorithm does not do this, and also has a*

*restriction that the points must be distinct and not duplicate*

*polar angles with respects to the reference. These restrictions*

*have been removed from this implementation.*

*- The points in the original polygon will be re-ordered.*

*- The hull array must be allocated and contain the required amount*

*of space (n in size is sufficient).*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <assert.h>

#define EPSILON 1E-10

**typedef** **struct** {

**double** x, y;

} Point;

Point \*P0;

**enum** {CCW, CW, CL};

**int** cross\_prod(Point \*p1, Point \*p2, Point \*p0)

{

**double** res, x1, x2, y1, y2;

x1 = p1->x - p0->x;

x2 = p2->x - p0->x;

y1 = p1->y - p0->y;

y2 = p2->y - p0->y;

res = x1\*y2 - x2\*y1;

**if** (fabs(res) < EPSILON)

**return** CL;

**else** **if** (res > 0.0)

**return** CW;

**else**

**return** CCW;

}

**int** polar\_cmp(Point \*p1, Point \*p2)

{

**int** res;

**double** d, x1, x2, y1, y2;

res = cross\_prod(p1, p2, P0);

**if** (res == CW)

**return** -1;

**else** **if** (res == CCW)

**return** 1;

**else** {

x1 = p1->x - P0->x;

x2 = p2->x - P0->x;

y1 = p1->y - P0->y;

y2 = p2->y - P0->y;

d = ((x1\*x1) + (y1\*y1)) - ((x2\*x2) + (y2\*y2));

**if** (fabs(d) < EPSILON)

**return** 0;

**else** **if** (d < 0.0)

**return** -1;

**else**

**return** 1;

}

}

**int** convex\_hull(Point \*poly, **int** n, Point \*hull)

{

**int** i, min, h;

**if** (n < 1)

**return** 0;

min = 0;

P0 = &hull[0];

\*P0 = poly[0];

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

**if** ((poly[i].y < P0->y) ||

((poly[i].y == P0->y) && (poly[i].x < P0->x))) {

min = i;

\*P0 = poly[i];

}

}

poly[min] = poly[0];

poly[0] = \*P0;

h = 1;

**if** (n == 1)

**return** h;

qsort(poly+1, n-1, **sizeof**(poly[1]),

(**int** (\*)(**const** **void** \*, **const** **void** \*))polar\_cmp);

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

**if** ((fabs(poly[i].x - hull[0].x) > EPSILON) ||

(fabs(poly[i].y - hull[0].y) > EPSILON)) {

**break**;

}

}

**if** (i == n)

**return** h;

hull[h++] = poly[i++];

**for** ( ; i < n; i++) {

**while** ((h > 1) &&

(cross\_prod(&poly[i], &hull[h-1], &hull[h-2]) != CCW)) {

h--;

}

hull[h++] = poly[i];

}

**return** h;

}

**int** main(**void**)

{

Point \*polygon, \*hull;

**int** n, hull\_size;

**int** i;

**while** (scanf("%d", &n) == 1 && n > 0) {

polygon = (Point \*)malloc(n \* **sizeof**(Point));

hull = (Point \*)malloc(n \* **sizeof**(Point));

assert(polygon && hull);

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

scanf("%lf %lf", &polygon[i].x, &polygon[i].y);

}

hull\_size = convex\_hull(polygon, n, hull);

printf("Sorted:\n");

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

printf("(%4.2f, %4.2f) ", polygon[i].x, polygon[i].y);

}

printf("\n");

printf("Hull size = %d\n", hull\_size);

**for** (i = 0; i < hull\_size; i++) {

printf("(%4.2f, %4.2f) ", hull[i].x, hull[i].y);

}

printf("\n");

free(polygon);

free(hull);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Distance between two points in 2D

*=================================================================*

*Description: Computes the distance between two points*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 8, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**double** x, y;

} Point;

**double** dist\_2d(Point a, Point b){

**return** sqrt(SQR(a.x-b.x)+SQR(a.y-b.y));

}

**int** main(){

Point a, b;

**while**(scanf("%lf %lf %lf %lf", &a.x, &a.y, &b.x, &b.y) == 4){

printf("Distance = %f\n", dist\_2d(a,b));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Distance from a point to an infinte line

*=================================================================*

*Description: Calculates the minimum(orthogonal) distance from a*

*point P to an infinite line. The line is defined*

*by two points A and B.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Howard Cheng, Gilbert Lee*

*Date: Sept 8, 2002*

*References: http://www.exaflop.org/docs/cgafaq/cga1.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Assumes that point A != point B, otherwise the line*

*would not be properly defined -> returns dist=nan*

*\*/*

#include <stdio.h>

#include <math.h>

**typedef** **struct**{

**double** x,y;

} Point;

#define SQR(x) ((x)\*(x))

**double** dist\_2d(Point a, Point b){

**return** sqrt(SQR(a.x-b.x)+SQR(a.y-b.y));

}

**double** dist\_iline(Point a, Point b, Point p){

**return** fabs(((a.y-p.y)\*(b.x-a.x)-

(a.x-p.x)\*(b.y-a.y))

/dist\_2d(a,b));

}

**int** main(){

Point a, b, p;

**while**(scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &p.x, &p.y) == 6){

printf("Distance = %f\n", dist\_iline(a,b,p));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Circle Intersection Area

*=================================================================*

*Description: Computes the area of intersection of two circles*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Scott Crosswhite*

*Date: Oct 24, 2002*

*References: http://hades.ph.tn.tudelft.nl/Internal/...*

*.../Mathworld/math/math/c/c308.htm*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define EPSILON 1e-8

#define SQR(x) ((x)\*(x))

**double** PI;

**typedef** **struct** {

**double** x, y;

} Point;

**typedef** **struct** {

Point o;

**double** r;

} Circle;

**double** CIArea (Circle A, Circle B) {

**double** d, dA, dB, tx, ty;

d = sqrt(SQR(B.o.x - A.o.x) + SQR(B.o.y - A.o.y));

**if** ((d < EPSILON) || (d + A.r <= B.r) || (d + B.r <= A.r))

**return** SQR((B.r < A.r) ? B.r : A.r) \* PI;

**if** (d >= A.r + B.r)

**return** 0;

dA = tx = (SQR(d) + SQR(A.r) - SQR(B.r))/d/2;

ty = sqrt(SQR(A.r) - SQR(tx));

dB = d - dA;

**return** SQR(A.r)\*acos(dA/A.r) - dA\*sqrt(SQR(A.r)-SQR(dA))

+ SQR(B.r)\*acos(dB/B.r) - dB\*sqrt(SQR(B.r)-SQR(dB));

}

**int** main () {

Circle A, B;

PI = acos(-1);

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&A.o.x, &A.o.y, &A.r, &B.o.x, &B.o.y, &B.r) == 6) {

printf("Area = %.5lf\n", CIArea(A, B));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Circle-Line Intersection

*=================================================================*

*Description: Given either a line or a line segment, and a circle*

*these routines calculate the number and coordinates*

*of the intersection points.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 8, 2002*

*References: mathworld.wolfram.com/Circle-LineIntersection.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

#define EPS 1e-8

**typedef** **struct**{

**double** x, y;

} Point;

**typedef** **struct**{

Point o;

**double** r;

} Circle;

**int** sgn(**double** x){

**return** x < 0 ? -1 : 1;

}

**double** dist\_2d(Point a, Point b){

**return** sqrt(SQR(a.x-b.x)+SQR(a.y-b.y));

}

**int** circ\_iline\_isect(Circle c, Point a, Point b,

Point \*r1, Point \*r2){

**double** dx = b.x-a.x, dy = b.y-a.y;

**double** sdr = SQR(dx)+SQR(dy), dr = sqrt(sdr);

**double** D,disc,x,y;

a.x -= c.o.x; a.y -= c.o.y;

b.x -= c.o.x; b.y -= c.o.y;

D = a.x\*b.y - b.x\*a.y;

disc = SQR(c.r\*dr)-SQR(D);

**if**(disc < 0) **return** 0;

x = sgn(dy)\*dx\*sqrt(disc);

y = fabs(dy)\*sqrt(disc);

r1->x = (D\*dy + x)/sdr + c.o.x;

r2->x = (D\*dy - x)/sdr + c.o.x;

r1->y = (-D\*dx + y)/sdr + c.o.y;

r2->y = (-D\*dx - y)/sdr + c.o.y;

**return** disc == 0 ? 1 : 2;

}

**int** circ\_lineseg\_isect(Circle c, Point a, Point b,

Point \*r1, Point \*r2){

**double** d = dist\_2d(a,b);

**int** res = circ\_iline\_isect(c,a,b,r1,r2);

**if**(res == 2 && dist\_2d(a,\*r2)+dist\_2d(\*r2,b) != d) res--;

**if**(res >= 1 && dist\_2d(a,\*r1)+dist\_2d(\*r1,b) != d){

\*r1 = \*r2;

res--;

}

**return** res;

}

**int** main(){

Circle c;

Point a, b, r1, r2, r3, r4;

**int** res1, res2;

**while**(scanf("%lf %lf %lf %lf %lf %lf %lf",

&c.o.x, &c.o.y, &c.r, &a.x, &a.y, &b.x, &b.y) == 7){

res1 = circ\_iline\_isect(c, a, b, &r1, &r2);

res2 = circ\_lineseg\_isect(c, a, b, &r3, &r4);

printf("Infinite line has %d intersection point(s)\n", res1);

**if**(res1 >= 1) printf("[%.6f,%.6f]\n", r1.x, r1.y);

**if**(res1 == 2) printf("[%.6f,%.6f]\n", r2.x, r2.y);

printf("Line segment has %d intersection point(s)\n", res2);

**if**(res2 >= 1) printf("[%.6f,%.6f]\n", r3.x, r3.y);

**if**(res2 == 2) printf("[%.6f,%.6f]\n", r4.x, r4.y);

printf("\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Circle Intersection Points

*=================================================================*

*Description: Computes the # and location of intersections.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Scott Crosswhite*

*Date: Dec 21, 2002*

*References: http://hades.ph.tn.tudelft.nl/.../MathWorld/*

*math/math/c/c308.htm*

*-----------------------------------------------------------------*

*Reliability: 1.5 (Spain 10425)*

*(Broderick's test batch)*

*Notes:*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define EPS 1e-8

#define SQR(x) ((x)\*(x))

**double** PI;

**typedef** **struct** {

**double** x,y;

} Point;

**typedef** **struct** {

Point o;

**double** r;

} Circle;

**enum** int\_t {NONE=0, ONE, TWO, AEQUALSB, AINB, BINA,

AINB\_TANGENT, BINA\_TANGENT};

Point rotate\_2d(Point p, Point o, **double** theta){

**double** m[2][2];

Point r;

m[0][0] = m[1][1] = cos(theta);

m[0][1] = -sin(theta);

m[1][0] = -m[0][1];

p.x -= o.x;

p.y -= o.y;

r.x = m[0][0] \* p.x + m[0][1] \* p.y + o.x;

r.y = m[1][0] \* p.x + m[1][1] \* p.y + o.y;

**if**(fabs(r.x) < EPS) r.x = 0;

**if**(fabs(r.y) < EPS) r.y = 0;

**return** r;

}

**int** CIType (Circle A, Circle B) {

**double** distance, dx = A.o.x - B.o.x, dy = A.o.y - B.o.y;

distance = sqrt(dx\*dx + dy\*dy);

**if** (distance < EPS && fabs(A.r-B.r) < EPS) **return** AEQUALSB;

**if** (fabs(distance - (A.r + B.r)) < EPS) **return** ONE;

**if** (distance > A.r + B.r) **return** NONE;

**if** (distance + A.r <= B.r) {

**if** (B.r - (distance+A.r) < EPS) **return** AINB\_TANGENT;

**return** AINB;

}

**if** (distance + B.r <= A.r) {

**if** (A.r - (distance+B.r) < EPS) **return** BINA\_TANGENT;

**return** BINA;

}

**return** TWO;

}

*/\* Return value is # of intersections \*/*

**int** CIPoints (Circle A, Circle B, Point \*s, Point \*t) {

**double** dx = B.o.x-A.o.x, dy = B.o.y-A.o.y;

**double** dA, d, c, a;

**int** type;

type = CIType(A, B);

d = sqrt(dx\*dx + dy\*dy);

**switch** (type) {

**case** AINB\_TANGENT:

s->x = B.o.x + (B.r/d)\*-dx;

s->y = B.o.y + (B.r/d)\*-dy;

**return** 1;

**case** BINA\_TANGENT: **case** ONE:

s->x = A.o.x + (A.r/d)\*dx;

s->y = A.o.y + (A.r/d)\*dy;

**return** 1;

**case** TWO:

c = atan2(dy, dx);

a = sqrt(4\*SQR(d)\*SQR(A.r) - SQR(SQR(d)-SQR(B.r)+SQR(A.r))) /d;

dA = (SQR(d) - SQR(B.r) + SQR(A.r)) / (2\*d);

t->x = s->x = dA + A.o.x;

s->y = a/2 + A.o.y;

t->y = -a/2 + A.o.y;

*/\* Rotate these points \*/*

\*s = rotate\_2d(\*s, A.o, c);

\*t = rotate\_2d(\*t, A.o, c);

**return** 2;

**default**:

**return** type;

}

}

**int** main () {

Circle A, B;

Point s, t;

**int** num;

PI = acos(-1);

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&A.o.x, &A.o.y, &A.r, &B.o.x, &B.o.y, &B.r) == 6) {

num = CIPoints(A, B, &s, &t);

printf("There are %d intersections. ", num);

**if** (num) printf("(%.3f, %.3f) ", s.x, s.y);

**if** (num == 2) printf("(%.3f, %.3f)", t.x, t.y);

printf("\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Circle Intersection Type

*=================================================================*

*Description: Determines the type of intersection of 2 circles.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Scott Crosswhite*

*Date: Oct 24, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: NONE - No intersection*

*ONE - One intersection (no other common points)*

*TWO - Two intersections*

*AEQUALSB - Same circle*

*AINB, BINA - One circle is in the other*

*AINB\_TANGENT, BINA\_TANGENT - As above, with 1 pt tangent*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define EPSILON 1e-8

**typedef** **struct** {

**double** x,y;

} Point;

**typedef** **struct** {

Point o;

**double** r;

} Circle;

**enum** int\_t {NONE=0, ONE, TWO, AEQUALSB, AINB, BINA,

AINB\_TANGENT, BINA\_TANGENT};

**int** CIType (Circle A, Circle B) {

**double** distance, dx = A.o.x - B.o.x, dy = A.o.y - B.o.y;

distance = sqrt(dx\*dx + dy\*dy);

**if** (distance < EPSILON && fabs(A.r-B.r) < EPSILON) **return** AEQUALSB;

**if** (fabs(distance - (A.r + B.r)) < EPSILON) **return** ONE;

**if** (distance > A.r + B.r) **return** NONE;

**if** (distance + A.r <= B.r) {

**if** (B.r - (distance+A.r) < EPSILON) **return** AINB\_TANGENT;

**return** AINB;

}

**if** (distance + B.r <= A.r) {

**if** (A.r - (distance+B.r) < EPSILON) **return** BINA\_TANGENT;

**return** BINA;

}

**return** TWO;

}

*/\* To test output \*/*

**char** type\_s[8][50] = {"No intersection",

"One intersection",

"Two intersections",

"Circles match",

"A is entirely in B",

"B is entirely in A",

"A is in B and tangent",

"B is in A and tangent"};

**int** main () {

**int** type;

Circle A, B;

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&A.o.x, &A.o.y, &A.r, &B.o.x, &B.o.y, &B.r) == 6) {

type = CIType(A, B);

printf("Type: %s \n", type\_s[type]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry –*

## Infinite Line Intersection

*=================================================================*

*Description: Given two infinite lines specified by two points,*

*determines whether they intersect at one point,*

*infinitely many points, or no points. In the first*

*case, the point of intersection is also returned.*

*The points of a line must be different (otherwise,*

*the line is not define).*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Howard Cheng*

*Date: Nov 01, 2003*

*References: www.exaflop.org/docs/cgafaq/cga1.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - returns 1 if intersect at a point*

*- returns 0 if no intersection*

*- returns -1 if lines coincide*

*\*/*

#include <stdio.h>

#include <math.h>

**typedef** **struct**{

**double** x, y;

} Point;

#define EPS 1e-8

**int** isect\_iline(Point a, Point b, Point c, Point d, Point \*p){

**double** r, denom, num1;

num1 = (a.y - c.y) \* (d.x - c.x) - (a.x - c.x) \* (d.y - c.y);

denom = (b.x - a.x) \* (d.y - c.y) - (b.y - a.y) \* (d.x - c.x);

**if** (fabs(denom) >= EPS) {

r = num1 / denom;

p->x = a.x + r\*(b.x - a.x);

p->y = a.y + r\*(b.y - a.y);

**return** 1;

}

**if** (fabs(num1) >= EPS) **return** 0;

**return** -1;

}

**int** main(){

Point a, b, c, d, p; **int** res;

**while** (scanf("%lf %lf %lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y, &d.x, &d.y) == 8) {

res = isect\_iline(a, b, c, d, &p);

**if** (res == 1) {

printf("Intersect at (%0.2f, %0.2f)\n", p.x, p.y);

} **else** **if** (res == 0) {

printf("Don't intersect\n");

} **else** {

printf("Infinite number of intersections\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Line Segment Intersection

*=================================================================*

*Description: Given two line segments specified by endpoints,*

*determine whether they intersect at one point,*

*infinitely many points, or no points. In the first*

*case, the point of intersection is also returned.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Howard Cheng*

*Date: Nov 1, 2003*

*References: www.exaflop.org/docs/cgafaq/cga1.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - The endpoints of a line must be different.*

*- Returns 1 if segments intersect at a point,*

*0 if line segments don't intersect*

*-1 if the lines coincide*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define EPS 1E-8

**typedef** **struct** {

**double** x, y;

} Point;

**int** intersect\_line(Point a, Point b, Point c, Point d, Point \*p){

Point t; **double** r, s, denom, num1, num2;

num1 = (a.y - c.y)\*(d.x - c.x) - (a.x - c.x)\*(d.y - c.y);

num2 = (a.y - c.y)\*(b.x - a.x) - (a.x - c.x)\*(b.y - a.y);

denom = (b.x - a.x)\*(d.y - c.y) - (b.y - a.y)\*(d.x - c.x);

**if** (fabs(denom) >= EPS) {

r = num1 / denom;

s = num2 / denom;

**if** (0-EPS <= r && r <= 1+EPS &&

0-EPS <= s && s <= 1+EPS) {

p->x = a.x + r\*(b.x - a.x);

p->y = a.y + r\*(b.y - a.y);

**return** 1;

}

**return** 0;

}

**if** (fabs(num1) >= EPS) **return** 0;

**if** (a.x > b.x || (a.x == b.x && a.y > b.y)) { t = a; a = b; b = t; }

**if** (c.x > d.x || (c.x == d.x && c.y > d.y)) { t = c; c = d; d = t; }

**if** (a.x == b.x) {

**if** (b.y == c.y) {

\*p = b; **return** 1;

} **else** **if** (a.y == d.y) {

\*p = a; **return** 1;

} **else** **if** (b.y < c.y || d.y < a.y)

**return** 0;

} **else** {

**if** (b.x == c.x) {

\*p = b; **return** 1;

} **else** **if** (a.x == d.x) {

\*p = a; **return** 1;

} **else** **if** (b.x < c.x || d.x < a.x)

**return** 0;

}

**return** -1;

}

**int** main(){

Point a, b, c, d, p; **int** res;

**while** (scanf("%lf %lf %lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y, &d.x, &d.y) == 8) {

res = intersect\_line(a, b, c, d, &p);

**if** (res == 1) {

printf("Intersect at (%0.2f, %0.2f)\n", p.x, p.y);

} **else** **if** (res == 0) {

printf("Don't intersect\n");

} **else** {

printf("Infinite number of intersections\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2-D Geometry:*

## Area of intersection between two general polygons

*=================================================================*

*Description: Takes in two general simple polygons*

*(not necessarily convex) and computes the area of*

*their intersection.*

*Complexity: O(n^2) worst case*

*-----------------------------------------------------------------*

*Author: Jason Klaus*

*Date: November 3, 2003*

*References: www.cg.tuwien.ac.at/~theussl/papers/polygons.ps.gz*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Simplified the documented triangulation algorithm*

*somewhat so it is not as long but a constant time*

*slower.*

*\*/*

#include <cstdio>

#include <list>

#include <vector>

#include <algorithm>

**using** **namespace** std;

#define EPS 1e-8

*// 1 for clockwise ordering of points, -1 for counter-clockwise*

#define ORDER 1

**struct** Point {

**double** x, y;

**bool** **operator** < (**const** Point &p) **const** {

**return** (y + EPS < p.y) || ((y - EPS < p.y) && (x + EPS < p.x));

}

**bool** **operator** == (**const** Point &p) **const** {

**return** !(\***this** < p) && !(p < \***this**);

}

};

**struct** Triangle {

Point p[3];

};

*//2D cross product of vectors a->b, c->d*

**inline** **double** cross(**const** Point &a, **const** Point &b,

**const** Point &c, **const** Point &d){

**return** ((b.x - a.x)\*(d.y - c.y) - (d.x - c.x)\*(b.y - a.y));

}

*// This function classifies p as either being "left of" [-1],*

*// "right of" [+1] or "on" [0] the line a -> b.*

**int** leftRight(**const** Point &a, **const** Point &b, **const** Point &p){

**double** res = cross(a, b, a, p);

**if** (res > EPS)

**return** -1;

**else** **if** (res < -EPS)

**return** 1;

**return** 0;

}

*// This function returns non-zero if point b in the sequence a->b->c*

*// is a concave point or zero if it is convex.*

*// (If inside angle >= 180 deg, concave, otherwise convex)*

**int** isConcave(Point &a, Point &b, Point &c){

**return** (ORDER\*leftRight(a, b, c) <= 0);

}

*// This function returns non-zero if point p is located on or*

*// inside the triangle <a b c>.*

**int** isInsideTriangle(Point &a, Point &b, Point &c, Point &p){

**int** r1 = leftRight(a, b, p);

**int** r2 = leftRight(b, c, p);

**int** r3 = leftRight(c, a, p);

**return** ((ORDER\*r1 >= 0) && (ORDER\*r2 >= 0) && (ORDER\*r3 >= 0));

}

*// Takes in a list of n ordered points forming the polygon P, and*

*// a vector of n-2 triangles in T. P is modified during the*

*// triangulation. (n >= 3)*

**void** triangulate(list<Point> &P, vector<Triangle> &T){

list<Point>::iterator a, b, c, q;

Triangle t;

T.clear();

**if** (P.size() < 3) **return**;

**for** (a=b=P.begin(), c=++b, ++c; c != P.end(); a=b, c=++b, ++c) {

**if** (!isConcave(\*a, \*b, \*c)) {

**for** (q = P.begin(); q != P.end(); q++) {

**if** (q == a) {

++q;

++q;

**continue**;

}

**if** (isInsideTriangle(\*a, \*b, \*c, \*q)) **break**;

}

**if** (q == P.end()) {

t.p[0] = \*a;

t.p[1] = \*b;

t.p[2] = \*c;

T.push\_back(t);

P.erase(b);

b = a;

**if** (b != P.begin()) b--;

}

}

}

}

*//Finds the pt of intersection between line segments a->b and c->d*

*//Returns 1 if there is one point of intersection, stored in p.*

*//Returns 0 if there is no point of intersection, or infinitely many*

**int** isectLineSegs(Point &a, Point &b, Point &c, Point &d, Point &p)

{

**double** r, s, dn, n1, n2;

n1 = cross(c, d, c, a);

n2 = -cross(a, b, a, c);

dn = cross(a, b, c, d);

**if** ((dn > EPS) || (dn < -EPS)) {

r = n1/dn;

s = n2/dn;

**if** ((-EPS < r) && (r < 1+EPS) &&

(-EPS < s) && (s < 1+EPS)) {

p.x = a.x + r\*(b.x - a.x);

p.y = a.y + r\*(b.y - a.y);

**return** 1;

}

}

**return** 0;

}

**double** areaPoly(vector<Point> &P){

**double** area = 0.0;

vector<Point>::iterator p, q;

**for** (p = P.end()-1, q = P.begin(); q != P.end(); p = q++) {

area += p->x\*q->y - p->y\*q->x;

}

**return** area/2.0;

}

Point P0;

**bool** radialLessThan(**const** Point &a, **const** Point &b){

**return** (ORDER == leftRight(P0, a, b));

}

**double** isectAreaTriangles(Triangle &a, Triangle &b){

vector<Point> P;

vector<Point>::iterator s, e, ne;

Point p;

Triangle T[2] = { a, b };

**int** i, j, r, t, u, v;

**double** area = 0;

P.clear();

**for**(r=1, t=0; t < 2; r = t++) **for**(i = 2, j = 0; j < 3; i = j++){

**if**(isInsideTriangle(T[r].p[0],T[r].p[1],T[r].p[2],T[t].p[i]))

P.push\_back(T[t].p[i]);

**for** (u = 2, v = 0; v < 3; u = v++)

**if**(isectLineSegs(T[t].p[i],T[t].p[j],T[r].p[u],T[r].p[v],p))

P.push\_back(p);

}

**if** (!P.empty()) {

s = P.begin();

e = P.end();

sort(s, e);

ne = unique(s, e);

P.erase(ne, e);

**if** (P.size() >= 3) {

P0 = P[0];

sort(s+1, ne, radialLessThan);

area = areaPoly(P);

}

}

**return** area;

}

**double** isectAreaGpoly(list<Point> &P, list<Point> &Q){

**double** area = 0.0;

vector<Triangle> S, T;

vector<Triangle>::iterator s, t;

triangulate(P, S);

triangulate(Q, T);

**for** (s = S.begin(); s != S.end(); s++)

**for** (t = T.begin(); t != T.end(); t++)

area += isectAreaTriangles(\*s, \*t);

**return** -ORDER\*area;

}

**int** main(){

**int** n, i, j; Point p; list<Point> P[2];

**for** (j = 0; j < 2; j++) {

scanf("%d", &n);

P[j].clear();

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

scanf("%lf %lf", &p.x, &p.y);

P[j].push\_back(p);

}

printf("Polygon %d:\n", j+1);

**for** (list<Point>::iterator q=P[j].begin();q != P[j].end(); q++)

printf(" (%.2f, %.2f)\n", q->x, q->y);

printf("\n");

}

printf("Area of Intersection: %.6f\n", isectAreaGpoly(P[0],P[1]));

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Laser Reflections

*=================================================================*

*Description: Given the starting point of a laser beam, and its*

*directional vector, determine where the beam hits a*

*wall (mirror), and its new direction after reflection.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 07, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Returns 0 if the beam misses the wall, 1 if the beam*

*hits the wall, and -1 if the beam hits the wall*

*edge-on. If the beam hit an end-point, the*

*reflection vector is undefined, and -2 is returned.*

*\*/*

#include <stdio.h>

#include <math.h>

#define EPS 1E-8

**typedef** **struct** {

**double** x, y;

} Point;

Point **operator** + (Point a, Point b) {

Point p;

p.x = a.x + b.x;

p.y = a.y + b.y;

**return** p;

}

Point **operator** \* (**double** k, Point a) {

Point p;

p.x = k\*a.x;

p.y = k\*a.y;

**return** p;

}

Point **operator** \* (Point a, **double** k) {

**return** (k\*a);

}

Point **operator** / (Point a, **double** k) {

**return** a\*(1/k);

}

Point **operator** - (Point a, Point b) {

**return** a+(-1.0\*b);

}

**double** dot(Point a, Point b) {

**return** a.x\*b.x + a.y\*b.y;

}

**double** lng2(Point a) { *// find the length of a vector, squared*

**return** dot(a,a);

}

**double** lng(Point a) { *// find the length of a vector*

**return** sqrt(dot(a,a));

}

Point closest\_pt\_iline(Point a, Point b, Point p) {

**double** along = dot(b-a,p-a)/lng2(b-a);

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

}

*/\* return 1 for intersection, 0 for not, -1 for coincidental \*/*

**int** intersect\_iline(Point a, Point b, Point c, Point d, Point \*p) {

**double** den, num;

num = (a.y - c.y)\*(d.x - c.x) - (a.x - c.x)\*(d.y - c.y);

den = (b.x - a.x)\*(d.y - c.y) - (b.y - a.y)\*(d.x - c.x);

**if** (fabs(den) >= EPS) {

\*p = a + (b-a)\*num/den;

**return** 1;

} **else** {

**if** (fabs(num) >= EPS) {

**return** 0;

} **else** {

**return** -1;

}

}

}

Point reflect(Point a, Point b, Point c) {

**return** 2.0\*closest\_pt\_iline(a,b,c) - c;

}

**int** bounce(Point ori, Point dir, Point a, Point b,

Point \*np, Point \*ndir) {

Point tmp;

**int** res;

res = intersect\_iline(ori, ori+dir, a, b, &tmp);

**if** (res == -1) **return** -1;

**if** (res != 1 || dot(tmp-ori,dir) < 0 ||

fabs(lng(a-tmp) + lng(b-tmp) - lng(a-b)) > EPS)

**return** 0;

\*np = tmp;

**if** (lng2(a-tmp) < EPS || lng2(b-tmp) < EPS) **return** -2;

\*ndir = reflect(a,b,tmp+dir) - tmp;

**return** 1;

}

**int** main() {

Point wall[2], ori, vec, newori, newvec;

**int** res;

**while** (scanf("%lf %lf %lf %lf", &wall[0].x, &wall[0].y,

&wall[1].x, &wall[1].y) == 4) {

scanf("%lf %lf", &ori.x, &ori.y);

scanf("%lf %lf", &vec.x, &vec.y);

res = bounce(ori, vec, wall[0], wall[1], &newori, &newvec);

**if** (res==0) printf("missed\n");

**else** **if** (res==-1) printf("hit edge-on\n");

**else** **if** (res==-2) printf("hit a corner\n");

**else** printf("hit %g %g, new vector is %g %g\n",

newori.x, newori.y, newvec.x, newvec.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Lattice-polygons - Pick's Theorem

*=================================================================*

*Description: Given a lattice N-gon (a polygon where ends points*

*are on integer coordinates), this routine counts*

*the number of lattice points on the boundary, as*

*well as the number of interior lattice points.*

*Complexity: O(roughly N) (N calls to gcd())*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 12, 2002*

*References: http://mathworld.wolfram.com/PicksTheorem.html*

*-----------------------------------------------------------------*

*Reliability: 1 successful use (Spain 10088) Sept 2002*

*Notes: Pick's Formula is: A = I + B/2 - 1*

*where A = area of the closed lattice polygon*

*I = # of interior lattice points*

*B = # of lattice points on boundary*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAXN 1005

**typedef** **struct**{

**long** **long** x, y;

} Point;

**int** gcd(**long** **long** a, **long** **long** b){

**int** r;

a = abs(a);

b = abs(b);

**while** (b) {

r = a % b;

a = b;

b = r;

}

**return** a;

}

**double** area\_poly(Point \*p, **int** n){

**double** sum = 0;

**int** i, j;

**for**(i = n-1, j = 0; j < n; i = j++)

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

**return** sum/2.0;

}

**void** lat\_poly\_pick(Point \*p, **int** n, **long** **long** \*I, **long** **long** \*B){

**int** i, j, dx, dy;

**double** A = fabs(area\_poly(p, n));

\*B = 0;

**for**(i = n-1, j = 0; j < n; i = j++){

dx = abs(p[i].x - p[j].x);

dy = abs(p[i].y - p[j].y);

\*B += gcd(dx,dy);

}

\*I = A+1-\*B/2.0;

}

**int** main(){

**int** i, n;

**long** **long** numI, numB;

Point p[MAXN];

**while**(scanf("%d", &n) == 1 && n){

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

scanf("%lld %lld", &p[i].x, &p[i].y);

}

lat\_poly\_pick(p, n, &numI, &numB);

printf("The lattice polygon has %lld interior points and %lld boundary points\n", numI, numB);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Largest rectangle not enclosing points

*=================================================================*

*Description: Given a field of length W by H, and a set of*

*N points, find one of the largest rectangles*

*which do not enclose any points*

*Complexity: O(N^3) where N is the number of points*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Feb 15, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - Field is [0..W] x [0..H]. Offset as needed.*

*- This works by trying all possible left and right*

*boundaries, and determining the largest possible*

*rectangle given those two bounds.*

*\*/*

#define MAXN 100

#include <stdio.h>

#include <set>

#include <vector>

**using** **namespace** std;

**typedef** **struct**{

**double** x, y;

} Point;

**typedef** **struct**{

**double** left,right,top,bot;

} Rect;

Rect max\_Rect(Point \*p, **int** n, **double** width, **double** height){

set< **double** > xset, yset, ys; Rect r;

**double** old, curr, bot, top, area, best;

set<**double**>::iterator left, right, y;

xset.insert(0); xset.insert(width);

yset.insert(0); yset.insert(height);

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

xset.insert(p[i].x);

yset.insert(p[i].y);

}

area = -1;

**for**(left = xset.begin(); left != xset.end(); left++){

**for**(right = left; right != xset.end(); right++){

ys.clear();

ys.insert(height);

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

**if**(\*left < p[i].x && p[i].x < \*right) ys.insert(p[i].y);

old = best = 0;

**for**(y = ys.begin(); y != ys.end(); y++){

**if**(\*y-old > best){

best = \*y-old;

bot = old;

top = \*y;

}

old = \*y;

}

curr = best\*(\*right-\*left);

**if**(curr > area){

area = curr;

r.left = \*left;

r.right = \*right;

r.bot = bot;

r.top = top;

}

}

}

**return** r;

}

**int** main(){

**int** i, n; **double** w, h;

Rect r; Point p[MAXN];

**while**(scanf(" %lf %lf %d", &w, &h, &n) == 3){

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

scanf("%lf %lf", &p[i].x, &p[i].y);

r = max\_Rect(p, n, w, h);

printf("Largest rectangle: %.2f -> %.2f, %.2f -> %.2f\n",

r.left, r.right, r.bot, r.top);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Polygon Midpoints to Verticies

*=================================================================*

*Description: Consider a polygon with n sides. Given the*

*midpoints of each side, this code will find the*

*vertices of the polygon.*

*Complexity: O(n)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Nov 19, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: This only works when n is odd; the answer is not*

*unique for polygons with an even number of vertices.*

*The polygon does not need to be convex. The points*

*can be in clockwise or counterclockwise order.*

*\*/*

#include <stdio.h>

**typedef** **struct** {

**double** x, y;

} Point;

**void** midpts2vert(Point \*midpts, **int** n, Point \*poly) {

**int** i;

poly[0] = midpts[0];

**for** (i = 1; i < n; i += 2) {

poly[0].x += midpts[i+1].x - midpts[i].x;

poly[0].y += midpts[i+1].y - midpts[i].y;

}

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

poly[i].x = 2.0\*midpts[i-1].x - poly[i-1].x;

poly[i].y = 2.0\*midpts[i-1].y - poly[i-1].y;

}

}

**int** main() {

Point midpts[1000], poly[1000];

**int** i, n;

**while** (scanf("%d",&n)==1) {

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

scanf("%lf %lf", &midpts[i].x, &midpts[i].y);

midpts2vert(midpts, n, poly);

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

printf("%0.3f %0.3f\n", poly[i].x, poly[i].y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Minimum bounding circle

*=================================================================*

*Description: Given a set of points, this returns the circle with*

*the minimum area which completely contains all those*

*points*

*Complexity: O(n^3) worst case, where n is the number of points*

*but on average O(n)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Jan 24, 2003*

*References: http://www.cs.unc.edu/~eberly/gr\_cont.htm*

*-----------------------------------------------------------------*

*Reliability: 2 (Spain 10005 - Packing polygons)*

*Problem C Aliens Jan 21, 2003*

*Notes: This is a simplification of the old min\_circle code*

*by Scott Crosswhite (/Old/min\_circle.c)*

*The input array is sorted to increase stability of*

*an answer. This may be removed to increased speed*

*\*/*

#include <stdio.h>

#include <math.h>

#include <float.h>

#include <stdlib.h>

#define EPS 1e-6

#define MAXN 100

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**double** x, y;

} Point;

**typedef** **struct**{

**double** x, y, r;

} Circle;

**int** inside(Point p, Circle c){

**return** SQR(p.x-c.x)+SQR(p.y-c.y) <= SQR(c.r);

}

Circle Circle1(Point p){

Circle c;

c.x = p.x; c.y = p.y; c.r = 0;

**return** c;

}

Circle Circle2(Point p1, Point p2){

Circle c;

c.x = 0.5\*(p1.x + p2.x);

c.y = 0.5\*(p1.y + p2.y);

c.r = 0.5\*sqrt(SQR(p1.x-p2.x)+SQR(p1.y-p2.y));

**return** c;

}

Circle Circle3(Point p1, Point p2, Point p3){

Circle res; **double** a,b,c,d,e,f,g;

a = p2.x - p1.x; b = p2.y - p1.y;

c = p3.x - p1.x; d = p3.y - p1.y;

e = (p2.x + p1.x)\*a + (p2.y + p1.y)\*b;

f = (p3.x + p1.x)\*c + (p3.y + p1.y)\*d;

g = 2.0\*(a\*(p3.y - p2.y) - b\*(p3.x - p2.x));

**if** (fabs(g) < EPS){

res.x = res.y = res.r = DBL\_MAX;

**return** res;

}

res.x = (d\*e - b\*f) / g;

res.y = (a\*f - c\*e) / g;

res.r = sqrt(SQR((p1.x-res.x))+SQR((p1.y-res.y)));

**return** res;

}

Circle min\_circle(Point \*p, **int** n){

**int** i, j, k; Point t; Circle c = Circle1(p[0]);

*/\* Randomize point array to avoid doctored input - may modify the*

*limit on the for loop to increase/decrease randomness \*/*

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

j = rand() % n;

k = rand() % n;

t = p[j]; p[j] = p[k]; p[k] = t;

}

**for**(i = 1; i < n; i++) **if**(!inside(p[i], c)){ c = Circle1(p[i]);

**for**(j = 0; j < i; j++) **if**(!inside(p[j], c)){ c = Circle2(p[i],p[j]);

**for**(k = 0; k < j; k++) **if**(!inside(p[k], c)) c = Circle3(p[i],p[j],p[k]);}}

**return** c;

}

**int** main(){

Point pts[MAXN]; Circle res; **int** i, n;

**while**(scanf("%d", &n) == 1 && n){

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

scanf("%lf %lf", &pts[i].x, &pts[i].y);

res = min\_circle(pts, n);

printf("(%.3f, %.3f) r = %.3f\n", res.x, res.y, res.r);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Point in convex polygon

*=================================================================*

*Description: Given a point an a convex polygon, returns 1 if the*

*point is in the polygon*

*Can also check to see if point is on boundary*

*Complexity: O(N) where N is number of vertices of polygon*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk, Gilbert Lee*

*Date: Nov 14, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define EPS 1E-8

#define BOUNDARY 1 */\* What to return if the point is on the*

*boundary of the polygon (0 = false, 1 = true) \*/*

**typedef** **struct**{

**double** x, y;

} Point;

**double** area\_tri(Point a, Point b, Point c){

**double** area;

area = (b.x-a.x) \* (c.y-a.y)

-(b.y-a.y) \* (c.x-a.x);

**return** (fabs(area))/2;

}

**int** point\_in\_convex\_poly(Point \*poly, **int** n, Point p){

**double** areapoly = 0, areatri = 0, temp;

**int** bflag = 0;

**int** i, j;

**for**(i = 0, j = n-1; i < n; j = i++){

areapoly += area\_tri(poly[i], poly[j], poly[0]);

temp = area\_tri(poly[i], poly[j], p);

**if**(temp < EPS) bflag = 1;

areatri += temp;

}

**if**(fabs(areapoly-areatri) < EPS){

**if**(bflag) **return** BOUNDARY;

**else** **return** 1;

}

**return** 0;

}

**int** main(){

**int** i, n;

Point \*poly, p;

printf("How many vertices in the polygon? ");

scanf("%d", &n);

poly = (Point \*) malloc(n\***sizeof**(Point));

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

scanf("%lf %lf", &poly[i].x, &poly[i].y);

}

printf("Now enter points to test:\n");

**while**(scanf("%lf %lf", &p.x, &p.y) == 2){

**if**(point\_in\_convex\_poly(poly, n, p)){

printf("YES\n");

} **else** {

printf("NO\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Polygon Similarity & Congruency

*=================================================================*

*Description: Given polygons p1 and p2, determines whether or not*

*p2 is a transformed version of p1. Valid transforms*

*include flips, rotations, translation, scaling, and*

*reversing the order of the points.*

*Complexity: O(N^2)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: February 27, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Polygon congruency can also be determined.*

*The polygons must not be self-intersecting.*

*The polygons may be modified; make a copy if you*

*need to.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#define EPS 1E-8

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

**double** dist2(Point a, Point b) {

**return** SQR(a.x-b.x) + SQR(a.y-b.y);

}

**double** dot(Point a, Point b, Point c) {

**return** (a.x-b.x)\*(c.x-b.x) + (a.y-b.y)\*(c.y-b.y);

}

**double** cross(Point a, Point b, Point c) {

**return** (a.x-b.x)\*(c.y-b.y) - (c.x-b.x)\*(a.y-b.y);

}

*/\* given a polygon, remove verticies with 180 degree angles.*

*return the number of verticies remaining. \*/*

**int** poly\_reduce(Point \*poly, **int** n) {

Point \*npoly = malloc(n\***sizeof**(Point));

**int** i, cnt = 0;

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

**if** (fabs(cross(poly[i], poly[(i+1)%n], poly[(i+2)%n])) > EPS)

npoly[cnt++] = poly[(i+1)%n];

memcpy(poly,npoly,cnt\***sizeof**(Point));

free(npoly);

**return** cnt;

}

**int** poly\_similar(Point \*p1, **int** n, Point \*p2, **int** m) {

**double** ratio, turn;

Point a, b, c, d, e, f, tmp;

**int** i, j, ori;

n = poly\_reduce(p1,n);

m = poly\_reduce(p2,m);

**if** (n != m) **return** 0;

**for** (ori = 0; ori < 2; ori++) {

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

turn = cross(p1[0],p1[1],p1[2]);

turn /= cross(p2[i],p2[(i+1)%n],p2[(i+2)%n]);

ratio = dist2(p1[0],p1[1])/dist2(p2[i],p2[(i+1)%n]);

*/\* uncomment the following line for polygon congruency \*/*

*/\* if (fabs(ratio-1) > EPS) continue; \*/*

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

a = p1[j];

b = p1[(j+1)%n];

c = p1[(j+2)%n];

d = p2[(j+i)%n];

e = p2[(j+i+1)%n];

f = p2[(j+i+2)%n];

**if** (fabs(cross(a,b,c)/cross(d,e,f)-turn) > EPS ||

fabs(dist2(a,b)/dist2(d,e)-ratio) > EPS ||

fabs(dot(a,b,c)-dot(d,e,f)\*ratio) > EPS) **break**;

}

**if** (j==n) **return** 1;

}

**for** (i = n/2; i < n; i++)

tmp = p2[i]; p2[i] = p2[n-i-1]; p2[n-i-1] = tmp;

}

**return** 0;

}

**int** main() {

Point p1[1100], p2[1100];

**int** i, n, m;

**while** (scanf("%d %d", &n, &m)==2) {

**for** (i = 0; i < n; i++) scanf("%lf %lf", &p1[i].x, &p1[i].y);

**for** (i = 0; i < m; i++) scanf("%lf %lf", &p2[i].x, &p2[i].y);

**if** (!poly\_similar(p1,n,p2,m)) printf("NOT ");

printf("SIMILAR\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Generalized Equation of a line given two points

*=================================================================*

*Description: Given two points, computes the unique values*

*A, B and C for the line going through A and B such*

*that Ax + By = C*

*and A >= 0*

*and A^2 + B^2 = 1*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 7, 2003*

*References: www.uwm.edu/~ericskey/TANOTES/Ageometry/node4.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - The good thing about a generalized equation is*

*that unlike the slope/offset version, it is*

*always defined*

*- Two lines are parallel iff both their A's are the*

*same, and their B's are the same.*

*\*/*

#include <stdio.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**double** x, y;

} Point;

**typedef** **struct**{

**double** a, b, c;

} Line;

Line pt2line(Point a, Point b){

**double** dx = a.x-b.x, dy = a.y-b.y;

**double** len = sqrt(SQR(dx)+SQR(dy));

Line res;

**if**(dy < 0){

dy \*= -1;

dx \*= -1;

}

res.a = dy/len;

res.b = -dx/len;

res.c = res.a\*a.x + res.b\*a.y;

**return** res;

}

**int** main(){

Point a, b;

Line line;

**while**(scanf("%lf %lf %lf %lf", &a.x, &a.y, &b.x, &b.y) == 4){

line = pt2line(a,b);

printf("Equation of line going through (%.3f,%.3f)->(%.3f,%.3f):\n",

a.x, a.y, b.x, b.y);

printf("(%.3f) X + (%.3f) Y = %.3f\n",

line.a, line.b, line.c);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D Geometry:*

## Point in Polygon Test

*=================================================================*

*Description: Determines if a point is strictly inside, outside,*

*or on the boundary of a simple polygon.*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 12, 2003*

*References: This is more or less taken from Randolph Franklin's*

*http://www.ecse.rpi.edu/Homepages/wrf/geom/pnpoly.html*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 634)*

*Notes: If your points are guaranteed not to be on the*

*boundary, remove that check. You could also return*

*a third value, neither 0 nor 1, for boundary cases.*

*\*/*

#include <stdio.h>

#include <math.h>

#define BOUNDARY 1

#define EPS 1E-8

#define SQR(x) ((x)\*(x))

**typedef** **struct** {

**double** x, y;

} Point;

**double** dist2d(Point a, Point b) {

**return** sqrt(SQR(a.x-b.x) + SQR(a.y-b.y));

}

**int** pt\_in\_poly(Point \*p, **int** n, Point a) {

**int** i, j, c = 0;

**for** (i = 0, j = n-1; i < n; j = i++) {

**if** (dist2d(p[i],a)+dist2d(p[j],a)-dist2d(p[i],p[j]) < EPS)

**return** BOUNDARY;

**if** ((((p[i].y<=a.y) && (a.y<p[j].y)) ||

((p[j].y<=a.y) && (a.y<p[i].y))) &&

(a.x < (p[j].x-p[i].x) \* (a.y - p[i].y)

/ (p[j].y-p[i].y) + p[i].x)) c = !c;

}

**return** c;

}

**int** main() {

Point poly[1000], trial;

**int** i, n, m;

**while** (scanf("%d %d", &n, &m)==2 && n) {

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

scanf("%lf %lf", &poly[i].x, &poly[i].y);

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

scanf("%lf %lf", &trial.x, &trial.y);

printf("%s\n", pt\_in\_poly(poly,n,trial) ? "IN" : "OUT");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D\_Geometry:*

## Point Left/Right/In Line test

*===================================================================*

*Description:*

*Given a directed line segment and a point, this code returns*

*whether the point is to the left of, to the right of, or colinear*

*with the line segment.*

*Complexity: O(1)*

*-------------------------------------------------------------------*

*Author: Jason Klaus*

*Date: Nov 8, 2002*

*References: Based on Cross products, pg.936,*

*Introduction to Algorithms (2nd Edition),*

*Cormen, Leiserson, Rivest, Stein*

*-------------------------------------------------------------------*

*Reliability: 0 successes.*

*Notes:*

*- The line segment runs from Point a to Point b*

*- If Point a == Point b, then any Point p will be considered*

*colinear.*

*- The point to be compared is Point p.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <assert.h>

#define EPSILON 1E-10

**typedef** **struct** {

**double** x, y;

} Point;

**enum** {LEFT, RIGHT, CL};

**int** pt\_leftright(Point a, Point b, Point p)

{

**double** res;

res = (p.x - a.x)\*(b.y - a.y) -

(p.y - a.y)\*(b.x - a.x);

**if** (fabs(res) < EPSILON)

**return** CL;

**else** **if** (res > 0.0)

**return** RIGHT;

**return** LEFT;

}

**int** main(**void**)

{

Point a, b, p;

**int** res;

**while** (scanf("%lf %lf %lf %lf %lf %lf", &a.x, &a.y, &b.x, &b.y,

&p.x, &p.y) == 6) {

res = pt\_leftright(a, b, p);

printf("(%.2lf, %.2lf) is ", p.x, p.y);

**switch** (res) {

**case** LEFT:

printf("Left of");

**break**;

**case** RIGHT:

printf("Right of");

**break**;

**default**:

printf("Co-Linear with");

**break**;

}

printf(" the line segment (%.2lf, %.2lf) -> (%.2lf, %.2lf)\n",

a.x, a.y, b.x, b.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Rectangle in Rectangle Test

*=================================================================*

*Description: Given two rectangles defined by their lengths and*

*widths, returns whether or not the first rectangle*

*fits completely within the second.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 13, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 3 successful use (Feb 2003)*

*(ACM 1999 Central European: Box)*

*(ACM 2002 European: Problem B: Bricks)*

*(Prob B: Equipment Box Nov 8, 2003)*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**int** n; */\* Number of solutions \*/*

**long** **double** x[2]; */\* Solutions \*/*

} Result;

Result solve\_quad(**long** **double** a, **long** **double** b, **long** **double** c){

Result r;

**long** **double** z = SQR(b)-4\*a\*c;

**if**(z < 0){

r.n = 0;

} **else** **if**(z == 0){

r.n = 1;

r.x[0] = -b/(2\*a);

} **else** {

r.n = 2;

r.x[0] = (-b+sqrt(z))/(2\*a);

r.x[1] = (-b-sqrt(z))/(2\*a);

}

**return** r;

}

**int** fit\_diag(**double** x1, **double** y1, **double** x2, **double** y2){

**long** **double** A = SQR(x1)+SQR(y1);

**long** **double** B = -2\*SQR(x1)\*x2;

**long** **double** C = SQR(x1)\*(SQR(x2)-SQR(y1));

**long** **double** a;

Result r = solve\_quad(A,B,C);

**long** **double** w1, w2;

**int** i;

**for**(i = 0; i < r.n; i++){

**if**(0 <= r.x[i] && r.x[i] <= x2){

a = r.x[i];

w1 = sqrt(SQR(x1)-SQR(a));

w2 = sqrt(SQR(y1)-SQR(x2-a));

**if**(y2 >= sqrt(SQR(x1)-SQR(a))+sqrt(SQR(y1)-SQR(x2-a))) **return** 1;

}

}

**return** 0;

}

**int** rect\_in\_rect\_test(**double** x1, **double** y1, **double** x2, **double** y2){

**if**(x1 <= x2 && y1 <= y2) **return** 1;

**if**(x1 <= y2 && y1 <= x2) **return** 1;

**if**(x1 > x2 && y1 > y2) **return** 0;

**if**(x1 > y2 && y1 > x2) **return** 0;

**if**(fit\_diag(x1,y1,x2,y2) ||

fit\_diag(x1,y1,y2,x2) ||

fit\_diag(y1,x1,x2,y2) ||

fit\_diag(y1,x1,y2,x2)) **return** 1;

**return** 0;

}

**int** main(){

**double** len1, wid1, len2, wid2;

**while**(scanf("%lf %lf %lf %lf", &len1, &wid1, &len2, &wid2) == 4){

**if**(rect\_in\_rect\_test(len1, wid1, len2, wid2)){

printf("The first rectangle can fit inside the second\n");

} **else** {

printf("Impossible to fit\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Reflect a point across a line

*=================================================================*

*Description: Given a line defined by two points A and B, and*

*another point c, returns the reflection of point C*

*across the line.*

*Assumes that the A and B are different*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Nov 9, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**double** x, y;

} Point;

Point closest\_pt\_iline(Point a, Point b, Point c) {

Point p;

**double** dp;

b.x -= a.x;

b.y -= a.y;

dp = (b.x\*(c.x-a.x) + b.y\*(c.y-a.y)) / (SQR(b.x)+SQR(b.y));

p.x = b.x\*dp + a.x;

p.y = b.y\*dp + a.y;

**return** p;

}

Point reflect(Point a, Point b, Point c) {

Point d, p;

d = closest\_pt\_iline(a,b,c);

p.x = 2.0\*d.x - c.x;

p.y = 2.0\*d.y - c.y;

**return** p;

}

**int** main() {

Point a, b, c, d;

**while** (scanf("%lf %lf %lf %lf %lf %lf",

&a.x, &a.y, &b.x, &b.y, &c.x, &c.y)==6) {

d = reflect(a,b,c);

printf("%g %g\n", d.x, d.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D-Geometry:*

## Rotation of a point around a point

*=================================================================*

*Description: Rotates a point P around an origin point O, and*

*returns the new point. The parameter theta is the*

*amount to rotate P counter-clockwise, and is*

*measured in radians.*

*Complexity: O(1) (trig functions)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 8, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 2 (Spain 10406, 10250)*

*Notes: The routine has a built in fixer for -0.00*

*\*/*

#include <stdio.h>

#include <math.h>

#define EPS 1e-8

**typedef** **struct**{

**double** x, y;

} Point;

Point rotate\_2d(Point p, Point o, **double** theta){

**double** m[2][2];

Point r;

m[0][0] = m[1][1] = cos(theta);

m[0][1] = -sin(theta);

m[1][0] = -m[0][1];

p.x -= o.x;

p.y -= o.y;

r.x = m[0][0] \* p.x + m[0][1] \* p.y + o.x;

r.y = m[1][0] \* p.x + m[1][1] \* p.y + o.y;

**if**(fabs(r.x) < EPS) r.x = 0;

**if**(fabs(r.y) < EPS) r.y = 0;

**return** r;

}

**int** main(){

Point p,o,r;

**double** deg, PI = acos(-1);

**while**(scanf("%lf %lf %lf %lf %lf",

&p.x, &p.y, &o.x, &o.y, &deg) == 5){

deg \*= PI/180.0;

r = rotate\_2d(p,o,deg);

printf("[%.3f,%.3f] rotated %.3f radians around [%.3f,%.3f] = [%.3f,%.3f]\n",

p.x, p.y, deg, o.x, o.y, r.x, r.y);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2D\_Geometry:*

## Soddy Circle Radius

*=================================================================*

*Description: Given three tangent circles, this finds the radius*

*of the circle that is tangent to the other three.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Feb 23, 2003*

*References: http://mathworld.wolfram.com/FourCoinsProblem.html*

*http://mathworld.wolfram.com/SoddyCircles.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

**double** soddy\_rad(**double** r1, **double** r2, **double** r3){

**return** (r1\*r2\*r3)/(r2\*r3+r1\*(r2+r3)+2\*sqrt(r1\*r2\*r3\*(r1+r2+r3)));

}

**int** main(){

**double** r1, r2, r3;

**while**(scanf("%lf %lf %lf", &r1, &r2, &r3) == 3){

printf("Soddy circle radius = %f\n", soddy\_rad(r1, r2, r3));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 2-D Geometry:*

## Triangulation of a simple 2D polygon

*=================================================================*

*Description: Splits a simple polygon (not necessarily convex)*

*of n verticies into n-2 disjoint triangles whose*

*verticies are taken from the original polygon.*

*Complexity: O(n^2) worst case*

*-----------------------------------------------------------------*

*Author: Jason Klaus*

*Date: June 5, 2003*

*References: www.cg.tuwien.ac.at/~theussl/papers/polygons.ps.gz*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Simplified the documented algorithm somewhat so it*

*is not as long but a constant time slower.*

*\*/*

#include <cstdio>

#include <list>

#include <vector>

**using** **namespace** std;

#define EPS 1e-8

*// 1 for clockwise ordering of points, -1 for counter-clockwise*

#define ORDER 1

**struct** Point {

**double** x, y;

};

**struct** Triangle {

Point p[3];

};

*// This function classifies p as either being left of (-1),*

*// right of (1) or on (0) the line a -> b.*

**int** leftRight(Point &a, Point &b, Point &p){

**double** res = ((b.x - a.x)\*(p.y - a.y) -

(p.x - a.x)\*(b.y - a.y));

**if** (res > EPS)

**return** -1;

**else** **if** (res < -EPS)

**return** 1;

**return** 0;

}

*// This function returns non-zero if point b in the sequence a->b->c*

*// is a concave point or zero if it is convex.*

*// (If inside angle >= 180 deg, concave, otherwise convex)*

**int** isConcave(Point &a, Point &b, Point &c){

**return** (ORDER\*leftRight(a, b, c) <= 0);

}

*// This function returns non-zero if point p is located on or*

*// inside the triangle <a b c>.*

**int** isInsideTriangle(Point &a, Point &b, Point &c, Point &p){

**int** r1 = leftRight(a, b, p);

**int** r2 = leftRight(b, c, p);

**int** r3 = leftRight(c, a, p);

**return** ((ORDER\*r1 >= 0) && (ORDER\*r2 >= 0) && (ORDER\*r3 >= 0));

}

*// Takes in a list of n ordered points forming the polygon P, and a*

*// vector of n-2 triangles in T. P is modified during the*

*// triangulation. (n >= 3)*

**void** triangulate(list<Point> &P, vector<Triangle> &T){

list<Point>::iterator a, b, c, q;

Triangle t;

T.clear();

**if** (P.size() < 3) **return**;

**for** (a=b=P.begin(), c=++b, ++c; c != P.end(); a=b, c=++b, ++c) {

**if** (!isConcave(\*a, \*b, \*c)) {

**for** (q = P.begin(); q != P.end(); q++) {

**if** (q == a) {

++q;

++q;

**continue**;

}

**if** (isInsideTriangle(\*a, \*b, \*c, \*q)) **break**;

}

**if** (q == P.end()) {

t.p[0] = \*a;

t.p[1] = \*b;

t.p[2] = \*c;

T.push\_back(t);

P.erase(b);

b = a;

**if** (b != P.begin()) b--;

}

}

}

}

**int** main(){

**int** n, i, j;

Point p;

list<Point> P;

vector<Triangle> T;

scanf("%d", &n);

P.clear();

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

scanf("%lf %lf", &p.x, &p.y);

P.push\_back(p);

}

printf("Polygon:\n");

**for** (list<Point>::iterator q = P.begin(); q != P.end(); q++){

printf(" (%.2f, %.2f)\n", q->x, q->y);

}

printf("\n");

triangulate(P, T);

printf("Triangles:\n");

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

**for** (j = 0; j < 3; j++)

printf(" (%.2f, %.2f)", T[i].p[j].x, T[i].p[j].y);

printf("\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# 3D Geometry

############################################

*/\* 3D-Geometry:*

## Area of a convex polygon in a plane given in 3D

*=================================================================*

*Description: Given the points of a planar convex polygon in 3D*

*space, returns its area.*

*The points of the polygon must be either in cw or*

*ccw order.*

*Complexity: O(N) where N is the number of points*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 06, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 - Topcoder Round 2 2003*

*Notes: - Uses Heron's formula v1. Change to v2 if needed*

*- There must be at least 3 points*

*- Doesn't work for non-convex polygons*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define SQR(x) ((x)\*(x))

**typedef** **struct**{

**double** x, y, z;

} Point;

**double** area\_heron(**double** a, **double** b, **double** c){

**double** s = (a+b+c)/2.0;

**if**(a > s || b > s || c > s) **return** -1;

**return** sqrt(s\*(s-a)\*(s-b)\*(s-c));

}

**double** dist3d(Point a, Point b){

**return** sqrt(SQR(a.x-b.x)+SQR(a.y-b.y)+SQR(a.z-b.z));

}

**double** area\_poly3d(Point \*p, **int** n){

**int** i; **double** total = 0;

**for**(i = 2; i < n; i++)

total += area\_heron(dist3d(p[0], p[i]),

dist3d(p[0], p[i-1]),

dist3d(p[i], p[i-1]));

**return** total;

}

**int** main(){

**int** i, n; Point \*p;

**while**(scanf("%d", &n) == 1 && n >= 3){

p = (Point \*)malloc(n\***sizeof**(Point));

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

scanf("%lf %lf %lf", &p[i].x, &p[i].y, &p[i].z);

printf("Area of convex polygon is %.3f\n", area\_poly3d(p, n));

free(p);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 3D Geometry:*

## Geometry Primitives

*=================================================================*

*Description: A wide range of 3D geometry primitives are given.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 14, 2003*

*References: http://astronomy.swin.edu.au/~pbourke/geometry/planeline/*

*http://astronomy.swin.edu.au/~pbourke/geometry/lineline3d/*

*http://geometryalgorithms.com/algorithm\_archive.htm*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: All the operators (+,-,\*,/) are mandatory, but*

*aside from that, you can just type in what you*

*need. You should type in the functions you need,*

*try to compile the program, and then see what's*

*missing.*

*\*/*

#include <stdio.h>

#include <math.h>

#include <assert.h>

#define EPS 1E-8

#define pt(a) &(a.x), &(a.y), &(a.z)

**typedef** **struct** {

**double** x, y, z;

} Point;

Point **operator** + (Point a, Point b) {

Point p;

p.x = a.x + b.x;

p.y = a.y + b.y;

p.z = a.z + b.z;

**return** p;

}

Point **operator** \* (**double** k, Point a) {

Point p;

p.x = k\*a.x;

p.y = k\*a.y;

p.z = k\*a.z;

**return** p;

}

Point **operator** \* (Point a, **double** k) {

**return** (k\*a);

}

Point **operator** / (Point a, **double** k) {

**return** a\*(1/k);

}

Point **operator** - (Point a, Point b) {

**return** a+(-1.0\*b);

}

**double** dot(Point a, Point b) {

**return** a.x\*b.x + a.y\*b.y + a.z\*b.z;

}

Point cross(Point a, Point b) {

Point cp;

cp.x = a.y\*b.z-b.y\*a.z;

cp.y = b.x\*a.z-a.x\*b.z;

cp.z = a.x\*b.y-b.x\*a.y;

**return** cp;

}

**double** lng2(Point a) { *// find the length of a vector, squared*

**return** dot(a,a);

}

**double** lng(Point a) { *// find the length of a vector*

**return** sqrt(dot(a,a));

}

*/\**

### Closest Point in Line

*\*/*

Point closest\_pt\_iline(Point a, Point b, Point p) {

**double** along = dot(b-a,p-a)/lng2(b-a);

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

}

*/\**

### Closest Point in Line Segment

*\*/*

Point closest\_pt\_seg(Point a, Point b, Point p) {

**double** along;

**if** (lng2(b-a) < EPS) **return** a;

along = dot(b-a,p-a)/lng2(b-a);

**if** (along < 0) along = 0;

**if** (along > 1) along = 1;

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

}

*/\* Planes are commonly represented in one of two ways:*

*as a vector normal (perpendicular) to the plane and a*

*single point on the plane, or as three non-collinear*

*points on the plane. \*/*

*/\**

### Closest point to plane represented by vetor normal and a other point

*\*/*

Point closest\_pt\_plane(Point norm, Point a, Point p) {

Point res = cross(cross(norm,p-a),norm);

**if** (lng2(res) < EPS) **return** a;

**return** res\*dot(res,p-a)/lng2(res);

}

*/\**

### Closest point to plane represented by three others points

*\*/*

Point closest\_pt\_plane(Point a, Point b, Point c, Point p) {

Point norm;

norm = cross(b-a,c-a);

assert(lng2(norm) > EPS); *// collinearity*

**return** closest\_pt\_plane(norm,a,p);

}

*/\* Given a sphere and an infinite line, determine if the two*

*intersect, and if so, find the points of intersection. If the*

*line is tangent to the sphere, the points will be the same \*/*

*/\**

### Find the intersection points of a infinite line and a sphere

*\*/*

**int** sphere\_iline\_isect(Point c, **double** r, Point a, Point b,

Point \*p, Point \*q) {

Point vec, mid = closest\_pt\_iline(a,b,c);

**if** (lng2(c-mid) > r\*r) **return** 0;

vec = (a-b)\*sqrt((r\*r - lng2(c-mid))/lng2(a-b));

\*p = mid + vec;

\*q = mid - vec;

**return** 1;

}

*/\* Often we are given a set of points in 3D space, and we know*

*that they all lie in a certain plane. We can translate*

*that problem into a 2D problem using this code.*

*Select three of the points to define your plane. The points*

*must be distinct and not collinear. This code will find the*

*position of another point, p, in reference to an arbitrary,*

*but deterministic set of axes defined by the first three points.*

*This code just assumes that the points are actually coplanar. \*/*

*/\**

### Translate a 3D problem with planes in a 2d problem

*\*/*

Point to\_plane(Point a, Point b, Point c, Point p) {

Point norm, ydir, xdir, res;

norm = cross(b-a,c-a);

assert(lng2(norm) > EPS); *// collinearity*

xdir = (b-a)/lng(b-a); *// create orthonormal vectors*

ydir = cross(norm,xdir);

ydir = ydir/lng(ydir);

res.x = dot(p-a,xdir);

res.y = dot(p-a,ydir);

res.z = 0;

**return** res;

}

*/\**

### Given two lines in 3D space, find distance of closest approach

*\*/*

**double** line\_line\_dist(Point a, Point b, Point c, Point d) {

Point perp = cross(b-a,d-c);

**if** (lng2(perp) < EPS) */\* parallel \*/*

perp = cross(b-a,cross(b-a,c-a));

**if** (lng2(perp) < EPS) **return** 0; */\* coincident \*/*

**return** fabs(dot(a-c,perp))/lng(perp);

}

*/\**

### Given two lines in 3D space, find distance of closest approach and returns the points of each line

*\*/*

**double** closest\_approach(Point a, Point b, Point c, Point d,

Point \*p, Point \*q) {

**double** s = dot(d-c,b-a), t = dot(a-c,d-c);

**double** num, den, tmp;

den = lng2(b-a)\*lng2(d-c) - s\*s;

num = t\*s - dot(a-c,b-a)\*lng2(d-c);

**if** (fabs(den) < EPS) { */\* parallel \*/*

\*p = a;

\*q = (d-c)\*t/lng2(d-c) + c;

**if** (fabs(s) < EPS) \*q = a; */\* coincident \*/*

} **else** { */\* skew \*/*

tmp = num/den;

\*p = a + (b-a)\*tmp;

\*q = c + (d-c)\*(t + s\*tmp)/lng2(d-c);

}

**return** lng(\*p-\*q);

}

*/\**

### Is the point p on the infinite line ab?

*\*/*

**int** on\_iline(Point a, Point b, Point p) {

**return** (lng2(p-closest\_pt\_iline(a,b,p)) < EPS);

}

*/\**

### Is the point p on the segment ab?

*\*/*

**int** on\_seg(Point a, Point b, Point p) {

**return** (lng(a-p) + lng(p-b) - lng(a-b) < EPS);

}

*/\**

### Given a plane and a line ab, determine if the two intersect, and if so, find the single point of intersection

*\*/*

**int** plane\_iline\_isect(Point norm, Point ori, Point a, Point b, Point \*p) {

**double** along, den = dot(norm,b-a);

**if** (fabs(den) < EPS) { */\* parallel \*/*

**if** (lng2(cross(ori-a,b-a)) < EPS) **return** -1; */\* coincident \*/*

**return** 0; */\* non-intersecting \*/*

}

along = dot(norm,ori-a)/den;

*/\* if you want to intersect a plane with a finite segment,*

*check that (along <= 1 && along => 0) \*/*

\*p = a + along\*(b-a);

**return** 1;

}

**int** main() {

Point a, b, c, d, e, f;

**while** (scanf("%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf",

pt(a), pt(b), pt(c), pt(d))==12) {

printf("%g ",line\_line\_dist(a,b,c,d));

printf("%g\n",closest\_approach(a,b,c,d, &e, &f));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 3d geometry:*

## Great Circle

*=================================================================*

*Description: Given the latitude and longitude of two points in*

*degrees, calculates the distance over the sphere*

*between them. Latitude is given in the range*

*[-90, 90] degrees, and Longitude is given in the*

*range [-180,180] degrees.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Broderick Arneson*

*Date: October 4th, 2002.*

*References: Page on the net*

*-----------------------------------------------------------------*

*Reliability: Spain 10316*

*Notes: - Set radius to what you need.*

*\*/*

#include <stdio.h>

#include <math.h>

**double** pi;

**double** greatcircle (**double** lat1, **double** long1,

**double** lat2, **double** long2) {

**double** radius = 1.0;

**double** a = pi\*(lat1/180.0);

**double** b = pi\*(lat2/180.0);

**double** c = pi\*((long2-long1)/180.0);

**return** radius\*acos(sin(a)\*sin(b) + cos(a)\*cos(b)\*cos(c));

}

**int** main () {

**double** lat1, lat2, long1, long2;

pi = acos(-1);

**while** (scanf("%lf %lf %lf %lf ",&lat1,&long1,&lat2,&long2)==4) {

d = greatcircle(lat1, long1, lat2, long2);

printf("Great Circle Distance = %.5lf\n", d);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* 3D Geometry:*

## Sphere from 4 Points

*=================================================================*

*Description: Given 4 points in 3D space, determines the*

*parameters of a sphere on which all 4 points lie.*

*This only works for non-coplanar points.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Nov 1, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

#define EPS 1e-8

**typedef** **struct**{

**double** x, y, z;

} Point;

**double** sqr(**double** x){ **return** x\*x;}

*/\**

### Solves the determinant of a n by n matrix recursively

*\*/*

**double** solveDet(**double** m[4][4], **int** n){

**double** s[4][4], res = 0, x;

**int** i, j, skip, ssize;

**if**(n == 2) **return** m[0][0]\*m[1][1]-m[0][1]\*m[1][0];

**for**(skip = 0; skip < n; skip++){

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

**for**(j = 0, ssize = 0; j < n; j++){

**if**(j == skip) **continue**;

s[i-1][ssize++] = m[i][j];

}

x = solveDet(s, n-1);

**if**(skip % 2) res -= m[0][skip]\*x;

**else** res += m[0][skip]\*x;

}

**return** res;

}

*/\**

### Sphere from four points

*Given 4 points:*

*Returns 0 if the points are coplanar*

*Returns 1 if the points are not coplanar with:*

*o = center of sphere*

*r = radius of sphere*

*\*/*

**int** sphere\_from\_4pts(Point p[4], Point \*o, **double** \*r){

**double** m[4][5], s[4][4], sol[5];

**int** ssize, skip, i, j;

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

s[i][0] = p[i].x;

s[i][1] = p[i].y;

s[i][2] = p[i].z;

s[i][3] = 1;

}

**if**(fabs(solveDet(s, 4)) < EPS) **return** 0;

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

m[i][0] = sqr(m[i][1]=p[i].x)

+ sqr(m[i][2]=p[i].y)

+ sqr(m[i][3]=p[i].z);

m[i][4] = 1;

}

**for**(skip = 0; skip < 5; skip++){

**for**(i = 0; i < 4; i++)

**for**(j = 0, ssize=0; j < 5; j++){

**if**(j == skip) **continue**;

s[i][ssize++] = m[i][j];

}

sol[skip] = solveDet(s, 4);

}

**for**(i = 1; i < 5; i++)

sol[i] /= (sol[0] \* ((i%2) ? 1 : -1));

**for**(i = 1; i < 4; i++)

sol[4] += sqr(sol[i]/=2);

o->x = sol[1];

o->y = sol[2];

o->z = sol[3];

\*r = sqrt(sol[4]);

**return** 1;

}

**int** main(){

**int** tnum, i;

Point p[4], o;

**double** r;

scanf("%d", &tnum);

**while**(tnum--){

**for**(i = 0; i < 4; i++)

scanf("%lf %lf %lf", &p[i].x, &p[i].y, &p[i].z);

**if**(sphere\_from\_4pts(p, &o, &r))

printf("Centered at [%f, %f, %f] with radius %f\n",

o.x, o.y, o.z, r);

**else**

printf("The points are coplanar\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Arithmetic

############################################

*/\* Arithmetic:*

## Discrete Logarithm solver

*=================================================================*

*Description: Given prime P, B, and N, finds the smallest*

*exponent L such that B^L == N (mod P)*

*Complexity: O(sqrt(P))*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 13, 2002*

*References: http://plg.uwaterloo.ca/~acm00/020126/data/B.cpp*

*-----------------------------------------------------------------*

*Reliability: 1 successful use (Spain Problem 10225) Sept 2002*

*Notes: The function either returns the exponent L, or*

*-1 if no solution is found*

*\*/*

#include <stdio.h>

#include <map>

#include <math.h>

**using** **namespace** std;

#define UI unsigned

#define UL unsigned long

#define ULL unsigned long long

map<UI,UI> M;

UL times (UL a, UL b, UL m){

**return** (ULL) a \* b % m;

}

UL power(UL val, UL power, UL m){

UL res = 1, p;

**for**(p = power; p; p=p>>1){

**if**(p & 1) res = times(res, val, m);

val = times(val, val, m);

}

**return** res;

}

**int** discrete\_log(UI p, UI b, UI n){

UL i, j, jump;

M.clear();

jump = (**int**)sqrt(p);

**for** (i = 0; i < jump && i < p-1; i++){

M[power(b,i,p)] = i+1;

}

**for** (i = 0; i < p-1; i+= jump){

**if** (j = M[times(n,power(b,p-1-i,p),p)]) {

j--;

**return** (i+j)%(p-1);

}

}

**return** -1;

}

**int** main(){

UI p, b, n;

**int** x;

**while**(scanf("%d %d %d", &p, &b, &n) == 3){

x = discrete\_log(p,b,n);

**if**(x < 0) printf("no solution\n");

**else** printf("%d\n", x);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Arithmetic:*

## Fast Exponentition

*=================================================================*

*Description: Given integer x and nonnegative number N, computes*

*x^N quickly*

*Complexity: O(lg N)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 17, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: x can be negative.*

*N >= 0 otherwise routine will infinitely loop*

*The limit for the results is:*

*[-9223372036854775808 = -2^63, to*

*+9223372036854775807 = +2^63 - 1]*

*\*/*

#include <stdio.h>

#define LL long long

LL fast\_exp(**int** b, **int** n){

LL res = 1, x = b, p;

**for**(p = n; p; p >>= 1, x \*= x)

**if**(p & 1) res \*= x;

**return** res;

}

**int** main(){

**int** b, n;

**while**(scanf("%d %d", &b, &n) == 2){

printf("%d^%d = %Ld\n", b, n, fast\_exp(b,n));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\**

## Fast Exponentiation mod m

*\**

*\* Author: Howard Cheng*

*\**

*\* Given b, n, and m, computes b^n mod m quickly.*

*\**

*\*/*

#include <stdio.h>

#include <assert.h>

**int** fast\_exp(**int** b, **int** n, **int** m)

{

**int** res = 1;

**int** x = b;

**while** (n > 0) {

**if** (n & 0x01) {

n--;

res = (res \* x) % m;

} **else** {

n >>= 1;

x = (x \* x) % m;

}

}

**return** res;

}

**int** main(**void**)

{

**int** b, n, m;

**while** (scanf("%d %d %d", &b, &n, &m) == 3) {

printf("%d^%d mod %d = %d\n", b, n, m, fast\_exp(b, n, m));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Arithmatic:*

## Simpson's Rule for Numerical Intergration

*=================================================================*

*Description: Numerical integration of the function f from a to b*

*Splits the interval [a,b] into 2k pieces.*

*The error is <= (b-a)/180.0 \* M \* h^4*

*where:*

*M = max( abs(f''''(x))) for x in [a,b]*

*h = (b-a)/2k*

*Note that this means the integrals for low degree*

*polynomials are computed exactly*

*Complexity: O(k)*

*-----------------------------------------------------------------*

*Author: Adam Beacham, Brian Lau*

*Date: Oct 4, 2002*

*References: Any Calculus book.*

*-----------------------------------------------------------------*

*Reliability: 0/0*

*Notes:*

*\*/*

#include <assert.h>

#include <stdio.h>

**double** Simpson(**double** a, **double** b, **int** k, **double** (\*f)(**double**)){

**double** dx, x, t;

**int** i;

*/\* assert( (a - b) != 0 && k > 0); \*/*

dx = (b-a)/(2.0\*k);

t = 0;

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

t += (i==0 ? 1.0 : 2.0) \* (\*f)(a+2.0\*i\*dx);

t += 4.0 \* (\*f)(a+(2.0\*i+1.0)\*dx);

}

t += (\*f)(b);

**return** t \* (b-a)/6.0/k;

}

**double** example\_function(**double** x)

{

**return** x\*x;

}

**int** main(**void**)

{

**int** k;

**double** a = 0, b=5.0;

printf("Integral from %f to %f is:\n",a,b);

**for**( k=1; k<=40; k++ ) {

printf(" k = %3d: %f\n",k,Simpson(a,b,k,example\_function));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Arithmetic:*

## Cubic equation solver

*=================================================================*

*Description: Finds solutions to the cubic equation:*

*ax^3+bx^2+cx+d = 0*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 8, 2002*

*References: www.snippets.org/snippets/portable/CUBIC+C.php3*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <math.h>

**typedef** **struct**{

**int** n; */\* Number of solutions \*/*

**double** x[3]; */\* Solutions \*/*

} Result;

**double** PI;

Result solve\_cubic(**double** a, **double** b, **double** c, **double** d){

Result s;

**long** **double** a1 = b/a, a2 = c/a, a3 = d/a;

**long** **double** q = (a1\*a1 - 3\*a2)/9.0, sq = -2\*sqrt(q);

**long** **double** r = (2\*a1\*a1\*a1 - 9\*a1\*a2 + 27\*a3)/54.0;

**double** z = r\*r-q\*q\*q;

**double** theta;

**if**(z <= 0){

s.n = 3;

theta = acos(r/sqrt(q\*q\*q));

s.x[0] = sq\*cos(theta/3.0) - a1/3.0;

s.x[1] = sq\*cos((theta+2.0\*PI)/3.0) - a1/3.0;

s.x[2] = sq\*cos((theta+4.0\*PI)/3.0) - a1/3.0;

} **else** {

s.n = 1;

s.x[0] = pow(sqrt(z)+fabs(r),1/3.0);

s.x[0] += q/s.x[0];

s.x[0] \*= (r < 0) ? 1 : -1;

s.x[0] -= a1/3.0;

}

**return** s;

}

**int** main(){

**double** a,b,c,d;

Result r;

**int** i;

PI = acos(-1);

**while**(scanf("%lf %lf %lf %lf", &a, &b, &c, &d) == 4){

r = solve\_cubic(a,b,c,d);

printf("%d solution(s)\n", r.n);

**for**(i = 0; i < r.n; i++){

printf("x = %f\n", r.x[i]);

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Arithmetic:*

## Quadratic equation solver

*=================================================================*

*Description: Finds solutions to the quadratic equation:*

*ax^2+bx+c = 0*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 8, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 Successful use (Sept 2002)*

*(Spain - 10357)*

*Notes: When there are two solutions for x, r.x[0] is not*

*necessarily smaller than r.x[1]*

*\*/*

#include <stdio.h>

#include <math.h>

**typedef** **struct**{

**int** n; */\* Number of solutions \*/*

**double** x[2]; */\* Solutions \*/*

} Result;

Result solve\_quad(**double** a, **double** b, **double** c){

Result r;

**double** z = b\*b-4\*a\*c;

**if**(z < 0){

r.n = 0;

} **else** **if**(z == 0){

r.n = 1;

r.x[0] = -b/(2\*a);

} **else** {

r.n = 2;

r.x[0] = (-b+sqrt(z))/(2\*a);

r.x[1] = (-b-sqrt(z))/(2\*a);

}

**return** r;

}

**int** main(){

**double** a,b,c;

Result r;

**int** i;

**while**(scanf("%lf %lf %lf", &a, &b, &c) == 3){

r = solve\_quad(a,b,c);

printf("%d solution(s)\n", r.n);

**for**(i = 0; i < r.n; i++){

printf("x = %f\n", r.x[i]);

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## Coupons Problem

*=================================================================*

*Description: Coupons are given away in boxes of cereal. There are*

*'m' different kinds of coupons (with equiprobable*

*distribution). How many boxes of cereal would you*

*have to buy, on average, to collect them all?*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 19, 2003*

*References: http://mathforum.org/library/drmath/view/56657.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

**double** ncoupons(**int** m) {

**double** num = 0.0;

**int** i;

**for** (i = 1; i <= m; i++) num += m/(**double**) i;

**return** num;

}

*/\* A related problem: If you buy 'n' boxes of cereal, what is the*

*probability you get at least one of each of the 'm' coupons?*

*This is solved by the T(n,m) function. The T(n,m) function is*

*recursive, so this implementation creates a table using dynamic*

*programming (only once) and queries it thereafter. Like most*

*combinatoric problems, this one blows up very fast. This solution*

*works for n < 100. \*/*

**double** nways[100][100];

**void** make\_coupon\_table() {

**double** fact = 1.0;

**int** i, j;

**for** (i = 1; i < 100; i++) {

nways[i][1] = 1.0;

**for** (j = 2; j < i; j++)

nways[i][j] = j\*(nways[i-1][j] + nways[i-1][j-1]);

nways[i][i] = fact \*= i;

}

}

**double** query\_table(**int** m, **int** n) {

**if** (n < m) **return** 0.0;

**if** (m == 0) **return** 1.0;

**if** (n >= 100 || m >= 100) exit(1);

**return** nways[n][m]/pow(m,n);

}

**int** main() {

**int** i, j;

**for** (i = 1; i < 34; i++)

printf("On average, it takes %g boxes to collect all %d\n",

ncoupons(i), i);

make\_coupon\_table();

**for** (i = 0; i < 100; i++) {

printf("With %d boxes, there is a ",i);

printf("%g probability of collecting all 26\n",

query\_table(26,i));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Combinatorics

############################################

*/\* Arithmetic:*

## Binomial coefficient

*=================================================================*

*Description: Given n objects in total, returns the number of ways*

*to choose k of those objects with no regard to order*

*Complexity: O(N^2) to generate table*

*O(1) lookup cost*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 16, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: This program is accurate to up to n = 68. At n = 69*

*there is a slight deviation. Note that all results*

*with n <= 68 are representable exactly by*

*floating point.*

*\*/*

#include <stdio.h>

#include <float.h>

#define MAXN 100

**long** **double** bin[MAXN+1][MAXN+1];

**void** getBinCoeff(){

**int** i, k;

**for**(k = 0; k <= MAXN; k++){

bin[k][0] = bin[k][k] = 1;

**for**(i = 1; i < k; i++)

bin[k][i] = bin[k-1][i-1]+bin[k-1][i];

}

}

**int** main(){

**int** n,k;

getBinCoeff();

**while**(scanf("%d %d", &n, &k) == 2){

printf("%d C %d = %.0Lf\n", n,k,bin[n][k]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Combinatorics:*

## Digit Occurence count

*=================================================================*

*Description: Given a digit and a number N, return the number of*

*times the digit occurs from 1..N.*

*Complexity: O(lg N)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: May 22, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <string.h>

#include <math.h>

**long** **long** digit\_count(**int** digit, **int** max){

**long** **long** res = 0;

**char** buff[15];

**int** i, count;

**if**(max <= 0) **return** 0;

*/\* Number of times "digit" occurs in the one's place \*/*

res += max/10 + ((max % 10) >= digit ? 1 : 0);

*/\* Since we start from 1, if digit = 0, remove 1 since "0"*

*doesn't count \*/*

**if**(digit == 0) res--;

*/\* Get the number of occurences in max/10-1, and multiply this by*

*10 since we can choose 10 possible last digits [0-9] \*/*

res += digit\_count(digit, max/10 - 1) \* 10;

*/\* The number of occurences in max/10 is equal to (1+max%10) \* the*

*number of times "digit" occurs in max/10 \*/*

sprintf(buff, "%d", max/10);

**for**(i = 0, count = 0; i < strlen(buff); i++)

**if**(buff[i] == digit+'0') count++;

res += (1 + max%10) \* count;

**return** res;

}

**int** main(){

**int** digit, max;

**while**(1){

printf("Enter a digit and a number: ");

scanf("%d %d", &digit, &max);

printf("The number of times '%d' occurs from 1..%d = %lld\n",

digit, max, digit\_count(digit, max));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Combinatorics:*

## Digits in N! (N factorial)

*=================================================================*

*Description: Given N, computes the number of digits that N! will*

*occupy in base B.*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee, Ashley Zinyk*

*Date: Feb 14, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Problem B - Big Number Feb 11, 2003)*

*Notes: The results fit nicely in doubles up to N = 10^7, but*

*run time may be slow.*

*Returns 1 for n < 0, although technically undefined.*

*The base must be larger than 1.*

*\*/*

#include <stdio.h>

#include <math.h>

**long** **long** fac\_digit(**int** n, **int** b) {

**double** sum = 0; **int** i;

**for** (i = 2; i <= n; i++) sum += log(i);

**return** (**long** **long**) floor(1+sum/log(b)); */\* don't use ceil! \*/*

}

**int** main() {

**int** n, b;

**while** (scanf("%d %d", &n, &b)==2) {

printf("%d! has %lld symbol(s) when represented "

"in base %d\n", n, fac\_digit(n,b), b);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Combinatorics:*

## Josephus Ring Survivor

*=================================================================*

*Description: Suppose that there are n people in a ring, [0..n-1].*

*Count around the ring, starting from 0, and*

*dismissing every m-th person.*

*Given m, this function builds an array survive[i]*

*which contains the last person left in the ring if*

*there were i people to begin with.*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 01, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 2 (Spain 151 - Power Crisis)*

*(Spain 180 - Eeny Meeny)*

*Notes: - survive[i] is only define when i >= 1*

*\*/*

#include <stdio.h>

#define MAXN 1000005

**int** survive[MAXN];

**void** josephus(**int** n, **int** m){

**int** i;

survive[1] = 0;

**for**(i = 2; i <= n; i++)

survive[i] = (survive[i-1]+(m%i))%i;

}

**int** main(){

**int** n, m;

**while**(scanf("%d %d", &n, &m)==2){

josephus(n,m);

printf("Survivor is %d\n", survive[n]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Combinatorics:*

## Necklace/Bracelet Enumeration

*=================================================================*

*Description: Calculate the number of necklaces/bracelets with N*

*beads of K different colors.*

*Necklaces are unique up to rotation while bracelets*

*can be reflected as well (i.e. mirror images count*

*only once)*

*Complexity: O(N^2)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Nov 13, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 Successful use (Spain 10294)*

*Notes: Burnside Lemma's states that the number of such is*

*equal to the total sum of all invariants / # of*

*transformations.*

*Basically, this is what the algorithm calculates.*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 51

**int** seen[MAXN];

**double** getNecklaces(**int** n, **int** t){

**int** r, i, pos;

**double** p, sum = 0;

**for**(r = 0; r < n; r++){

memset(seen, 0, **sizeof**(seen));

p = 1;

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

**if**(seen[i]) **continue**;

p \*= t;

seen[i] = 1;

pos = i;

**while**(!seen[(pos+r)%n]){

pos += r;

pos %= n;

seen[pos] = 1;

}

}

sum += p;

}

**return** sum/n;

}

**double** getBracelets(**int** n, **int** t){

**int** r, i, pos;

**double** p, sum;

sum = n\*getNecklaces(n, t);

**for**(r = 0; r < n; r++){

memset(seen, 0, **sizeof**(seen));

p = 1;

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

**if**(seen[i]) **continue**;

p \*= t;

pos = i;

**while**(!seen[(2\*n-(pos+r))%n]){

pos = (2\*n-(pos+r))%n;

seen[pos] = 1;

}

}

sum += p;

}

**return** sum/(2\*n);

}

**int** main(){

**int** n, k;

**while**(scanf("%d %d", &n, &k) == 2){

printf("%.0f %.0f\n", getNecklaces(n,k), getBracelets(n,k));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Combinatorics –*

## Permutation index on distinct characters

*=================================================================*

*Description: Given a string formed of distinct characters,*

*returns the index of the permutation from 0..N!-1.*

*Complexity: O(N^2) where N is size of the string*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Nov 14, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: This does not work when characters can be the same*

*for example: "aaba"*

*\*/*

#include <stdio.h>

#include <string.h>

**int** permdex (**char** \*s){

**int** i, j, size = strlen(s);

**int** index = 0;

**for** (i = 1; i < size; i++){

**for** (j = i; j < size; j++)

**if** (s[i-1] > s[j]) index ++;

index \*= size - i;

}

**return** index;

}

**int** main(){

**char** s[100];

**while**(scanf(" %s", s) == 1){

printf("Index of %s = %d\n", s, permdex(s));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Dynamic Programming

############################################

*/\* Dynamic Programming:*

## Longest Ascending Subsequence – O(nlogn)

*=================================================================*

*Description: Given an array of size n, asc\_seq returns the length*

*of the longest ascending subsequence, as well as one*

*of the subsequences in S.*

*sasc\_seq returns the length of the longest strictly*

*ascending subsequence.*

*Complexity: O(n log n)*

*-----------------------------------------------------------------*

*Author: Howard Cheng / Gilbert Lee*

*Date: Oct 25, 2002*

*References: Gries, D. The Science of Programming*

*-----------------------------------------------------------------*

*Reliability: 0/0*

*Notes: If you want to do the same things with descending*

*subsequences, just reverse the array before calling*

*the routines.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

**int** asc\_seq(**int** \*A, **int** n, **int** \*S){

**int** \*m, \*seq, i, k, low, up, mid, start;

m = malloc((n+1) \* **sizeof**(**int**));

seq = malloc(n \* **sizeof**(**int**));

*/\* assert(m && seq); \*/*

**for** (i = 0; i < n; i++) seq[i] = -1;

m[1] = start = 0;

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

**if** (A[i] >= A[m[k]]) {

seq[i] = m[k++];

start = m[k] = i;

} **else** **if** (A[i] < A[m[1]]) {

m[1] = i;

} **else** {

*/\* assert(A[m[1]] <= A[c] && A[c] < A[m[k]]); \*/*

low = 1;

up = k;

**while** (low != up-1) {

mid = (low+up)/2;

**if**(A[m[mid]] <= A[i]) low = mid;

**else** up = mid;

}

seq[i] = m[low];

m[up] = i;

}

}

**for** (i = k-1; i >= 0; i--) {

S[i] = A[start];

start = seq[start];

}

free(m); free(seq);

**return** k;

}

**int** sasc\_seq(**int** \*A, **int** n, **int** \*S){

**int** \*m, \*seq, i, k, low, up, mid, start;

m = malloc((n+1) \* **sizeof**(**int**));

seq = malloc(n \* **sizeof**(**int**));

*/\* assert(m && seq); \*/*

**for** (i = 0; i < n; i++) seq[i] = -1;

m[1] = start = 0;

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

**if** (A[i] > A[m[k]]) {

seq[i] = m[k++];

start = m[k] = i;

} **else** **if** (A[i] < A[m[1]]) {

m[1] = i;

} **else** **if** (A[i] < A[m[k]]) {

low = 1;

up = k;

**while** (low != up-1) {

*/\* assert(A[m[h]] <= A[c] && A[c] < A[m[j]]); \*/*

mid = (low+up)/2;

**if**(A[m[mid]] <= A[i]) low = mid;

**else** up = mid;

}

**if** (A[i] > A[m[low]]) {

seq[i] = m[low];

m[up] = i;

}

}

}

**for** (i = k-1; i >= 0; i--) {

S[i] = A[start];

start = seq[start];

}

free(m); free(seq);

**return** k;

}

**int** main(**void**)

{

**int** \*A, \*S, n, i, k;

**while** (scanf("%d", &n) == 1 && n > 0) {

A = malloc(n\***sizeof**(**int**));

S = malloc(n\***sizeof**(**int**));

*/\* Read in array \*/*

**for** (i = 0; i < n; i++) scanf("%d", &A[i]);

*/\* Find longest ascending subsequence \*/*

k = asc\_seq(A, n, S);

printf("length = %d\n", k);

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

printf("%d ", S[i]);

}

printf("\n");

*/\* Find longest strictly ascending subsequence \*/*

k = sasc\_seq(A, n, S);

printf("length = %d\n", k);

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

printf("%d ", S[i]);

}

printf("\n");

free(A);

free(S);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Dynamic Programming:*

## Longest ascending subsequence - O(N^2)

*=================================================================*

*Description: Given an array of size N, asc\_seq returns the length*

*of the longest ascending subsequence, and stores one*

*such subsequence in the array S.*

*Ascending is defined by cmp(A,B), which returns*

*1 if A may come after B in the sequence, and 0*

*otherwise*

*Complexity: O(N^2)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Jan 16, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - To support other data types, just redefine Item*

*and cmp*

*- Various cmp functions are included below*

*\*/*

#include <stdio.h>

#define MAXN 500

**typedef** **int** Item;

**int** cmp(Item a, Item b){

**return** a >= b; */\* Ascending (Nondecreasing) \*/*

**return** a > b; */\* Strictly ascending \*/*

**return** a <= b; */\* Descending (Nonincreasing) \*/*

**return** a < b; */\* Strictly descending \*/*

}

**int** longest(**int** n, Item \*A, Item \*S){

**int** most = 0, mi, i, j, last[MAXN], len[MAXN];

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

len[i] = 1;

**for**(j = 0; j < i; j++)

**if**(cmp(A[i], A[j]) && len[j]+1 > len[i]){

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

last[i] = j;

}

**if**(len[i] > most){

most = len[i];

mi = i;

}

}

*/\* Skip if you do not need to build S \*/*

**for**(i = most-1; i >= 0; i--){

S[i] = A[mi];

mi = last[mi];

}

**return** most;

}

**int** main(){

**int** n, result, i;

Item A[MAXN], S[MAXN];

scanf("%d", &n);

**for**(i = 0; i < n; i++) scanf("%d", &A[i]);

result = longest(n, A, S);

printf("Size of longest subsequence: %d\n", result);

**for**(i = 0; i < result; i++) printf("%d ", S[i]);

printf("\n");

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Dynamic Programming:*

## Edit Distance (with path recovery)

*=================================================================*

*Description: Computes the edit distance for two strings. Namely*

*this involves certain costs for Replacement,*

*Insertions and Deletions. Given these costs, and*

*two words, this program calculates the minimum cost*

*way to transform the first string to the second. An*

*added bonus is the PathRecovery() subroutine, which*

*prints out exactly what happens step by step.*

*Complexity: O(N^2) where N is the size of the larger of the two*

*strings*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: May 10, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain Problem 526)*

*Notes: Some modifications need to be made in order to fix*

*path recovery.*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 90

**char** move[MAXN][MAXN]; */\* Type of command used \*/*

**int** g[MAXN][MAXN]; */\* Cost of changes \*/*

**int** editDistance(**char** \*src, **char** \*dst, **int** replace, **int** insert, **int** **delete**){

**int** i, j, l1, l2;

l1 = strlen(src);

l2 = strlen(dst);

**for**(j = 0; j <= l1; j++){

g[0][j] = j;

move[0][j] = 'D';

}

**for**(i = 1; i <= l2; i++){

g[i][0] = i;

move[i][0] = 'I';

**for**(j = 1; j <= l1; j++){

g[i][j] = g[i-1][j-1]+replace;

move[i][j] = 'R';

**if**(g[i-1][j]+insert < g[i][j]){

g[i][j] = g[i-1][j]+insert;

move[i][j] = 'I';

}

**if**(g[i][j-1]+**delete** < g[i][j]){

g[i][j] = g[i][j-1]+**delete**;

move[i][j] = 'D';

}

**if**(src[j-1] == dst[i-1] && g[i-1][j-1] < g[i][j]){

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

move[i][j] = 'N';

}

}

}

**return** g[l2][l1];

}

**int** counter;

**void** PathRecovery(**int** x, **int** y, **int** \*delta, **char** \*src, **char** \*dst){

**int** ndelta;

**if**(x == 0 && y == 0){

\*delta = 0;

**return**;

}

**else** {

**switch**(move[x][y]){

**case** 'R':

PathRecovery(x-1,y-1,&ndelta,src,dst);

\*delta = ndelta;

printf("%d Replace %d,%c\n", counter++, y+ndelta, dst[x-1]);

**break**;

**case** 'I':

PathRecovery(x-1,y,&ndelta,src,dst);

\*delta = ndelta+1;

printf("%d Insert %d,%c\n", counter++, y+ndelta+1, dst[x-1]);

**break**;

**case** 'D':

PathRecovery(x,y-1,&ndelta,src,dst);

\*delta = ndelta-1;

printf("%d Delete %d\n", counter++, y+ndelta);

**break**;

**case** 'N':

PathRecovery(x-1,y-1,&ndelta,src,dst);

\*delta = ndelta;

**break**;

}

}

}

**int** main(){

**int** x, first = 1, delta;

**char** s1[MAXN], s2[MAXN];

**while**(fgets(s1, MAXN, stdin)){

**if**(first) first = 0;

**else** printf("\n");

fgets(s2, MAXN, stdin);

s1[strlen(s1)-1] = 0;

s2[strlen(s2)-1] = 0;

**if**(first) first = 0;

**else** printf("\n");

x = editDistance(s1, s2, 1,1,1);

printf("%d\n", x);

counter = 1;

PathRecovery(strlen(s2), strlen(s1), &delta, s1, s2);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Dynamic Programming:*

## Integer Parititoning

*=================================================================*

*Description: Template for calculating the number of ways of*

*partitioning the integer N into M parts.*

*Complexity: O(N^2)*

*-----------------------------------------------------------------*

*Author: Gilbert LEe*

*Date: Feb 10, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 10313 Pay the Price)*

*Notes: A partition of a number N is a representation of*

*N as the sum of positive integers*

*e.g. 5 = 1+1+1+1+1*

*The number of ways of partitioning an integer N*

*into M parts is equal to the number of ways of*

*partitioning the number N with the largest element*

*being of size M. This is best seen with a Ferres-*

*Young diagram:*

*Suppose N = 8, M = 3:*

*4 = \* \* \* \**

*3 = \* \* \**

*1 = \**

*3 2 2 1*

*By transposition from rows to columns, this equality*

*can be seen.*

*P(N, M) = P(N-1, M-1) + P(N-M, M)*

*P(0, M) = P(N, 0) = 0*

*P(N, 1) = 1*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 300

#define ULL unsigned long long

ULL A[MAXN+1][MAXN+1];

**void** Build(){

**int** i, j;

memset(A, 0, **sizeof**(A));

A[0][0] = 1;

**for**(i = 1; i <= MAXN; i++){

A[i][1] = 1;

**for**(j = 2; j <= i; j++)

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

}

}

**int** main(){

**int** n, m;

Build();

**while**(scanf("%d %d", &n, &m) == 2){

printf("Partitions of %d into %d parts: %llu\n", n, m, A[n][m]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Dynamic Programming:*

## Longest Common Subsequence

*=================================================================*

*Description: Given two arrays A and B with sizes n and m*

*respectively, compute the length of the longest*

*common subsequence. This routine also returns in*

*the array 's' a longest common subsequence (it*

*may not be unique). One can specify which one to*

*choose when multiply longest common subsequences*

*exist.*

*Complexity: O(N\*M)*

*-----------------------------------------------------------------*

*Author: Howard Cheng, Gilbert Lee*

*Date: Nov 19, 2002*

*References: www.ics.uci.edu/~eppstein/161/960229.html*

*-----------------------------------------------------------------*

*Reliability: 2 (Spanish Problem 10405, 10066) Dec 2002*

*Notes: - Added way to change type of array*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#define MAXN 20

#define Atype int

#define max(x,y) (((x)>(y))?(x):(y))

**int** LCS(Atype \*A, **int** n, Atype \*B, **int** m, Atype \*s)

{

**int** L[MAXN+1][MAXN+1];

**int** i, j, k;

**for**(i = n; i >= 0; i--) **for**(j = m; j >= 0; j--){

**if**(i == n || j == m){

L[i][j] = 0;

} **else** **if**(A[i] == B[j]){

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

} **else** {

L[i][j] = max(L[i+1][j], L[i][j+1]);

}

}

*/\* The following is not needed if you are not interested in*

*a longest common subsequence \*/*

k = 0;

i = j = 0;

**while**(i < n && j < m){

**if**(A[i] == B[j]){

s[k++] = A[i++];

j++;

} **else** **if**(L[i+1][j] > L[i][j+1]){

i++;

} **else** **if**(L[i+1][j] < L[i][j+1]){

j++;

} **else** {

*/\* put tie-breaking conditions here \*/*

*/\* eg. pick the one that starts at the first one the earliest \*/*

j++;

}

}

**return** L[0][0];

}

**int** main(**void**)

{

Atype A[MAXN], B[MAXN], s[MAXN];

**int** m, n, i, l;

**while** (scanf("%d %d", &n, &m) == 2 && 1 <= n && 1 <= m &&

n <= MAXN && m <= MAXN) {

**for** (i = 0; i < n; i++) scanf("%d", &A[i]);

**for** (i = 0; i < m; i++) scanf("%d", &B[i]);

l = LCS(A, n, B, m, s);

**for** (i = 0; i < l; i++) printf("%d ", s[i]);

printf("\nLen = %d\n", l);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Dynamic Programming:*

## Maximum Submatrix Sum

*=================================================================*

*Description: Given a matrix with n rows and m columns, find*

*the rectangular submatrix with the largest sum.*

*Complexity: O(n\*m^2)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: December 09, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 3 (Spain 108, 836, 10074)*

*Notes: This code can easily be converted to work with a*

*matrix of doubles. Matricies have the coordinate (0,0) in*

*the upper left corner and (n-1,m-1) in the lower right.*

*Submatricies are inclusive, i.e. the submatrix*

*(t=5, b=6, l=3, r=4) contains the elements:*

*(5,3), (5,4), (6,3), (6,4).*

*The result is not guaranteed to be unique.*

*\*/*

*/\* fix compile warning \*/*

#include <stdio.h>

#define MAXN 100

#define Atype double

**typedef** **struct** {

**int** top, bot, left, right;

Atype sum;

} submat\_t;

submat\_t max\_submatrix(Atype vals[MAXN][MAXN], **int** n, **int** m) {

submat\_t best, prev[MAXN][MAXN];

**int** row, i, j;

Atype sum;

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

**for** (j = 0; j <= i; j++) prev[i][j].sum = -1;

best.sum = vals[0][0];

best.left = best.right = best.top = best.bot = 0;

**for** (row = 0; row < n; row++) {

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

**for** (sum = 0, j = i; j < m; j++) {

sum += vals[row][j];

**if** (prev[i][j].sum <= 0) {

prev[i][j].sum = 0;

prev[i][j].top = row;

}

prev[i][j].sum += sum;

**if** (prev[i][j].sum > best.sum) { */\* put tie-breaking here \*/*

best = prev[i][j];

best.right = j;

best.left = i;

best.bot = row;

}

}

}

}

**return** best;

}

*/\* If you have matricies that are long and skinny (like 7x40 or 100x20)*

*call this function instead of max\_submatrix. If the problem can be*

*solved faster by transposing the matrix, this function will*

*automatically determine that and solve the problem. It's usage*

*is identical to max\_submatrix. \*/*

submat\_t max\_submat\_t(Atype mat[MAXN][MAXN], **int** n, **int** m) {

Atype transp[MAXN][MAXN];

**int** i, j, tmp;

submat\_t res;

**if** (m <= n) **return** max\_submatrix(mat, n, m);

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

**for** (j = 0; j < n; j++) transp[j][i] = mat[i][j];

res = max\_submatrix(transp, m, n);

tmp = res.top;

res.top = res.left;

res.left = tmp;

tmp = res.bot;

res.bot = res.right;

res.right = tmp;

**return** res;

}

**int** main() {

Atype mat[100][100];

**int** nrows, ncols, i, j;

submat\_t max;

**while** (scanf("%d %d", &nrows, &ncols)==2) {

**for** (i = 0; i < nrows; i++)

**for** (j = 0; j < ncols; j++)

scanf("%lf",&mat[i][j]);

max = max\_submatrix(mat,nrows,ncols);

printf("(%d,%d) (%d,%d) has a sum of %g.\n",

max.top, max.left, max.bot, max.right, max.sum);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Generators

############################################

*/\* Generator:*

## Catalan Numbers

*=================================================================*

*Description: Generates the first few catalan numbers*

*Complexity: O(n)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Dec 07, 2002*

*References: None.*

*-----------------------------------------------------------------*

*Reliability: 1*

*Notes: The sequence of catalan numbers grows pretty quickly.*

*For n > 33, long longs are too small, so use the same*

*algorithm with bignum.*

*You may want to tweak the zeroth catalan number so that it*

*has meaning in your application (for instance, how many*

*ways are there to bracket zero terms?*

*\*/*

#include <stdio.h>

**long** **long** **int** cat[33];

**void** getcat() {

**int** i;

cat[0] = cat[1] = 1;

**for** (i = 2; i < 33; i++)

cat[i] = cat[i-1]\*(4\*i-6)/i;

}

**int** main() {

**int** i;

getcat();

**for** (i = 0; i < 33; i++) {

printf("%lld\n",cat[i]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Generators:*

## Binary Strings generator - (ordered by cardinality)

*=================================================================*

*Description: Generates all possible binary strings of size N*

*ordered in increasing order by the number of bits*

*set, then lexographically.*

*For example: when N = 3, the routine generates:*

*0 0 0 ] Cardinality 0*

*0 0 1 ]*

*0 1 0 | Cardinality 1*

*1 0 0 ]*

*0 1 1 ]*

*1 0 1 | Cardinality 2*

*1 1 0 ]*

*1 1 1 ] Cardinality 3*

*Complexity: O(2^N)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 9, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: This routine may be useful if we want to find the*

*solution with the lowest cardinality.*

*For example, you may want to break off a search*

*after a certain number of solutions have been found.*

*\*/*

#include <stdio.h>

#define MAXN 20

**char** bit[MAXN];

*/\* Add whatever work needs to be done in this routine \*/*

*/\* For this sample, we just print out the bits \*/*

**void** Process(**int** n){

**int** i;

**for**(i = 0; i < n; i++) printf("%2d", bit[i]);

printf("\n");

}

**void** recurse(**int** n, **int** curr, **int** left){

**if**(curr == n){

Process(n);

} **else** {

**if**(curr+left < n){

bit[curr] = 0;

recurse(n, curr+1, left);

}

**if**(left){

bit[curr] = 1;

recurse(n, curr+1, left-1);

}

}

}

*/\* This function generates the strings based on cardinality*

*You may wish to add break conditions in this routine if*

*necessary \*/*

**void** gen\_bin\_card(**int** n){

**int** i;

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

printf("Cardinality %d:\n", i);

recurse(n, 0, i);

}

}

**int** main(){

**int** n;

**while**(scanf("%d", &n) == 1){

printf("Binary string length N = %d\n", n);

gen\_bin\_card(n);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Generators:*

## K-Subset functions

*=================================================================*

*Description: Some functions dealing with subsets of {0,1,...,N-1}*

*of size K.*

*Subsets are represented by integer arrays, of size*

*K, with values in the range [0,N-1]*

*============ Lexicographically ordered =============*

*rank\_lexkset(A,n,k) returns the rank (i.e the index*

*in the lexicographically sorted list of all*

*K-subsets) of subset A.*

*unrank\_lexkset(r,n,k,A) builds the rth*

*lexicographic K-subset into the array A*

*next\_kset(A,B,n,k) builds the K-subset that comes*

*lexicographically after A into the array B. The*

*function returns 1 if this is done sucessfully, and*

*0 otherwise*

*=========== Co-lexicographically ordered ===========*

*rank\_colexkset(A,n,k) returns the rank of A in the*

*co-lexicographically sorted list*

*unrank\_colexkset(r,n,k,A) builds the rth*

*co-lexicographic K-subset into the array A*

*============= Minimum change ordered ===============*

*rank\_minkset(A,n,k) returns the rank of A in a*

*minimal change ordering known as revolving door*

*unrank\_minkset(r,n,k,A) builds the rth K-subset*

*of the minimum change ordering*

*minchange(r,n,k,plus,minus) sets plus to the*

*index that is added, and minus to the index that*

*is removed from the rth K-subset to get to the*

*(r+1)th K-subset. The function returns 1*

*if this is done sucessfully, and 0 otherwise*

*Complexity: rank\_lexkset => O(n)*

*unrank\_lexkset => O(n)*

*next\_lexkset => O(k)*

*rank\_colexkset => O(k)*

*unrank\_colexkset => O(n)*

*unrank\_minkset => O(n)*

*minchange => O(n+k^2)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Jan 17, 2003*

*References: Combinatorial Algorithms - Donald L. Kreher*

*Chapter 2.3*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: bin[n][k] stores n choose k. This may be redone*

*using arbitrary precision arithmetic, if necessary*

*\*/*

#include <stdio.h>

#define MAXN 200

#define MAXK 12

#define ULL unsigned long long

#define min(a,b) (((a)<(b))?(a):(b))

ULL bin[MAXN+1][MAXK+1];

*/\* ============ Lexicographically ordered ============= \*/*

ULL rank\_lexkset(**int** \*A, **int** n, **int** k){

ULL r; **int** i, j;

**for**(i = r = 0; i < k; i++)

**for**(j = i?A[i-1]+1:0; j < A[i]; j++)

r += bin[n-1-j][k-1-i];

**return** r;

}

**void** unrank\_lexkset(ULL r, **int** n, **int** k, **int** \*A){

**int** i, x;

**for**(i = x = 0; i < k; i++){

**while**(r >= bin[n-1-x][k-1-i])

r -= bin[n-1-x++][k-1-i];

A[i] = x++;

}

}

**int** next\_lexkset(**int** \*A, **int** \*B, **int** n, **int** k){

**int** i, j;

**for**(i = k-1; i >= 0; i--)

**if**(A[i] != n-k+i){

**for**(j = 0; j < i; j++) B[j] = A[j];

**for**(j = i; j < k; j++) B[j] = A[i]+1+j-i;

**return** 1;

}

**return** 0;

}

*/\* ============ Co-lexicographically ordered ============= \*/*

ULL rank\_colexkset(**int** \*A, **int** n, **int** k){

ULL r; **int** i;

**for**(i = r = 0; i < k; i++)

r += bin[A[i]][k-i];

**return** r;

}

**void** unrank\_colexkset(ULL r, **int** n, **int** k, **int** \*A){

**int** i, x = n;

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

**while**(bin[x][k-i] > r) x--;

A[i] = x;

r -= bin[x][k-i];

}

}

*/\* ============ Minimal Change ordering ============= \*/*

ULL rank\_minkset(**int** \*A, **int** n, **int** k){

ULL r; **int** s = 1, i;

**if**(k % 2) r = -1; **else** r = 0;

**for**(i = k; i > 0; i--){

r += s\*bin[A[i-1]+1][i];

s \*= -1;

}

**return** r;

}

**void** unrank\_minkset(ULL r, **int** n, **int** k, **int** \*A){

**int** x = n, i;

**for**(i = k; i > 0; i--){

**while**(bin[x][i] > r) x--;

A[i-1] = x;

r = bin[x+1][i]-r-1;

}

}

**int** minchange(**int** r, **int** n, **int** k, **int** \*plus, **int** \*minus){

**int** A[MAXK], B[MAXK], i, j;

**if**(r == bin[n][k]-1) **return** 0;

unrank\_minkset(r, n, k, A);

unrank\_minkset(r+1, n, k, B);

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

**for**(j = 0; j < k; j++)

**if**(A[i] == B[j]){

B[j] = -1;

**break**;

}

**if**(j == k) \*minus = A[i];

}

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

**if**(B[i] != -1){

\*plus = B[i];

**break**;

}

}

**return** 1;

}

**void** buildBin(){

**int** n,k;

**for**(n = 0; n <= MAXN; n++){

bin[n][0] = 1;

**for**(k = 1; k <= min(n,MAXK); k++)

bin[n][k] = bin[n-1][k]+bin[n-1][k-1];

}

}

**int** main(){

**int** A[MAXK], B[MAXK], n, k, i, j, plus, minus;

buildBin(); */\* Build table of binomial coefficients \*/*

**while**(scanf("%d %d", &n, &k) == 2){

**for**(i = 0; i < bin[n][k]; i++){

printf("Rank[%d] Lex:", i);

unrank\_lexkset(i, n, k, A);

**for**(j = 0; j < k; j++) printf("%d ", A[j]);

printf("{%llu} Co-lex:", rank\_lexkset(A,n,k));

unrank\_colexkset(i, n, k, B);

**for**(j = 0; j < k; j++) printf(" %d", B[j]);

printf(" {%llu} Minchange:", rank\_colexkset(B, n, k));

unrank\_minkset(i, n, k, A);

**for**(j = 0; j < k; j++) printf(" %d", A[j]);

**if**(minchange(i, n, k, &plus, &minus)){

printf(" [+%d][-%d] {%llu}\n",

plus, minus, rank\_minkset(A, n, k));

} **else** {

printf(" No change {%llu}\n", rank\_minkset(A, n, k));

}

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Generators: N-Queens Solution Generator*

*=================================================================*

*Description: This code generates a solution for the n-queens*

*problem. Solutions exist for n = 1 || n > 3.*

*Solutions are stored in the array called row[],*

*where row[i] represents the row where the queen in*

*the i'th column is placed. Coordinates are [0..n-1]*

*Complexity: O(n)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 9, 2002*

*References: www.bridges.canterbury.ac.nz/features/eight.html*

*-----------------------------------------------------------------*

*Reliability: 1 (Sept 2002)*

*Notes: - This solution is only 1 of perhaps many*

*solutions*

*- No checking is done by the routine for valid n*

*\* Solves Spanish Problem 10094*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 10001

**int** row[MAXN];

**void** gen\_nqueens(**int** n){

**int** i;

**if**(n % 2){

gen\_nqueens(n-1);

row[n-1] = n-1;

} **else** {

**if**(n % 6 == 2){

**for**(i = 0; i < n/2; i++) row[i] = (n/2+2\*i-1)%n;

**for**(i = n/2; i < n; i++) row[i] = (n/2+2\*i+2)%n;

} **else** {

**for**(i = 0; i < n/2; i++) row[i] = 2\*i+1;

**for**(i = n/2; i < n; i++) row[i] = (2\*i)%n;

}

}

}

**int** main(){

**int** i, n;

**while**(scanf("%d", &n) == 1){

**if**(n == 1 || n > 3){

gen\_nqueens(n);

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

printf("%d%c", row[i], i == n-1 ? '\n' : ' ');

} **else** {

printf("Impossible\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Generators:*

## Pythagorean Triples

*=================================================================*

*Description: A Pythagorean triple is a set of three positive*

*integers a < b < c such that a^2 + b^2 = c^2. A*

*primitive triple has the property that a,b,c are*

*relatively prime. Given N, this code generates*

*all the primitive triples such that c < N.*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: Dec 15, 2002*

*References: Leonhard Euler, so you know it's fast.*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 106)*

*Notes: There will be no more than N/4 triples generated;*

*allocate this much space. The function will*

*return the actual number of triples generated.*

*The triples will be in no particular order; you may*

*want to sort the results after generating them.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

**typedef** **struct** {

**int** a, b, c;

} ptrip;

**int** gcd(**int** a, **int** b) {

**int** r;

**while** (b) {

r = a % b;

a = b;

b = r;

}

**return** a;

}

**int** gen\_triples(**int** n, ptrip \*res) {

**int** a, b, p, q, cnt = 0;

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

**for** (q = 1+p%2; q < p && p\*p+q\*q < n; q += 2) {

**if** (gcd(p,q)!=1) **continue**;

a = p\*p-q\*q;

b = 2\*p\*q;

**if** (a < b) { res[cnt].a = a; res[cnt].b = b; }

**else** { res[cnt].a = b; res[cnt].b = a; }

res[cnt++].c = p\*p+q\*q;

}

}

**return** cnt;

}

**int** main() {

ptrip \*arra = NULL;

**int** i, max, ntrip;

**while** (scanf("%d",&max)==1) {

arra = realloc(arra,**sizeof**(ptrip)\*max/4);

ntrip = gen\_triples(max,arra);

printf("%d\n",ntrip);

**for** (i = 0; i < ntrip; i++)

printf("%d %d %d\n", arra[i].a, arra[i].b, arra[i].c);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Graph Theory

############################################

*/\* Graph Theory:*

## Articulation Points in a Graph (Adj List version)

*=================================================================*

*Description: An articulation point in a undirected graph is a*

*vertex which disconnects the graph when removed.*

*This routine takes a graph represented by an*

*adjacency list, and finds all articulation points in*

*the graph.*

*If b is an articulation point, then there exists*

*two distinct vertices a and c, such that b is on*

*EVERY path from a to c.*

*- An array called ART[] contains 1 if node i is an*

*articulation point and 0 otherwise.*

*- use addEdge(int x, int y) to create a undirected*

*edge between node x and y*

*- use clearList() to clear out all elements of the*

*adjacency list*

*Complexity: O(V+E)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Oct 23, 2002 (mod Jan 16, 2003)*

*References: Algorithms in C, Robert Sedgewick pg.440-441*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - Nodes must be indexed from 0..N-1.*

*- Edges should not be added more than once to*

*prevent array overflow*

*- On graphs which are not connected, nodes which*

*disconnect components they are part of are*

*considered articulation points.*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 200

#define min(a,b) (((a)<(b))?(a):(b))

**typedef** **struct**{

**int** deg;

**int** adj[MAXN];

} Node;

Node alist[MAXN];

**char** ART[MAXN], val[MAXN];

**int** id;

**void** addEdge(**int** x, **int** y){

alist[x].adj[alist[x].deg++] = y;

alist[y].adj[alist[y].deg++] = x;

}

**void** clearList(){

memset(alist, 0, **sizeof**(alist));

}

**int** visit(**int** x, **int** root){

**int** i, y, m, res, child = 0;

res = val[x] = ++id;

**for**(i = 0; i < alist[x].deg; i++){

y = alist[x].adj[i];

**if**(!val[y]){

**if**(root && ++child > 1) ART[x] = 1;

m = visit(y, 0);

res = min(res, m);

**if**(m >= val[x] && !root) ART[x] = 1;

} **else** {

res = min(val[y], res);

}

}

**return** res;

}

**void** articulate(**int** n){

**int** i;

memset(ART, 0, **sizeof**(ART));

memset(val, 0, **sizeof**(val));

**for**(id = i = 0; i < n; i++)

**if**(!val[i]) visit(i, 1);

}

**int** main(){

**int** i, n, m, x, y, found;

*/\* Read in number of vertices, number of edges \*/*

**while**(scanf("%d %d", &n, &m) == 2){

*/\* Read in edge between node x and node y \*/*

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

scanf("%d %d", &x, &y);

addEdge(x,y);

}

*/\* Find articulation points \*/*

articulate(n);

**for**(found = i = 0; i < n; i++)

**if**(ART[i]){

printf("Node %d is an articulation point\n", i);

found = 1;

}

**if**(!found) printf("No articulation points\n");

clearList();

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Bellman-Ford Algorithm - negative weigth cycles

*=================================================================*

*Description: Given a directed graph G, expressed in terms of its*

*edges, and a source vertex s, Bellman-Ford returns*

*whether or not it is possible to encounter a cycle*

*of negative weight. If such a cycle exists, it*

*returns 0, otherwise 1.*

*If no negative-weight cycle exists, then dist[v]*

*contains the shortest distance from source s to*

*v.*

*Complexity: O(NM) where N is # of nodes, M is # of edges*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 20, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 successful use (Sept 2002)*

*(Spain 558)*

*Notes: INF must be set to a value larger than any edge*

*\*/*

#include <stdio.h>

#include <string.h>

#include <limits.h>

#define INF INT\_MAX

#define MAXNODES 1005

#define MAXEDGES 2005

**typedef** **struct**{

**int** u,v,w;

} Edge;

**int** dist[MAXNODES], pred[MAXNODES];

Edge edge[MAXEDGES];

**int** n, m;

**int** BellmanFord(**int** source){

**int** i, j;

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

dist[i] = INF;

pred[i] = -1;

}

dist[source] = 0;

**for**(i = 0; i < n-1; i++)

**for**(j = 0; j < m; j++){

**if**(dist[edge[j].v] > dist[edge[j].u]+edge[j].w){

dist[edge[j].v] = dist[edge[j].u]+edge[j].w;

pred[edge[j].v] = edge[j].u;

}

}

**for**(j = 0; j < m; j++){

**if**(dist[edge[j].v] > dist[edge[j].u]+edge[j].w) **return** 0;

}

**return** 1;

}

**int** main(){

**int** tnum, i;

scanf("%d", &tnum);

**while**(tnum--){

scanf("%d %d", &n, &m);

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

scanf("%d %d %d", &edge[i].u, &edge[i].v, &edge[i].w);

}

**if**(BellmanFord(0)) printf("not ");

printf("possible\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Maximum Bipartite Matching

*=================================================================*

*Description: Given a bipartite graph G = {U,V,E}, this routine*

*computes the maximum bipartite matching.*

*For vertex i of set U:*

*match[i] = -1 means i is not matched*

*match[i] = x means the edge i->(x-|U|) is selected*

*\*\*\*\*\*\*\*\*\*\**

*For simplicity, use addEdge(i,j,n) to add edges, where*

*0 <= i < |U| and 0 <= j < |V| and |U| = n.*

*If there is an edge from vertex i of U to vertex*

*j of V then: e[i][j+|U|] = e[j+|U|][i] = 1.*

*\*\*\*\*\*\* \*\*\*\*\**

*Complexity: O(U+V) where U and V are the sizes of the sets*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: May 6, 2003*

*References: Waterloo code*

*-----------------------------------------------------------------*

*Reliability: 2 (Spain 10080 - Golpher holes II,*

*Housing Complexes Mar 8th, 2003)*

*Notes: - If |U| = n and |V| = m, then vertices are assumed*

*to be from [0,n-1] in set U and [0,m-1] in set V.*

*- Remember that match[i]-n gives the edge from i,*

*not just match[i].*

*- This code is roughly 2 times slower than the old*

*code since it doesn't try multiple BFS paths at*

*once, however, it's about 4 times shorter...*

*A bipartite graph is one where the vertices can be*

*separated into two disjoint sets U and V, such that*

*all edges in the graph are between a vertex in U and*

*a vertex iv V. A maximum matching on such a graph*

*is the selection of the most edges such that each*

*vertex is adjacent to at most one edge.*

*The maximum matching is equivalent to the minimal*

*vertex cover in a bipartite graph.*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 300 */\* How many vertices in U+V (in total) \*/*

**char** e[MAXN][MAXN]; */\* MODIFIED Adj. matrix (see note) \*/*

**int** match[MAXN], back[MAXN], q[MAXN], tail;

**void** addEdge(**int** x, **int** y, **int** n){

e[x][y+n] = e[y+n][x] = 1;

}

**int** find(**int** x, **int** n, **int** m){

**int** i, j, r;

**if**(match[x] != -1) **return** 0;

memset(back, -1, **sizeof**(back));

**for**(q[i=0]=x, tail = 1; i < tail; i++)

**for**(j = 0; j < n+m; j++){

**if**(!e[q[i]][j]) **continue**;

**if**(match[j] != -1){

**if**(back[j] == -1){

back[j] = q[i];

back[q[tail++] = match[j]] = j;

}

} **else** {

match[match[q[i]] = j] = q[i];

**for**(r = back[q[i]]; r != -1; r = back[back[r]])

match[match[r] = back[r]] = r;

**return** 1;

}

}

**return** 0;

}

**void** bipmatch(**int** n, **int** m){

**int** i;

memset(match, -1, **sizeof**(match));

**for**(i = 0; i < n+m; i++) **if**(find(i,n,m)) i = 0;

}

**int** main(){

**int** n, m, esize, x, y;

**int** i, count;

*/\* Read size of set U into n, size of set V into m \*/*

**while**(scanf("%d %d", &n, &m) == 2){

memset(e, 0, **sizeof**(e)); */\* Clear edges \*/*

scanf("%d", &esize); */\* get # of edges \*/*

**while**(esize--){

scanf("%d %d", &x, &y); */\* add edges \*/*

addEdge(x,y,n); */\* Edges [0,n-1]->[0,m-1] \*/*

}

bipmatch(n, m); */\* Perform matching \*/*

**for**(count = i = 0; i < n; i++){ */\* Print results \*/*

**if**(match[i] != -1){

printf("%d->%d\n", i, match[i]-n);

count++;

}

}

printf("Matching size: %d\n", count);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Articulation Points & Bridges (adj list)

*==========================================================*

*Description: An articulation point in a graph is a vertex that*

*will result in a connected graph being disconnected*

*upon removal. A bridge is an edge that will*

*result in a connected graph being disconnected*

*if it is severed.*

*-array entry art[v] is true iff vertex v is an*

*articulation point*

*-array entries bridge[i][0] and bridge[i][1] are*

*the endpoints of a bridge in the graph. Note*

*that if bridge (u,v) is represented in the array,*

*then (v,u) is not.*

*-variable bridges is the number of bridges in the graph*

*Complexity O(V + E)*

*-----------------------------------------------------------*

*Author: Zac Friggstad*

*Date: Jan 13, 2006*

*References: "The Design & Analysis of Computer Algorithms"*

*Aho, Hopcroft, Ullman*

*-----------------------------------------------------------*

*Reliability: 610 on Spain*

*Notes: -index vertices from 0 to n-1*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAX\_N 200

#define min(a,b) (((a)<(b))?(a):(b))

**typedef** **struct** {

**int** deg;

**int** adj[MAX\_N];

} Node;

Node alist[MAX\_N];

**bool** art[MAX\_N];

**int** df\_num[MAX\_N], low[MAX\_N], father[MAX\_N], count;

**int** bridge[MAX\_N\*MAX\_N][2], bridges;

**void** add\_edge(**int** v1, **int** v2) {

alist[v1].adj[alist[v1].deg++] = v2;

alist[v2].adj[alist[v2].deg++] = v1;

}

**void** add\_bridge(**int** v1, **int** v2) {

bridge[bridges][0] = v1;

bridge[bridges][1] = v2;

++bridges;

}

**void** clear() {

**for** (**int** i = 0; i < MAX\_N; ++i) alist[i].deg = 0;

}

**void** search(**int** v, **bool** root) {

**int** w, child = 0;

low[v] = df\_num[v] = count++;

**for** (**int** i = 0; i < alist[v].deg; ++i) {

w = alist[v].adj[i];

**if** (df\_num[w] == -1) {

father[w] = v;

++child;

search(w, **false**);

**if** (low[w] > df\_num[v]) add\_bridge(v, w);

**if** (low[w] >= df\_num[v] && !root) art[v] = **true**;

low[v] = min(low[v], low[w]);

}

**else** **if** (w != father[v]) {

low[v] = min(low[v], df\_num[w]);

}

}

**if** (root && child > 1) art[v] = **true**;

}

**void** articulate(**int** n) {

**int** child = 0;

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

art[i] = **false**;

df\_num[i] = -1;

father[i] = -1;

}

count = bridges = 0;

search(0, **true**);

}

**int** main() {

**int** n, m, v1, v2, c = 0;

**while** (**true**) {

scanf("%d %d", &n, &m);

**if** (!n && !m) **break**;

clear();

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

scanf("%d %d", &v1, &v2);

add\_edge(v1 - 1, v2 - 1);

}

articulate(n);

printf("Articulation Points:");

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

**if** (art[i]) printf(" %d", i + 1);

printf("\n");

printf("Bridges:");

**for** (**int** i = 0; i < bridges; ++i)

printf(" (%d,%d)", bridge[i][0] + 1, bridge[i][1] + 1);

printf("\n\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Eulerian Graphs

*=================================================================*

*Description: This routine establishes if an undirected graph is*

*Eulerian. If a graph has an Eulerian path, or an*

*Eulerian cycle, the buildPath() routine generates*

*the sequence of nodes visited in the global array*

*seq[] of size 'seqsize'*

*Before adding edges, call Init() to initialize all*

*necessary data structures.*

*Use the provided function addEdge(x,y,c) which*

*adds c number of edges between x and y.*

*isEulerian(int n, int \*start, int \*end) returns:*

*0 if the graph is not Eulerian*

*1 if the graph has a Euler cycle*

*2 if the graph a path, from start to end*

*with n being the number of nodes in the graph*

*For more information about Euler cycles and*

*Euler paths, see the Notes section.*

*Complexity: O(N) where N is the number of edges*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Jan 8, 2003*

*Mar 13, 2003 - Update to handle vertices with no*

*edges*

*References:*

*-----------------------------------------------------------------*

*Reliability: 2 (Problem B: Floor - Set from 2003 Jan 04)*

*(Problem A: Door Man - Set from 2003 Mar 13)*

*Notes: - Indexing of nodes MUST be from [0...n-1], since*

*the connectivity is checked using node 0.*

*- buildPath modifies the original graph g[][].*

*Uncomment the marked regions to keep a copy of*

*the graph in g2[][]. This also allows for*

*multiple calls to buildPath for the same input*

*graph*

*- A Euler cycle in an undirected graph is a cycle*

*which travels along every edge of the graph,*

*with the first node being the same as the last*

*node*

*- A Euler path in an undirected graph is a path*

*which travels along every edge of the graph,*

*but the first node does not necessarily need*

*to be identical to the last node*

*- A graph has a Euler cycle if it is connected,*

*and all nodes have even degree. If the graph*

*is connected, and only two nodes have odd*

*degree a Euler path exists starting and ending*

*from those two odd-degree nodes*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 105 */\* Number of nodes \*/*

#define MAXM 505 */\* Maximum number of edges \*/*

#define min(a,b) (((a)<(b))?(a):(b))

#define max(a,b) (((a)>(b))?(a):(b))

#define DEC(a,b) g[a][b]--;g[b][a]--;deg[a]--;deg[b]--

**int** sets[MAXN], deg[MAXN];

**int** g[MAXN][MAXN];

**int** seq[MAXM], seqsize;

*/\* Uncomment if you need copy of graph*

*int g2[MAXN][MAXN], deg2[MAXN];*

*\*/*

**int** getRoot(**int** x){

**if**(sets[x] < 0) **return** x;

**return** sets[x] = getRoot(sets[x]);

}

**void** Union(**int** a, **int** b){

**int** ra = getRoot(a), rb = getRoot(b);

**if**(ra != rb){

sets[ra] += sets[rb];

sets[rb] = ra;

}

}

**void** Init(){

memset(sets, -1, **sizeof**(sets));

memset(g, 0, **sizeof**(g));

memset(deg, 0, **sizeof**(deg));

}

**void** addEdge(**int** x, **int** y, **int** count){

g[x][y] += count; deg[x] += count;

g[y][x] += count; deg[y] += count;

Union(x,y);

}

**int** isEulerian(**int** n, **int** \*start, **int** \*end){

**int** odd = 0, i, count = 0, x;

*/\* Check if graph is connected. If all vertices*

*are guaranteed to be used then use this:*

*if(sets[getRoot(0)] != -n) return 0;*

*Otherwise, count only vertices used like this: \*/*

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

**if**(deg[i]){

x = i; count++;

}

**if**(sets[getRoot(x)] != -count) **return** 0;

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

**if**(deg[i]%2){

odd++;

**if**(odd == 1) \*start = i;

**else** **if**(odd == 2) \*end = i;

**else** **return** 0;

}

}

**return** odd ? 2 : 1;

}

**void** getPath(**int** n, **int** start, **int** end){

**int** temp[MAXM], tsize = 1, i, j;

temp[0] = start;

**while**(1){

j = temp[tsize-1];

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

**if**(i == end) **continue**;

**if**(g[i][j]){

temp[tsize++] = i;

DEC(i,j);

**break**;

}

}

**if**(i == n){

**if**(g[end][j]){

temp[tsize++] = end;

DEC(j,end);

}

**break**;

}

}

**for**(i = 0; i < tsize; i++)

**if**(!deg[temp[i]]) seq[seqsize++] = temp[i];

**else** getPath(n, temp[i], temp[i]);

}

**void** buildPath(**int** n, **int** start, **int** end){

seqsize = 0;

*/\* Uncomment if you need copy of graph*

*memcpy(g, g2, sizeof(g));*

*memcpy(deg, deg2, sizeof(deg));*

*\*/*

getPath(n, start, end);

}

**int** main(){

**int** i, x,y,start,end, n, m;

**while**(scanf("%d %d", &n, &m) == 2){

Init();

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

scanf("%d %d", &x, &y);

addEdge(x,y,1);

}

*/\* Uncomment if you need copy of graph*

*memcpy(g2, g, sizeof(g2));*

*memcpy(deg2, deg, sizeof(deg2));*

*\*/*

**switch**(isEulerian(n, &start, &end)){

**case** 0:

printf("Graph is not Eulerian\n"); **break**;

**case** 1:

printf("Graph has an Eulerian Cycle\n");

buildPath(n, 0, 0);

**for**(i = 0; i < seqsize; i++) printf("%d ", seq[i]);

printf("\n"); **break**;

**case** 2:

printf("Graph has an Eulerian path from %d to %d\n", start, end);

buildPath(n,start,end);

**for**(i = 0; i < seqsize; i++) printf("%d ", seq[i]);

printf("\n"); **break**;

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Directed Eulerian Graphs

*=================================================================*

*Description: Template for finding directed Eulerian paths/cycles*

*This is written so that it returns the*

*lexicographically first such path/cycle.*

*Complexity: O(M) where M is the # of edges*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Nov 10, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 2 (Problem C Catenyms - Jan 25, 2003)*

*(Problem H Play on Words - Nov 8, 2003)*

*Notes: This is the direct code for solving Catenyms.*

*Here is the underlying idea:*

*1. Count the indegree/outdegree of each node.*

*2. Check if the graph is connected. (done here with*

*Unionfind). If not, then not Eulerian.*

*3. A directed Euler cycle exists if for each*

*node the in degree = out degree.*

*4. A directed Euler path exists if there is exactly*

*source node with outdegree = 1+in degree and one*

*sink node with indegree = 1+out degree*

*5. Sort the edges in the order that you need.*

*6. Do a search for a path, choosing lexicographically*

*first when possible, then build from back,*

*splicing when necessary.*

*\*/*

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

*/\* Maximum number of nodes, max number of edges \*/*

#define MAXN 26

#define MAXM 1001

**typedef** **struct**{

**int** src, dst;

**char** name[21], used;

} Edge;

Edge e[MAXM];

**int** esize;

**int** sets[MAXN], odeg[MAXN], ideg[MAXN];

**int** seq[MAXM], seqsize;

**char** used[MAXN];

**int** cmp\_Edge(Edge \*a, Edge \*b){

**return** strcmp(a->name, b->name);

}

**int** getRoot(**int** x){

**return** sets[x] < 0 ? x : (sets[x] = getRoot(sets[x]));

}

**void** Union(**int** a, **int** b){

**int** ra = getRoot(a), rb = getRoot(b);

**if**(ra != rb){

sets[ra] += sets[rb];

sets[rb] = ra;

}

}

**void** getPath(**int** start, **int** end){

**int** temp[MAXM], tsize = 0, i, curr;

**for**(i = 0; i < esize; i++){

**if**(e[i].used || e[i].src != start) **continue**;

temp[tsize++] = i;

e[i].used = 1;

odeg[e[i].src]--;

curr = e[i].dst;

**break**;

}

**while**(curr != end){

**for**(i = 0; i < esize; i++){

**if**(e[i].used || e[i].src != curr) **continue**;

temp[tsize++] = i;

e[i].used = 1;

odeg[e[i].src]--;

curr = e[i].dst;

**break**;

}

}

**for**(i = tsize-1; i >= 0; i--){

**if**(odeg[e[temp[i]].dst]) getPath(e[temp[i]].dst, e[temp[i]].dst);

seq[seqsize++] = temp[i];

}

}

**void** init(){

memset(sets, -1, **sizeof**(sets));

memset(odeg, 0, **sizeof**(odeg));

memset(ideg, 0, **sizeof**(ideg));

memset(used, 0, **sizeof**(used));

}

**int** main(){

**int** tnum, i, start, end, flag, count;

**for**(scanf("%d", &tnum); tnum; tnum--){

init();

count = 0;

*/\* Read edges, in this case: words \*/*

**for**(scanf("%d", &esize), i = 0; i < esize; i++){

scanf(" %s", e[i].name);

*/\* Src is the first letter of word, dst is last letter \*/*

e[i].src = e[i].name[0]-'a';

e[i].dst = e[i].name[strlen(e[i].name)-1] - 'a';

e[i].used = 0;

*/\* Increment in/out degrees \*/*

odeg[e[i].src]++; ideg[e[i].dst]++;

*/\* Keep track of # of vertices in use \*/*

**if**(!used[e[i].src]){

count++;

used[e[i].src] = 1;

}

**if**(!used[e[i].dst]){

count++;

used[e[i].dst] = 1;

}

Union(e[i].src, e[i].dst);

}

*/\* Sort edges lexicographically \*/*

qsort(e, esize, **sizeof**(e[0]), (**void** \*) cmp\_Edge);

*/\* Check if the graph is connected \*/*

**if**(sets[getRoot(e[0].src)] != -count){

*/\* This means its not connected so no Euler path\*/*

printf("\*\*\*\n");

**continue**;

}

*/\* Determine src / dst if it exists \*/*

*/\* Flag is used to mark if the graph is not Eulerian \*/*

flag = 0;

start = end = -1;

**for**(i = 0; i < 26; i++){

**if**(!used[i]) **continue**;

**if**(ideg[i] == odeg[i]+1){

**if**(end == -1) end = i;

**else** {

flag = 1;

**break**;

}

} **else** **if**(odeg[i] == ideg[i]+1){

**if**(start == -1) start = i;

**else** {

flag = 1;

**break**;

}

} **else** **if**(odeg[i] != ideg[i]){

flag = 1;

**break**;

}

}

**if**(flag){

*/\* Graph is not Eulerian \*/*

printf("\*\*\*\n");

**continue**;

} **else** {

*/\* Graph has an Eulerian path or Eulerian cycle \*/*

*/\* Print out a eulerian path/cycle from start to end \*/*

**if**(start == -1 && end == -1){

*/\* Graph has euler cycle \*/*

**for**(i = 0; i < 26; i++)

**if**(used[i]) **break**;

seqsize = 0;

getPath(i, i);

**for**(i = seqsize-1; i >= 0; i--)

printf("%s%c", e[seq[i]].name, i ? '.' : '\n');

} **else** {

*/\* Graph has Euler path \*/*

seqsize = 0;

getPath(start, end);

**for**(i = seqsize-1; i >= 0; i--)

printf("%s%c", e[seq[i]].name, i ? '.' : '\n');

}

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Floyd's Algorithm - All pairs shortest path

*=================================================================*

*Description: Given a graph stored in an adjacency matrix, returns*

*the shortest distance from node i to node j in*

*d[i][j]. Weights of each edge must be nonnegative,*

*and -1 is used to indicate an empty edge*

*Complexity: O(N^3)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 17, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 Successful use (Sept 2002)*

*(Spain Problem 10354)*

*Notes: Costs may be changed to doubles, but then elements*

*need to be initialized individually (commented out*

*below).*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 101

#define CType int

CType g[MAXN][MAXN], dist[MAXN][MAXN];

**void** floyd(**int** n){

**int** i, j, k;

memcpy(dist, g, **sizeof**(g));

**for**(k = 0; k < n; k++) **for**(i = 0; i < n; i++) **for**(j = 0; j < n; j++){

**if**(dist[i][k] != -1 && dist[k][j] != -1){

CType temp = dist[i][k] + dist[k][j];

**if**(dist[i][j] == -1 || dist[i][j] > temp)

dist[i][j] = temp;

}

}

**for**(i = 0; i < n; i++) dist[i][i] = 0;

}

**int** main(){

**int** i, j, n;

CType w;

scanf("%d", &n);

*/\* Clear graph \*/*

memset(g, -1, **sizeof**(g));

*/\* If using doubles:*

*for(i = 0; i < n; i++) for(j = 0; j < n; j++) g[i][j] = -1;*

*\*/*

*/\* Read graph \*/*

**while**(scanf("%d %d %d", &i, &j, &w) == 3){

g[i][j] = g[j][i] = w;

}

floyd(n);

**while**(scanf("%d %d", &i, &j) == 2){

printf("%d %d: %d\n", i, j, dist[i][j]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph\_Theory:*

## Maximum Flow in a directed graph

*=================================================================*

*Description: Given a directed graph G, a source node, and a sink*

*node, computes the maximum flow from source to sink,*

*and calculates the flow along each edge.*

*In addition, the array S[] is marked to indicate*

*which side of the cut each vertex is on.*

*S[i] = 1 means vertex i is on source side*

*S[i] = 0 means vertex i is on sink side*

*An edge u->v with S[u] == 1 and S[v] == 0 is an*

*edge of the minimum cut.*

*Usage:*

*1. Clear graph as shown in main()*

*2. Add edges u->v by calling addEdge(u,v,cap)*

*3. maxflow(n, source, sink) returns maxflow*

*4. getEdge(u, v) returns a pointer to the edge*

*u->v, NULL if this edge does not exist*

*Complexity: O(|V|^2\*|E|) (Rough approximation)*

*-----------------------------------------------------------------*

*Author: Matthew McNaughton, Gilbert Lee*

*Date: Mar 10, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 2 Spain 10330 - Power Transmission*

*Spain 104?? - Sabotage*

*Notes: - Multiple edges from u to v may be added. They*

*are converted into a single edge with a capacity*

*equal to their sum*

*- Vertices are assumed to be numbered from 0..n-1*

*- The graph is stored as an adjaceny list in "g"*

*The vector<Edge> g[u] contains the edges adj.*

*to the vertex u.*

*- Basic idea is performing a BFS from the source*

*to find an augmenting path -> a path through*

*which flow may be increased. When there are no*

*more such paths, a maximum flow/minimum cut has*

*been found.*

*\*/*

#include <stdio.h>

#include <vector>

**using** **namespace** std;

#define MAXN 200

**typedef** **struct**{

**int** v, cap, flow;

} Edge;

vector<Edge> g[MAXN];

**int** pred[MAXN], maxcap[MAXN], S[MAXN];

**void** addEdge(**int** u, **int** v, **int** cap){

size\_t i; Edge e;

**for**(i = 0; i < g[u].size(); i++) */\* Add cap if edge exists \*/*

**if**(g[u][i].v == v){

g[u][i].cap += cap;

**return**;

}

e.v = v; e.cap = cap; e.flow = 0;

g[u].push\_back(e); */\* Add edge u->v \*/*

**for**(i = 0; i < g[v].size(); i++) */\* Add dummy reverse edge \*/*

**if**(g[v][i].v == u) **return**;

e.v = u; e.cap = 0;

g[v].push\_back(e);

}

Edge \*getEdge(**int** u, **int** v){

**for**(size\_t i = 0; i < g[u].size(); i++)

**if**(g[u][i].v == v) **return** &g[u][i];

**return** 0;

}

**int** maxflow(**int** n, **int** source, **int** sink){

vector<**int**> q; **int** i, v, flow, inc;

size\_t j; Edge \*e1, \*e2;

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

**for**(j = 0; j < g[i].size(); j++)

g[i][j].flow = 0; */\* Clear all flows \*/*

flow = 0;

**while**(1){ */\* BFS to find augmenting path \*/*

memset(S, 0, **sizeof**(S));

S[source] = 1;

q.clear(); q.push\_back(source);

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

v = q[i];

**if**(v == sink) **break**;

**for**(j = 0; j < g[v].size(); j++){

Edge e = g[v][j];

**if**(S[e.v]) **continue**;

**if**(e.cap && e.flow < e.cap){

q.push\_back(e.v); S[e.v] = 1; pred[e.v] = v;

maxcap[e.v] = e.cap - e.flow;

} **else** {

e1 = getEdge(e.v, v);

**if**(e1 && e1->cap && e1->flow > 0){

q.push\_back(e.v); S[e.v] = 1; pred[e.v] = v;

maxcap[e.v] = e1->flow;

}

}

}

}

**if**(v != sink) **break**; */\* No more augmenting paths \*/*

*/\* Calculate flow \*/*

**for**(inc = INT\_MAX, v = sink; v != source; v = pred[v])

inc = min(inc, maxcap[v]);

flow += inc;

*/\* Update flow \*/*

**for**(v = sink; v != source; v = pred[v]){

e1 = getEdge(pred[v], v);

e2 = getEdge(v, pred[v]);

**if**(e1 && e1->cap) e1->flow += inc;

**else** **if**(e2 && e2->cap) e2->flow -= inc;

**if**(e1 && e2 && e1->flow && e2->flow){

**if**(e1->flow > e2->flow){

e1->flow -= e2->flow;

e2->flow = 0;

} **else** {

e2->flow -= e1->flow;

e1->flow = 0;

}

}

}

}

**return** flow;

}

**int** main(){

**int** n; */\* Number of vertices \*/*

**int** m; */\* Number of edges \*/*

**int** source; */\* Source of the flow \*/*

**int** sink; */\* Sink of the flow \*/*

**int** flow; */\* The value of the max flow \*/*

**int** i, u, v, cap;

*/\* n = # of vertices, m = # of edges \*/*

**while**(scanf("%d %d %d %d", &n, &m, &source, &sink) == 4){

*/\* Clear graph \*/*

**for**(i = 0; i < n; i++) g[i].clear();

*/\* Read in m edges \*/*

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

scanf("%d %d %d", &u, &v, &cap);

addEdge(u,v,cap);

}

flow = maxflow(n, source, sink);

printf("The maximum flow: %d\n", flow);

printf("Min-cut edges:\n");

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

**if**(S[i] == 1){

**for**(size\_t j = 0; j < g[i].size(); j++)

**if**(S[g[i][j].v] == 0)

printf("(%d->%d)\n", i, g[i][j].v);

}

printf("Flow values:\n");

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

**for**(size\_t j = 0; j < g[i].size(); j++)

printf("(%d->%d): %d/%d\n", i, g[i][j].v, g[i][j].flow, g[i][j].cap);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Maximum Flow with Lower bounds on edges

*=================================================================*

*Description: Calculates the maximum flow along a network with*

*both upper and lower capacities on edges, from a*

*source node, to a sink node. As a side effect the*

*global array S[] is marked with a 1 if the ith*

*vertex is on the source side of the cut.*

*The function maxflow\_lb returns the maximum flow*

*if one exists, and -1 if a valid flow does not*

*exist.*

*1. Clear graph (shown below)*

*2. Add edges using addEdge(u,v,low,up,g)*

*3. Call maxflow\_lb*

*Complexity: O(|V|^2\*|E|) (Rough approximation)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 18, 2003*

*References: Graph Algorithms - Shimon Even Chapter 5.3*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - This implementation allows for multiple edges*

*between a pair of vertices*

*- the lower bound is assumed to be less than or*

*equal to the upper bound, and greater or*

*equal to zero*

*- No valid flow exists if the minimum flow*

*requirements for some edge can not be met.*

*- How it works:*

*Creates a new auxilary graph with a new source and*

*sink, and assigns upper capacities, while leaving*

*lower bounds 0. Finds a maximum flow on the new*

*graph, and then checks if it is valid. Moves the*

*flow over to the original graph, then perform*

*a maximum flow on the original graph*

*\*/*

#include <stdio.h>

#include <string.h>

#include <vector>

**using** **namespace** std;

#define MAXN 500

**typedef** **struct**{

**int** v, low, up, flow;

} Edge;

vector<Edge> g[MAXN];

vector<Edge> aux[MAXN];

**int** pred[MAXN], maxcap[MAXN], S[MAXN];

**void** addEdge(**int** u, **int** v, **int** low, **int** up, vector<Edge> G[]){

Edge e;

e.flow = 0; e.v = v; e.low = low; e.up = up;

G[u].push\_back(e);

**for**(size\_t i = 0; i < G[v].size(); i++) **if**(G[v][i].v == u) **return**;

e.v = u; e.low = e.up = 0;

G[v].push\_back(e);

}

**int** inc\_flow(**int** n, **int** src, **int** sink, vector<Edge> G[]){

size\_t i, j, k; Edge e, e1; **int** flow, v, inc, found;

vector<**int**> q;

flow = 0;

**while**(1){

memset(S, 0, **sizeof**(S));

S[src] = 1;

q.clear(); q.push\_back(src);

**for**(i = 0; i < q.size(); i++){

v = q[i]; **if**(v == sink) **break**;

**for**(j = 0; j < G[v].size(); j++){

e = G[v][j];

**if**(S[e.v]) **continue**;

**if**(e.flow < e.up){

q.push\_back(e.v); S[e.v] = 1; pred[e.v] = v;

maxcap[e.v] = e.up-e.flow;

} **else** {

**for**(k = 0; k < G[e.v].size(); k++){

e1 = G[e.v][k];

**if**(e1.v == v && e1.flow > e1.low){

q.push\_back(e.v); S[e.v] = 1; pred[e.v] = v;

maxcap[e.v] = e1.flow - e1.low;

**break**;

}

}

}

}

}

**if**(v != sink) **break**;

**for**(inc = INT\_MAX, v = sink; v != src; v = pred[v])

inc = min(inc, maxcap[v]);

flow += inc;

**for**(v = sink; v != src; v = pred[v]){

found = 0;

**for**(i = 0; i < G[pred[v]].size(); i++){

e = G[pred[v]][i];

**if**(e.v == v && e.flow + inc <= e.up){

found = 1;

G[pred[v]][i].flow += inc;

**break**;

}

}

**if**(!found){

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

e = G[v][i];

**if**(e.v == pred[v] && e.flow - inc >= e.low){

found = 1;

G[v][i].flow -= inc;

**break**;

}

}

}

assert(found);

}

}

**return** flow;

}

**int** maxflow\_lb(**int** n, **int** src, **int** sink){

**int** in[MAXN]={0}, out[MAXN]={0}, total = 0, flow, i;

**int** nsrc = n, nsink = n+1; size\_t j;

**for**(i = 0; i < n+2; i++) aux[i].clear();

*/\* Join each vertex to new src and sink with modified upper bounds \*/*

**for**(i = 0; i < n; i++) **for**(j = 0; j < g[i].size(); j++){

out[i] += g[i][j].low;

in[g[i][j].v] += g[i][j].low;

}

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

total += out[i];

addEdge(nsrc, i, 0, in[i], aux);

addEdge(i, nsink, 0, out[i], aux);

}

*/\* Copy old edges in with modified capacities \*/*

**for**(i = 0; i < n; i++) **for**(j = 0; j < g[i].size(); j++)

addEdge(i, g[i][j].v, 0, g[i][j].up-g[i][j].low, aux);

*/\* Add high capacity edges between old src and sink \*/*

addEdge(sink, src, 0, INT\_MAX, aux);

addEdge(src, sink, 0, INT\_MAX, aux);

*/\* Determine if a valid flow exists \*/*

**if**(inc\_flow(n+2, nsrc, nsink, aux) != total) **return** -1;

*/\* Move flow in auxillary graph into the original graph \*/*

**for**(i = 0; i < n; i++) **for**(j = 0; j < g[i].size(); j++){

Edge e = g[i][j];

**if**(e.up){

**for**(size\_t k = 0; k < aux[i].size(); k++)

**if**(aux[i][k].v == e.v && e.up-e.low == aux[i][k].up){

g[i][j].flow = aux[i][k].flow + g[i][j].low;

aux[i][k].v = -1;

**break**;

}

}

}

inc\_flow(n, src, sink, g); */\* Increment the flow \*/*

**for**(flow = j = 0; j < g[src].size(); j++) */\* Calc. max flow \*/*

flow += g[src][j].flow;

**return** flow;

}

**int** main(){

**int** n, m, src, sink, u, low, up, v, i, flow;

**while**(scanf("%d %d %d %d", &n, &m, &src, &sink) == 4){

*/\* 1. Clear graph \*/*

**for**(i = 0; i < n; i++) g[i].clear();

*/\* 2. Add edges \*/*

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

scanf("%d %d %d %d", &u, &v, &low, &up);

addEdge(u, v, low, up, g);

}

*/\* 3. Call maxflow\_lb \*/*

flow = maxflow\_lb(n, src, sink);

*/\* 4. Print off edges \*/*

**if**(flow == -1){

printf("No possible flow\n");

} **else** {

printf("Valid flow exists %d\n", flow);

**for**(i = 0; i < n; i++) **for**(size\_t j = 0; j < g[i].size(); j++){

Edge e = g[i][j];

**if**(e.up) printf("Edge %d->%d (%d,%d) Flow:%d %s\n",

i, e.v, e.low, e.up, e.flow,

S[i]&&!S[e.v] ? "CUT" : "NOT");

}

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph\_Theory:*

## Maximum Flow in a directed graph

*=================================================================*

*Description: Given a directed graph G, a source node (s), and a*

*sink node (t), computes the maximum flow from source*

*to sink, and calculates the flow along each edge.*

*Complexity: O(|V|^3)*

*-----------------------------------------------------------------*

*Author: Jason Klaus*

*Date: November 12, 2003*

*References: CLRS p690 - Relabel-to-front*

*-----------------------------------------------------------------*

*Reliability: 2 Spain 10330 - Power Transmission*

*Oct 28, 2003 - Problem G*

*(2003 Southeastern European Regional Contest)*

*Notes: - Faster then the original maximum flow code for graphs*

*with lots of edges.*

*- Multiple edges from u to v may be added. They are*

*converted into a single edge with a capacity equal to*

*their sum*

*- Vertices are assumed to be numbered from 0..n-1*

*- The graph is supplied as the number of nodes (n), the*

*zero-based indexes of the source (s) and the sink (t),*

*and a vector of edges u->v with capacity c (M).*

*\*/*

#include <cstdio>

#include <vector>

#include <list>

**using** **namespace** std;

#define MAXN 200

*//Edge u->v with capacity c*

**struct** Edge {

**int** u, v, c;

};

**int** F[MAXN][MAXN]; *//Flow of the graph*

**int** maxFlow(**int** n, **int** s, **int** t, vector<Edge> &M)

{

**int** u, v, c, oh, min, df, flow, H[n], E[n], T[n], C[n][n];

vector<Edge>::iterator m;

list<**int**> N;

list<**int**>::iterator cur;

vector<**int**> R[n];

vector<**int**>::iterator r;

**for** (u = 0; u < n; u++) {

E[u] = H[u] = T[u] = 0;

R[u].clear();

**for** (v = 0; v < n; v++)

C[u][v] = F[u][v] = 0;

}

**for** (m = M.begin(); m != M.end(); m++) {

u = m->u;

v = m->v;

c = m->c;

**if** (c && !C[u][v] && !C[v][u]) {

R[u].push\_back(v);

R[v].push\_back(u);

}

C[u][v] += c;

}

H[s] = n;

**for** (r = R[s].begin(); r != R[s].end(); r++) {

v = \*r;

F[s][v] = C[s][v];

F[v][s] = -C[s][v];

E[v] = C[s][v];

E[s] -= C[s][v];

}

N.clear();

**for** (u = 0; u < n; u++)

**if** ((u != s) && (u != t))

N.push\_back(u);

**for** (cur = N.begin(); cur != N.end(); cur++) {

u = \*cur;

oh = H[u];

**while** (E[u] > 0)

**if** (T[u] >= (**int**)R[u].size()) {

min = 10000000;

**for** (r = R[u].begin(); r != R[u].end(); r++) {

v = \*r;

**if** ((C[u][v] - F[u][v] > 0) && (H[v] < min))

min = H[v];

}

H[u] = 1 + min;

T[u] = 0;

}

**else** {

v = R[u][T[u]];

**if** ((C[u][v] - F[u][v] > 0) && (H[u] == H[v]+1)) {

df = C[u][v] - F[u][v];

**if** (df > E[u])

df = E[u];

F[u][v] += df;

F[v][u] = -F[u][v];

E[u] -= df;

E[v] += df;

}

**else**

T[u]++;

}

**if** (H[u] > oh)

N.splice(N.begin(), N, cur);

}

flow = 0;

**for** (r = R[s].begin(); r != R[s].end(); r++)

flow += F[s][\*r];

**return** flow;

}

**int** main()

{

**int** n, m, s, t, f;

Edge e;

vector<Edge> E;

**while** (scanf("%d %d %d %d", &n, &s, &t, &m) == 4) {

E.clear();

**while** (m--) {

scanf("%d %d %d", &e.u, &e.v, &e.c);

E.push\_back(e);

}

f = maxFlow(n, s, t, E);

printf("Maximum Flow: %d\n", f);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Minimum disjoint-vertex path cover on an acyclic directed graph

*=================================================================*

*Description: Given a directed ACYCLIC graph, this routine finds*

*the minimum number of directed vertex-disjoint paths*

*which cover all the vertices of the graph.*

*The paths are returned as a vector of vector of ints.*

*Nodes are assumed to be labelled [0..n-1].*

*Complexity: O(N^2)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Feb 12, 2003*

*References: "Covering the Points of a Digraph with*

*Point-Disjoint Paths and it's Application to Code*

*Generation" - F. T. Boesch & J. F. Gimpel*

*-----------------------------------------------------------------*

*Reliability: 1 (Air Raid - Problem A Feb 11, 2003)*

*Notes: The idea behind this algorithm is to split each*

*vertex v into two vertices: v\_in and v\_out. For*

*each of the original edges u->v, create a new*

*edge u\_out -> v\_in. Proceed by running maximum*

*bipartite matching on this new graph. Run through*

*each matching, following paths when they occur*

*(We need that the graph is acyclic so that such*

*a path does not result in a cycle)*

*The problem for general directed graphs is NP-C.*

*\*/*

#include <stdio.h>

#include <string.h>

#include <vector>

**using** **namespace** std;

#define VI vector<int>

#define VII vector<VI>

#define MAXN 300

**char** e[MAXN][MAXN]; */\* Adj. Matrix \*/*

**int** match[MAXN], back[MAXN], q[MAXN], tail;

**int** find(**int** x, **int** n, **int** m){

**int** i, j, r;

**if**(match[x] != -1) **return** 0;

memset(back, -1, **sizeof**(back));

**for**(q[i=0]=x, tail = 1; i < tail; i++)

**for**(j = 0; j < n+m; j++){

**if**(!e[q[i]][j]) **continue**;

**if**(match[j] != -1){

**if**(back[j] == -1){

back[j] = q[i];

back[q[tail++] = match[j]] = j;

}

} **else** {

match[match[q[i]] = j] = q[i];

**for**(r = back[q[i]]; r != -1; r = back[back[r]])

match[match[r] = back[r]] = r;

**return** 1;

}

}

**return** 0;

}

**void** bipmatch(**int** n, **int** m){

memset(match, -1, **sizeof**(match));

**for**(**int** i = 0; i < n+m; i++) **if**(find(i,n,m)) i = 0;

}

VII path\_cover(**int** n){

VII res; **char** done[MAXN] = {0};

bipmatch(n, n);

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

**if**(done[i]) **continue**;

VI p;

**for**(**int** j = i; j != -n-1; j = match[j]-n){

done[j] = 1;

p.push\_back(j);

}

res.push\_back(p);

}

**return** res;

}

**int** main(){

**int** n, m, x, y; VII path;

**while**(scanf("%d %d", &n, &m) == 2){ */\* n = number of vertices \*/*

memset(e, 0, **sizeof**(e)); */\* m = number of edges \*/*

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

scanf("%d %d", &x, &y); */\* vertices [0..n-1] \*/*

e[x][n+y] = e[n+y][x] = 1; */\* Offset y by n in bip. graph \*/*

}

path = path\_cover(n);

printf("Minimum number of vertex disjoint paths: %d\n", path.size());

**for**(size\_t i = 0; i < path.size(); i++){

printf("Path %d: ", i+1);

**for**(size\_t j = 0; j < path[i].size(); j++)

printf(" %d", path[i][j]);

printf("\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Minimum Spanning Tree (version 2)

*=================================================================*

*Description: Returns the edges / weight of the minimum spanning*

*tree, given the graph in terms of edges.*

*Vertices are numbered 0..N-1.*

*Complexity: O(M lg M) where M is the number of edges*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Dec 06, 2002*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 10034)*

*Notes:*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#define MAXN 1000

#define MAXM 1000000

#define EPS 1e-8

**int** n;

**typedef** **struct**{

**int** u, v; */\* Edge between u, v with weight w \*/*

**double** w;

} Edge;

**int** sets[MAXN];

Edge edge[MAXM], treeedge[MAXN];

**int** numedge;

**int** cmp\_Edge(Edge \*a, Edge \*b){

**if**(fabs(a->w-b->w) < EPS) **return** 0;

**if**(a->w < b->w) **return** -1;

**return** 1;

}

**int** getRoot(**int** x){

**if**(sets[x] < 0) **return** x;

**return** sets[x] = getRoot(sets[x]);

}

**void** Union(**int** a, **int** b){

**int** ra = getRoot(a);

**int** rb = getRoot(b);

**if**(ra != rb){

sets[ra] += sets[rb];

sets[rb] = ra;

}

}

**double** mintree(){

**double** weight = 0.0;

**int** i, count;

qsort(edge, numedge, **sizeof**(edge[0]), (**void** \*)cmp\_Edge);

**for**(i = count = 0; count < n-1; i++){

**if**(getRoot(edge[i].u) != getRoot(edge[i].v)){

Union(edge[i].u, edge[i].v);

weight += edge[i].w;

treeedge[count++] = edge[i];

}

}

**return** weight;

}

**int** main(){

**int** i;

**double** weight;

**while**(scanf("%d %d", &n, &numedge) == 2){

memset(sets, -1, **sizeof**(sets));

**for**(i = 0; i < numedge; i++){

scanf("%d %d %lf", &edge[i].u, &edge[i].v, &edge[i].w);

}

weight = mintree();

printf("Minimum cost tree = %f\n", weight);

printf("Tree edges: \n");

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

printf("%d %d %f\n", treeedge[i].u, treeedge[i].v, treeedge[i].w);

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Chinese Postman Problem

*=================================================================*

*Description: Given a graph, this problem returns the shortest*

*cost such that every edge is visited at least once.*

*(no path is returned)*

*Complexity: O(1\*3\*5\*7\*...\*(# of odd vertices - 1))*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 21, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Problem B: Jogging Trails 2002 Jun 01)*

*Notes: - vertices are numbered [0..n-1]*

*- multiple edges are allowed between vertices,*

*however the graph g only stored the least cost*

*(making the path hard to get back)*

*- The maximum # of vertices solvable is roughly 20*

*- How it works: Need to generate a Euler tour, so*

*try pairing off any remaining odd vertices with*

*the shortest path between them. Try all*

*perfect matchings to find the best such matching*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAXN 20

#define DISCONNECT -1

**int** g[MAXN][MAXN]; */\* Adj matrix (keep lowest cost multiedge)\*/*

**int** deg[MAXN]; */\* Degree count \*/*

**int** A[MAXN+1]; */\* Used by perfect matching generator \*/*

**int** sum; */\* Sum of costs \*/*

**int** odd;

**int** best;

**void** floyd(**int** n){

**int** i, j, k;

**for**(k = 0; k < n; k++) **for**(i = 0; i < n; i++) **for**(j = 0; j < n; j++)

**if**(g[i][k] != -1 && g[k][j] != -1){

**int** temp = g[i][k] + g[k][j];

**if**(g[i][j] == -1 || g[i][j] > temp)

g[i][j] = temp;

}

**for**(i = 0; i < n; i++) g[i][i] = 0;

}

**void** checkSum(){

**int** i, temp;

**for**(i = temp = 0; i < odd/2; i++)

temp += g[A[2\*i]][A[2\*i+1]];

**if**(best == -1 || best > temp) best = temp;

}

**void** perfmatch(**int** x){

**int** i, t;

**if**(x == 2) checkSum();

**else** {

perfmatch(x-2);

**for**(i = x-3; i >= 0; i--){

t = A[i];

A[i] = A[x-2];

A[x-2] = t;

perfmatch(x-2);

}

t = A[x-2];

**for**(i = x-2; i >= 1; i--) A[i] = A[i-1];

A[0] = t;

}

}

**int** postman(**int** n){

**int** i;

floyd(n);

**for**(odd = i = 0; i < n; i++)

**if**(deg[i]%2) A[odd++] = i;

**if**(!odd) **return** sum;

best = -1;

perfmatch(odd);

**return** sum+best;

}

**int** main(){

**int** i, u, v, c, n, m;

**while**(scanf("%d %d", &n, &m) == 2){

*/\* Clear graph and degree count \*/*

memset(g, -1, **sizeof**(g));

memset(deg, 0, **sizeof**(deg));

**for**(sum = i = 0; i < m; i++){

scanf("%d %d %d", &u, &v, &c);

u--; v--;

deg[u]++; deg[v]++;

**if**(g[u][v] == -1 || g[u][v] > c) g[u][v] = c;

**if**(g[v][u] == -1 || g[v][u] > c) g[v][u] = c;

sum += c;

}

printf("Best cost: %d\n", postman(n));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## K-cycle in a directed tournament graph

*=================================================================*

*Description: Given a tournament graph, and a value k, finds a*

*k-cycle if one exists*

*Complexity: O(N^2 ??) Approximate*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 20, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Problem B: Tour from 2003 Jan 21)*

*Notes: -A strongly connected tournament of size n is*

*pancyclic - it contains all cycles from 3 to n*

*in it.*

*\*/*

#include <stdio.h>

#include <vector>

**using** **namespace** std;

#define VI vector<int>

#define MAXN 1000

VI g[MAXN], curr;

vector< VI > scc;

VI kcycle;

**int** dfsnum[MAXN], low[MAXN], id;

**char** done[MAXN], adj[MAXN][MAXN];

**void** visit(**int** x){

curr.push\_back(x);

dfsnum[x] = low[x] = id++;

**for**(size\_t i = 0; i < g[x].size(); i++)

**if**(dfsnum[g[x][i]] == -1){

visit(g[x][i]);

low[x] <?= low[g[x][i]];

} **else** **if**(!done[g[x][i]])

low[x] <?= dfsnum[g[x][i]];

**if**(low[x] == dfsnum[x]){

VI c; **int** y;

**do**{

done[y = curr[curr.size()-1]] = 1;

c.push\_back(y);

curr.pop\_back();

} **while**(y != x);

scc.push\_back(c);

}

}

**void** strong\_conn(**int** n){

memset(dfsnum, -1, n\***sizeof**(**int**));

memset(done, 0, **sizeof**(done));

scc.clear(); curr.clear();

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

**if**(dfsnum[i] == -1) visit(i);

}

**int** FindCycle(VI v, **int** csize){

VI cycle; **int** i, j, k, n = v.size();

memset(low, -1, **sizeof**(low));

memset(done, 0, **sizeof**(done));

**for**(i = 0; i < n; i++) low[v[i]] = i;

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

**for**(size\_t j = 0; j < g[v[i]].size(); j++)

**if**(low[g[v[i]][j]] != -1)

adj[i][low[g[v[i]][j]]] = 1;

**for**(i = 0; i < n; i++) **for**(j = 0; j < n; j++){

**if**(!adj[i][j]) **continue**;

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

**if**(adj[j][k] && adj[k][i]){

cycle.push\_back(i); done[i] = 1;

cycle.push\_back(j); done[j] = 1;

cycle.push\_back(k); done[k] = 1;

**goto** found;

}

}

found:

**while**((**int**)cycle.size() < csize){

vector<**int**> in, out;

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

**if**(done[i]) **continue**;

**if**(adj[cycle[cycle.size()-1]][i] && adj[i][cycle[0]]){

cycle.push\_back(i);

done[i] = 1;

**goto** next;

}

**for**(size\_t j = cycle.size()-2; j >= 0; j--){

**if**(adj[cycle[j]][i] && adj[i][cycle[(j+1)%cycle.size()]]){

cycle.insert(&cycle[j+1], i);

done[i] = 1;

**goto** next;

}

}

**if**(adj[cycle[0]][i]) out.push\_back(i);

**else** in.push\_back(i);

}

**for**(size\_t i = 0; i < out.size(); i++){

**if**(done[i]) **continue**;

**for**(size\_t j = 0; j < in.size(); j++){

**if**(done[j]) **continue**;

**if**(adj[i][j]){

done[cycle[cycle.size()-1]] = 0;

done[i] = done[j] = 1;

cycle[cycle.size()-1] = i;

cycle.push\_back(j);

**goto** next;

}

}

}

next:

;

}

kcycle.clear();

**for**(size\_t i = 0; i < cycle.size(); i++)

kcycle.push\_back(v[cycle[i]]);

}

**int** getKCycle(**int** n, **int** k){

strong\_conn(n);

**for**(size\_t i = 0; i < scc.size(); i++){

**if**((**int**)scc[i].size() >= k){

FindCycle(scc[i], k);

**return** 1;

}

}

**return** 0;

}

**int** main(){

**int** n, i, x, y, k;

*/\* Read in number of vertices, size of cycle \*/*

**while**(scanf("%d %d", &n, &k) == 2){

*/\* Clear adj. lists \*/*

**for**(i = 0; i < n; i++) g[i].clear();

*/\* Build adj. lists \*/*

**for**(i = 0; i < n\*(n-1)/2; i++){

scanf("%d %d", &x, &y);

g[x].push\_back(y);

}

**if**(getKCycle(n, k)){

printf("%d-cycle exists:\n", k);

**for**(i = 0; i < k; i++)

printf("%d%c", kcycle[i], i == k-1 ? '\n' : ' ');

} **else** {

printf("No %d-cycle exists\n", k);

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## Stable Marriage

*=================================================================*

*Description: Given n women and n men, each with a list of*

*preferences as to whom they want to marry, match*

*each woman to a man in a stable way. In a stable*

*matching, there is no pair who would both rather be*

*with each other than with their current partners.*

*Complexity: O(N^3) worst case, O(N^2 lg N) average case*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 03, 2003*

*References: www.cs.dal.ca/apics/contest99/c.txt*

*-----------------------------------------------------------------*

*Reliability: 1 (Atlantic Canada Preliminaries, 1999)*

*Notes: Women order men, and men order women, according to*

*preference lists. Say we have 5 men, a, b, c, d, e,*

*and 5 women, A, B, C, D, E. Then A's preference list*

*might look like this: 3 1 0 2 4, which means that A*

*likes d the most and e the least. Your application*

*might have rank lists instead. A's ranklist is*

*2, 1, 3, 0, 4, which means that a is her #2 choice*

*(counting from 0) and b is her #1 choice. Convert*

*between the types as shown in the example.*

*This implementation doesn't handle ties or*

*incomplete preference lists.*

*\*/*

#include <stdio.h>

#include <string.h>

#define NMAX 100

**int** vpref[NMAX][NMAX]; */\* women's choice of men, in descending order \*/*

**int** upref[NMAX][NMAX]; */\* men's choice of men \*/*

**int** vrank[NMAX][NMAX]; */\* women rank men, i.e. each man's "score" \*/*

**int** urank[NMAX][NMAX]; */\* men rank women \*/*

**int** suitors[NMAX][NMAX]; */\* woman A's partner is suitors[A][0] \*/*

*/\* example: to get vpref from vrank, convert(vpref,vrank,n) \*/*

*/\* example: to get vrank from vpref, convert(vrank,vpref,n) \*/*

**void** convert(**int** a[NMAX][NMAX], **int** b[NMAX][NMAX], **int** n) {

**int** i, j;

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

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

a[i][b[i][j]] = j;

}

**void** stable(**int** n) {

**int** choice[NMAX], nsuit[NMAX], used[NMAX];

**int** i, j, best, nused;

memset(used,0,**sizeof**(used));

memset(nsuit,0,**sizeof**(nsuit));

memset(choice,0,**sizeof**(choice));

convert(upref,urank,n);

**for** (nused = 0; nused < n; ) {

**for** (i = 0; i < n; i++) { */\* men propose \*/*

**if** (used[i]) **continue**;

j = upref[i][choice[i]++];

suitors[j][nsuit[j]++] = i;

nused++;

}

**for** (i = 0; i < n; i++) { */\* women choose \*/*

**if** (!nsuit[i]) **continue**;

best = suitors[i][0];

**for** (j = 0; j < nsuit[i]; j++) {

**if** (vrank[i][suitors[i][j]] < vrank[i][best])

best = suitors[i][j];

used[suitors[i][j]] = 0;

}

suitors[i][0] = best;

used[best] = 1;

nused -= nsuit[i] - 1;

nsuit[i] = 1;

}

}

}

**int** main() {

**int** i, j, n;

scanf("%d",&n);

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

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

scanf("%d",&vpref[i][j]);

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

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

scanf("%d",&upref[i][j]);

convert(vrank,vpref,n);

convert(urank,upref,n);

stable(n);

**for** (i = 0; i < n; i++) printf("%d %d\n",i+1,suitors[i][0]+1);

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Strongly Connected Components

*=================================================================*

*Description: Given a directed graph, decomposes the graph into*

*its strongly connected components.*

*The components are returned in the vector of*

*vector of ints scc.*

*Complexity: O(V+E)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Feb 20, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Nodes are assumed to be labelled 0..n-1*

*- This routine works by finding the earliest node*

*reachable in the DFS tree. If the earliest such*

*node is itself, then everything after it in the*

*DFS tree is a strongly connected component.*

*\*/*

#include <stdio.h>

#include <vector>

**using** **namespace** std;

#define VI vector<int>

#define MAXN 1000

VI g[MAXN], curr;

vector< VI > scc;

**int** dfsnum[MAXN], low[MAXN], id;

**char** done[MAXN];

**void** visit(**int** x){

curr.push\_back(x);

dfsnum[x] = low[x] = id++;

**for**(size\_t i = 0; i < g[x].size(); i++)

**if**(dfsnum[g[x][i]] == -1){

visit(g[x][i]);

low[x] <?= low[g[x][i]];

} **else** **if**(!done[g[x][i]])

low[x] <?= dfsnum[g[x][i]];

**if**(low[x] == dfsnum[x]){

VI c; **int** y;

**do**{

done[y = curr[curr.size()-1]] = 1;

c.push\_back(y);

curr.pop\_back();

} **while**(y != x);

scc.push\_back(c);

}

}

**void** strong\_conn(**int** n){

memset(dfsnum, -1, n\***sizeof**(**int**));

memset(done, 0, **sizeof**(done));

scc.clear(); curr.clear();

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

**if**(dfsnum[i] == -1) visit(i);

}

**int** main(){

**int** n, m, i, x, y;

**while**(scanf("%d %d", &n, &m) == 2){

**for**(i = 0; i < n; i++) g[i].clear();

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

scanf("%d %d", &x, &y);

g[x].push\_back(y);

}

strong\_conn(n);

**for**(size\_t i = 0; i < scc.size(); i++){

printf("Component %d:", i+1);

**for**(size\_t j = 0; j < scc[i].size(); j++)

printf(" %d", scc[i][j]);

printf("\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Topological Sort

*=================================================================*

*Description: Given a directed acyclic graph, fills an array*

*called top[] with the nodes in topological order.*

*Complexity: O(N) where N is the number of nodes*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Feb 20, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: A topological sort of a DAG is a linear ordering of*

*it's vertices such that for any edge u->v in the*

*the graph, u comes before v in the ordering.*

*There may be more than one way to topological sort*

*a DAG. If the lexicographically first one is needed,*

*it may be a good idea to use a priority queue instead*

*and just pull off vertices with 0 in-degree*

*Vertices are numbered 0..N-1.*

*\*/*

#include <stdio.h>

#include <string.h>

#include <vector>

**using** **namespace** std;

#define MAXN 1000

vector<**int**> g[MAXN];

**int** id, top[MAXN], seen[MAXN];

**void** DFS(**int** x){

seen[x] = 1;

**for**(size\_t i = 0; i < g[x].size(); i++)

**if**(!seen[g[x][i]]) DFS(g[x][i]);

top[id--] = x;

}

**void** top\_sort(**int** n){

memset(seen, 0, **sizeof**(seen));

id = n-1;

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

**if**(!seen[i]) DFS(i);

}

**int** main(){

**int** n, m, i, x, y;

*/\* n = # of vertices, m = # of edges \*/*

**while**(scanf("%d %d", &n, &m) == 2){

**for**(i = 0; i < n; i++) g[i].clear();

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

scanf("%d %d", &x, &y);

g[x].push\_back(y);

}

top\_sort(n);

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

printf("%d ", top[i]);

printf("\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Tree Isomorphism by certificates

*=================================================================*

*Description: Given a tree this algorithm returns a certificate of*

*the tree. Two trees are isomorphic if and only if*

*the certificates of the two trees are the same.*

*The certificate computed is a binary string of*

*length 2n.*

*Complexity: O(N) (maybe a little more)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Feb 12, 2003*

*References: Combinatorial Algorithms: D Kreher*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - A tree is given by the number of vertices (t.n) and*

*and adjacency list.*

*- Vertices are numbered 0..n-1*

*- Note that with strings, equality can be checked by*

*"s1 == s2".*

*- Verification: 5-Line (0-1-2-3-4) = 0001100111*

*5-Star (0-[1-4]) = 0010101011*

*\*/*

#include <stdio.h>

#include <string>

#include <list>

#include <set>

**using** **namespace** std;

#define MAXN 1000

**typedef** **struct**{

**int** n; list<**int**> adj[MAXN];

} Tree;

string getTreeID(Tree t){

multiset<string> s[MAXN]; multiset<string>::iterator it;

set<**int**> leaf; set<**int**>::iterator j;

**char** parent[MAXN]; string id[MAXN], res[2];

**int** left = t.n, i, k, x;

**for**(i = 0; i < t.n; i++) id[i] = "01";

**while**(left > 2){

memset(parent, 0, **sizeof**(parent));

**for**(i = 0; i < t.n; i++) s[i].clear();

leaf.clear();

**for**(i = 0; i < t.n; i++)

**if**(t.adj[i].size() == 1){

leaf.insert(i);

x = \*t.adj[i].begin();

s[x].insert(id[i]);

parent[x] = 1;

left--;

}

**for**(i = 0; i < t.n; i++)

**if**(parent[i]){

x = id[i].size();

**if**(x > 2) s[i].insert(id[i].substr(1,x-2));

**for**(id[i] = "0", it = s[i].begin(); it != s[i].end(); ++it)

id[i] += \*it;

id[i] += '1';

}

**for**(i = 0; i < t.n; i++)

**if**(leaf.count(i) == 1) t.adj[i].clear();

**else**

**for**(j = leaf.begin(); j != leaf.end(); ++j)

**for**(k = 0; k < t.n; k++)

t.adj[k].remove(\*j);

}

**for**(i = x = 0; i < t.n; i++)

**if**(parent[i]) res[x++] = id[i];

**if**(left == 1) **return** res[0];

**return** (res[0] < res[1]) ? res[0]+res[1] : res[1]+res[0];

}

**int** main(){

**int** i, x, y;

Tree t;

string id;

**while**(scanf("%d", &t.n) == 1 && t.n){ */\* Get number of nodes \*/*

**for**(i = 0; i < t.n; i++)

t.adj[i].clear(); */\* Clear adj. list \*/*

**for**(i = 0; i < t.n-1; i++){ */\* Get edges of tree \*/*

scanf("%d %d", &x, &y);

t.adj[x].push\_back(y);

t.adj[y].push\_back(x);

}

id = getTreeID(t);

printf("This tree's ID is [%s]\n", id.c\_str());

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Graph Theory:*

## Maximum Weighted Bipartite Matching Combinatorics: Assignment Problem

*=================================================================*

*Description: Given N workers and N jobs to complete, where each worker has a*

*certain compatibility (weight) to each job, find an assignment*

*(perfect matching) of workers to jobs which maximizes the*

*compatibility (weight).*

*Complexity: O(n^3), where n is the number of workers or jobs.*

*-----------------------------------------------------------------*

*Author: Jason Klaus*

*Date: February 18, 2004*

*References: www.cs.umd.edu/class/fall2003/cmsc651/lec07.ps*

*-----------------------------------------------------------------*

*Reliability: 3*

*Notes: - W is a 2 dimensional array where W[i][j] is the weight of*

*worker i doing job j. Weights must be non-negative. If*

*there is no weight assigned to a particular worker and job*

*pair, set it to zero. If there is a different number of*

*workers than jobs, create dummy workers or jobs accordingly*

*with zero weight edges.*

*- M is a 1 dimensional array populated by the algorithm where*

*M[i] is the index of the job matched to worker i.*

*- This algorithm could be used on non-negative floating point*

*weights as well.*

*\*/*

#include <stdio.h>

*/\* Maximum number of workers/jobs \*/*

#define MAX\_N 100

**int** W[MAX\_N][MAX\_N], U[MAX\_N], V[MAX\_N], Y[MAX\_N]; */\* <-- weight variables \*/*

**int** M[MAX\_N], N[MAX\_N], P[MAX\_N], Q[MAX\_N], R[MAX\_N], S[MAX\_N], T[MAX\_N];

*/\* Returns the maximum weight, with the perfect matching stored in M. \*/*

**int** Assign(**int** n)

{

**int** w, y; */\* <-- weight variables \*/*

**int** i, j, m, p, q, s, t, v;

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

M[i] = N[i] = -1;

U[i] = V[i] = 0;

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

**if** (W[i][j] > U[i])

U[i] = W[i][j];

}

**for** (m = 0; m < n; m++) {

**for** (p = i = 0; i < n; i++) {

T[i] = 0;

Y[i] = -1;

**if** (M[i] == -1) {

S[i] = 1;

P[p++] = i;

}

**else**

S[i] = 0;

}

**while** (1) {

**for** (q = s = 0; s < p; s++) {

i = P[s];

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

**if** (!T[j]) {

y = U[i] + V[j] - W[i][j];

**if** (y == 0) {

R[j] = i;

**if** (N[j] == -1)

**goto** end\_phase;

T[j] = 1;

Q[q++] = j;

}

**else** **if** ((Y[j] == -1) || (y < Y[j])) {

Y[j] = y;

R[j] = i;

}

}

}

**if** (q == 0) {

y = -1;

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

**if** (!T[j] && ((y == -1) || (Y[j] < y)))

y = Y[j];

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

**if** (T[j])

V[j] += y;

**if** (S[j])

U[j] -= y;

}

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

**if** (!T[j]) {

Y[j] -= y;

**if** (Y[j] == 0) {

**if** (N[j] == -1)

**goto** end\_phase;

T[j] = 1;

Q[q++] = j;

}

}

}

**for** (p = t = 0; t < q; t++) {

i = N[Q[t]];

S[i] = 1;

P[p++] = i;

}

}

end\_phase:

i = R[j];

v = M[i];

M[i] = j;

N[j] = i;

**while** (v != -1) {

j = v;

i = R[j];

v = M[i];

M[i] = j;

N[j] = i;

}

}

**for** (i = w = 0; i < n; i++)

w += W[i][M[i]];

**return** w;

}

**int** main()

{

**int** w; */\* <-- weight variables \*/*

**int** n, i, j;

**while** ((scanf("%d", &n) == 1) && (n != 0)) {

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

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

scanf("%d", &W[i][j]);

}

}

w = Assign(n);

printf("Optimum weight: %d\n", w);

printf("Matchings:\n");

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

printf("%d matched to %d\n", i, M[i]);

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Miscellaneous

############################################

*/\* Miscellaneous:*

## Bit Count

*=================================================================*

*Description: Takes an integer value and returns the number of*

*bits that are set to one.*

*Complexity: O(N) where N is the number of '1' bits.*

*-----------------------------------------------------------------*

*Author: Patrick Earl*

*Date: Sept 25, 2002*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: As is, it accepts large unsigned values. It can be*

*modified to check larger integers by using long long*

*instead of int.*

*\*/*

#include <stdio.h>

**int** bitcount(**int** a){

**int** c = 0;

**while**(a){

c++;

a &= a-1;

}

**return** c;

}

**int** main(){

**int** a;

**while**(scanf("%d", &a) == 1){

printf("Number of 1 bits in %d is %d\n", a, bitcount(a));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## Roman Numerals

*=================================================================*

*Description: Converts an arabic number (as an int) to roman*

*numerals (as a null-terminated string) and vice versa.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 07, 2003*

*References: http://www.wilkiecollins.demon.co.uk/roman/front.htm*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Remember that the romans didn't use negative*

*numbers or the number zero, so those aren't valid*

*numbers. Numbers greater than or equal to 5000*

*aren't valid either; those roman numbers can't*

*easily be represented in ASCII.*

*The romans didn't have well-defined numbers; this*

*implementation uses the rules you see for dating*

*television shows, so things like IX are legal,*

*but IM is not.*

*\*/*

#include <stdio.h>

#include <string.h>

#include <string>

#include <map>

**using** **namespace** std;

map<string, **int**, less <string> > dict;

**char** nums[5000][20];

**void** gen\_roman() {

**char** \*roman[13] =

{"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};

**int** i, j, n, arab[13] = {1000,900,500,400,100,90,50,40,10,9,5,4,1};

string key;

**for** (i = 0; i < 5000; i++) {

nums[i][0] = 0;

**for** (n = i, j = 0; n; j++)

**for** (; n >= arab[j]; n -= arab[j])

strcat(nums[i],roman[j]);

key = nums[i];

dict[key] = i;

}

}

**char** \*to\_roman(**int** n) {

**if** (n < 1 || n >= 5000) **return** 0;

**return** nums[n];

}

**int** to\_arabic(**char** \*in) {

string key = in;

**if** (!dict.count(key)) **return** -1;

**return** dict[key];

}

**int** main() {

**int** i;

gen\_roman();

**for** (i = 1; i < 5000; i++)

printf("%d = %s\n",to\_arabic(to\_roman(i)),to\_roman(i));

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## Rubik's Cube simulator

*=================================================================*

*Description: Given a flattened Rubik's cube, provides utilities*

*for clockwise rotation of each face*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Jan 27, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 Problem A: Grandapa's Rubik Cube*

*(ACM South America 2002) Jan 26, 2003*

*Notes: Each cube must be represented by a 9 by 12 character*

*array, with the faces numbered as follows.*

*0123456789AB Rotate(Cube c, int f) rotates face f*

*0...+-+...... 90 degrees clockwise.*

*1...|4|......*

*2...+-+...... +-----+ +-----+*

*3+-++-++-++-+ | ^ | rotate | |*

*4|0||1||2||3| | | | -----> | \*-->|*

*5+-++-++-++-+ | \* | cw 90 | |*

*6...+-+...... +-----+ +-----+*

*7...|5|...... Rotating 90 degrees counter-clockwise*

*8...+-+...... may be done by calling Rotate 3 times*

*isSolved(Cube c) returns 1 if each face contains the*

*same characters, and 0 otherwise*

*printCube(Cube c) is provided to see cube c.*

*\*/*

#include <stdio.h>

#include <string.h>

**typedef** **struct**{

**char** g[9][12];

} Cube;

**int** cx[6] = {4,4,4,4,1,7};

**int** cy[6] = {1,4,7,10,4,4};

**int** rx[6][4][3] = {{{0,1,2}, {5,4,3}, {6,7,8}, {3,4,5}},

{{2,2,2}, {5,4,3}, {6,6,6}, {3,4,5}},

{{2,1,0}, {5,4,3}, {8,7,6}, {3,4,5}},

{{5,4,3}, {8,8,8}, {3,4,5}, {0,0,0}},

{{3,3,3}, {3,3,3}, {3,3,3}, {3,3,3}},

{{5,5,5}, {5,5,5}, {5,5,5}, {5,5,5}}};

**int** ry[6][4][3] = {{{3,3,3}, {11,11,11}, {3,3,3}, {3,3,3}},

{{3,4,5}, {2,2,2}, {5,4,3}, {6,6,6}},

{{5,5,5}, {5,5,5}, {5,5,5}, {9,9,9}},

{{8,8,8}, {3,4,5}, {0,0,0}, {5,4,3}},

{{0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}},

{{11,10,9}, {8,7,6}, {5,4,3}, {2,1,0}}};

**int** isSolved(Cube c){

**int** i, x, y;

**for**(i = 0; i < 6; i++)

**for**(x = cx[i]-1; x <= cx[i]+1; x++)

**for**(y = cy[i]-1; y <= cy[i]+1; y++)

**if**(c.g[x][y] != c.g[cx[i]][cy[i]]) **return** 0;

**return** 1;

}

Cube Rotate(Cube c, **int** f){

**char** t[3][3]; **int** i, j, x = cx[f], y = cy[f];

**for**(i = 0; i < 3; i++) **for**(j = 0; j < 3; j++)

t[i][j] = c.g[x+i-1][y+j-1];

**for**(i = 0; i < 3; i++) **for**(j = 0; j < 3; j++)

c.g[x+i-1][y+j-1] = t[2-j][i];

**for**(i = 0; i < 3; i++)

t[0][i] = c.g[rx[f][0][i]][ry[f][0][i]];

**for**(j = 0; j < 3; j++) **for**(i = 0; i < 3; i++)

c.g[rx[f][j ][i]][ry[f][j ][i]] =

c.g[rx[f][j+1][i]][ry[f][j+1][i]];

**for**(i = 0; i < 3; i++)

c.g[rx[f][j][i]][ry[f][j][i]] = t[0][i];

**return** c;

}

**void** printCube(Cube c){

**int** i, j;

**for**(i = 0; i < 9; i++){

**for**(j = 0; j < 12; j++)

printf("%c", c.g[i][j] ? c.g[i][j] : ' ');

printf("\n");

}

}

**int** main(){

**int** i, j, d; Cube c;

memset(c.g, 0, **sizeof**(c.g));

**for**(i = 0; i < 3; i++) **for**(j = 3; j < 6; j++)

scanf(" %c", &c.g[i][j]);

**for**(i = 3; i < 6; i++) **for**(j = 0; j < 12; j++)

scanf(" %c", &c.g[i][j]);

**for**(i = 6; i < 9; i++) **for**(j = 3; j < 6; j++)

scanf(" %c", &c.g[i][j]);

printCube(c);

**while**(scanf(" %d", &d) == 1){

c = Rotate(c, d);

printf("\n");

printCube(c);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## Square index rotation/flipping

*=================================================================*

*Description: For a square of size n with indices 0..n-1,*

*Given x and y and a orientation*

*returns the corresponding x,y coordinate.*

*Complexity: O(1)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Jan 27, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 Problem G: Noise Effect*

*(ACM South America - 2002) Jan 26, 2003*

*Notes: Here are diagrams of the various orientations:*

*x y*

*^ ^ <--+ +-->*

*y| |x |y x|*

*+--> <--+ v v*

*x y*

*d=0 d=1 d=2 d=3*

*y x*

*^ <--+ +--> ^*

*|y |x y| x|*

*<--+ v v +-->*

*x y*

*d=4 d=5 d=6 d=7*

*\*/*

#include <stdio.h>

**int** n, g[10][10];

**int** dx(**int** x, **int** y, **int** d){

**switch**(d){

**case** 0: **case** 6: **return** x;

**case** 1: **case** 5: **return** n-1-y;

**case** 2: **case** 4: **return** n-1-x;

**case** 3: **case** 7: **return** y;

}

**return** -1;

}

**int** dy(**int** x, **int** y, **int** d){

**switch**(d){

**case** 0: **case** 4: **return** y;

**case** 1: **case** 7: **return** x;

**case** 2: **case** 6: **return** n-1-y;

**case** 3: **case** 5: **return** n-1-x;

}

**return** -1;

}

**int** main(){

**int** i, j, d;

scanf("%d", &n);

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

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

g[i][j] = i\*n+j;

**for**(d = 0; d < 8; d++){

printf("Orientation [%d]\n", d);

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

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

printf("%d", g[dx(i,j,d)][dy(i,j,d)]);

printf("\n");

}

printf("\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## String Reverse

*=================================================================*

*Description: Given a string of characters, reverse it.*

*Complexity: O(N)*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 14, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 483)*

*Notes: This is extremely simple but sometimes you forget.*

*\*/*

#include <stdio.h>

#include <string.h>

**char** \*strrev(**char** \*a) {

**int** i, n = strlen(a);

**char** tmp;

**for** (i = 0; i < n/2; i++) {

tmp = a[i];

a[i] = a[n-i-1];

a[n-i-1] = tmp;

}

**return** a;

}

**int** main() {

**char** word[1000];

**while** (scanf("%s",word)==1)

printf("%s\n",strrev(word));

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Misc:*

## Uniquely Decipherable Codes

*=================================================================*

*Description: A set of code words is said to be uniquely*

*decipherable if every possible message can be broken*

*down into the code words in only one unique way.*

*E.g. {0,01} is UD, where {0,1,01} is not since the*

*message 01 can be either [01] or [0][1]*

*UDFind(codewords) returns an empty string is the*

*codewords are UD, otherwise the smallest*

*lexicographic message that is not UD.*

*Complexity: O(N^2 \* L^2) where N is # of codewords, L is maximum*

*length of a code word*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 13, 2003*

*References: Graph Algorithms - Shimon Even - pg. 69-70*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - Note that prefix-free codes (e.g. Huffman) are UD*

*but some non-prefix-free codes are UD ({0,01})*

*- Use ud\_test.cc if you only want to test if a set*

*of codewords is UD.*

*- The routine handles all characters (including*

*spaces if necssary): just change input format*

*\*/*

#include <stdio.h>

#include <string>

#include <vector>

#include <set>

#include <queue>

#include <algorithm>

**using** **namespace** std;

#define MAXL 200 */\* Maximum length of a codeword \*/*

#define VS vector<string>

#define SS(x) (x.size())

#define TAIL(x,y) y.substr(SS(x),SS(y)-SS(x))

#define ADD(x,y) s.insert(TAIL(x,y));c.a=x;c.b=TAIL(x,y);tails.push(c)

#define ADD2(x,y) c.a=top.a+x;c.b=TAIL(x,y);tails.push(c)

#define PRE(x,y) (SS(x) < SS(y) && x==y.substr(0,SS(x)))

#define CHECK(x,y) if(PRE(x,y) && !s.count(TAIL(x,y))){ADD(x,y);}

#define CHECK2(x,y) if(PRE(x,y) && !s.count(TAIL(x,y))){ADD2(x,y);}

**struct** Code{

string a,b;

**bool** **operator**<(**const** Code &o) **const**{

string x = a+b; string y = o.a+o.b;

**return** (SS(x)>SS(y)) || (SS(x)==SS(y) && x>y);

}

};

**bool** cmpSS(**const** string &a, **const** string &b){

**return** SS(a) < SS(b) || (SS(a) == SS(b) && a < b);

}

string UDFind(VS code){

**int** i, j, n = code.size(); string t; Code c;

set<string> s; */\* Mark tails already seen \*/*

priority\_queue<Code> tails; */\* Keep track of current tails \*/*

sort(code.begin(), code.end(), cmpSS);

s.clear();

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

**if**(code[i] == code[j]) **return** code[i];

CHECK(code[i], code[j]);

CHECK(code[j], code[i]);

}

s.clear();

**while**(!tails.empty()){

Code top = tails.top(); tails.pop();

**if**(s.count(top.b)) **continue**;

s.insert(top.b);

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

**if**(top.b == code[j]) **return** top.a+top.b;

CHECK2(top.b, code[j]);

CHECK2(code[j], top.b);

}

}

**return** "";

}

**int** main(){

**int** i, n; **char** buff[MAXL]; VS code;

**while**(scanf("%d", &n) == 1){

code.clear();

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

scanf(" %s", buff);

code.push\_back(buff);

}

printf("[%s]\n", UDFind(code).c\_str());

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Misc:*

## Uniquely Decipherable Codes Test

*=================================================================*

*Description: A set of code words is said to be uniquely*

*decipherable if every possible message can be broken*

*down into the code words in only one unique way.*

*E.g. {0,01} is UD, where {0,1,01} is not since the*

*message 01 can be either [01] or [0][1]*

*isUDTest(codewords) returns 1 if the codewords are*

*UD and 0 otherwise.*

*Complexity: O(N^2 \* L^2) where N is # of codewords, L is maximum*

*length of a code word*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 13, 2003*

*References: Graph Algorithms - Shimon Even - pg. 69-70*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - Note that prefix-free codes (e.g. Huffman) are UD*

*but some non-prefix-free codes are UD ({0,01})*

*- Use ud\_find.cc if you want to find a minimum length*

*string which is not uniquely decipherable.*

*- The routine handles all characters (including*

*spaces if necssary): just change input format*

*\*/*

#include <stdio.h>

#include <string>

#include <vector>

#include <set>

**using** **namespace** std;

#define MAXL 200 */\* Max codeword length \*/*

#define VS vector<string>

#define SS(x) (x.size())

#define TAIL(x,y) y.substr(SS(x),SS(y)-SS(x))

#define ADD(x,y) s.insert(TAIL(x,y));tails.push\_back(TAIL(x,y))

#define PRE(x,y) (SS(x) < SS(y) && x==y.substr(0,SS(x)))

#define CHECK(x,y) if(PRE(x,y) && !s.count(TAIL(x,y))){ADD(x,y);}

**int** isUDTest(VS code){

**int** i, j, n = code.size(); VS tails;

set<string> s; string t;

tails.clear(); s.clear();

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

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

**if**(code[i] == code[j]) **return** 0;

CHECK(code[i], code[j]);

CHECK(code[j], code[i]);

}

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

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

**if**(tails[i] == code[j]) **return** 0;

CHECK(tails[i], code[j]);

CHECK(code[j], tails[i]);

}

**return** 1;

}

**int** main(){

**int** i, n; **char** buff[MAXL]; VS code;

**while**(scanf("%d", &n) == 1){

code.clear();

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

scanf(" %s", buff);

code.push\_back(buff);

}

printf("%s\n", isUDTest(code) ? "YES" : "NO");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Number Theory

############################################

*/\* Number Theory:*

## GCD of binary numbers (Algorithm B)

*=================================================================*

*Description: Computes the greatest common divisor of two*

*integers expressed in binary.*

*Complexity: O((lg2(uv))^2)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Dec 29, 2002*

*References: http://www.nist.gov/dads/HTML/binaryGCD.html*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: When efficiency isn't a big concern, the standard*

*gcd algorithm is probably a better choice.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

**int** gcd\_b(**int** u, **int** v){

**int** g = 1, t;

**while**(!(u & 1 || v & 1)){

u >>= 1;

v >>= 1;

g <<= 1;

}

**while**(u){

**if**(!(u&1)) u >>= 1;

**else** **if**(!(v&1)) v >>= 1;

**else** {

t = abs(u-v)>>1;

**if**(u < v) v = t;

**else** u = t;

}

}

**return** g\*v;

}

**int** main(){

**int** a, b;

**while**(scanf("%d %d", &a, &b) == 2){

printf("GCD %d %d = %d\n", a, b, gcd\_b(a,b));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\**

## Farey Sequence Generator

*Author: Zachary Friggstad*

*Date: Aug, 2008*

*---------------------------------------------------------------*

*The Farey Sequence of order n is the list of*

*all reduced fractions between 0 and 1 (inclusive)*

*in sorted order.*

*e.g. order 6:*

*0/1, 1/6, 1/5, 1/4, 2/5, 1/3, 1/2, 2/3, 3/5, 3/4, 4/5, 5/6, 1/1*

*Given any positive integer n, this algorithm will generate*

*the Farey sequence in order with one term being generated per*

*loop iteration.*

*---------------------------------------------------------------*

*Complexity:*

*- linear time in the size of the output*

*- constant space*

*Reference:*

*"Introduction to the Theory of Numbers" - Hardy & Wright*

*Reliablity:*

*UVa - 10408*

*\*/*

#include <iostream>

#include <algorithm>

**using** **namespace** std;

**void** farey(**int** n) {

**int** h = 0, k = 1, x = 1, y = 0;

**do** {

cout << h << '/' << k << endl;

**int** r = (n-y)/k;

y += r\*k;

x += r\*h;

swap(x,h);

swap(y,k);

x = -x;

y = -y;

} **while** (k > 1);

cout << "1/1" << endl;

}

**int** main() {

**int** n;

**while** (cin >> n)

farey(n);

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Parsing

############################################

*/\* Parsing:*

## Evaluating Arithmetic Expressions

*=================================================================*

*Description: Given a infix expression in a string, the parser*

*evaluates the string and reports errors if the*

*parsing fails or the evaluation causes a division by*

*zero.*

*Complexity: O(N) where N is the size of the string*

*-----------------------------------------------------------------*

*Author: Patrick Earl*

*Date: Nov 14, 2002*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: Leading zeros are handled.*

*By default, the code uses long long, but it cannot handle*

*the minimum long long value, only that value plus one.*

*Overflow in the math operations is not checked.*

*Whitespace is ignored.*

*Expressions follow this grammar:*

*Expr = Term (('+'|'-') Term) \**

*Term = Factor (('\*'|'/') Factor) \**

*Factor = Unit | ('+'|'-')? Factor*

*Unit = LongLongInteger | '('Expr')'*

*\*/*

#include <stdio.h>

#include <ctype.h>

#include <limits.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

**enum** {

SUCCESS,

DIV\_BY\_ZERO, */\* A division by zero would have occured. \*/*

PARSE\_UNEXPECTED,*/\* Unexpected character. Also reported for empty*

*string. \*/*

PARSE\_TOOLONG, */\* Extra non-whitespace junk after complete*

*expression has been parsed. \*/*

PARSE\_OVERFLOW, */\* Integer is too long to fit in an int. \*/*

PARSE\_NOPAREN */\* No matching parentheses. \*/*

};

*/\* If you change this, you need to change the low level parsing as well. \*/*

**typedef** **long** **long** TYPE;

**char** \*s;

**int** pos;

**int** error;

**char** gc() {

**while**(isspace(s[pos])) pos++;

**return** s[pos++];

}

**void** ugc() {

pos--;

}

TYPE Expr();

TYPE Unit() {

**char** c;

TYPE expr;

**char** num[100];

**char** \*ref;

**int** p, zero;

c = gc();

**if**(c == '(') {

expr = Expr();

**if**(error) **return** 0;

**if**(gc() != ')') {

error = PARSE\_NOPAREN;

**return** 0;

}

**return** expr;

} **else** **if**(isdigit(c)) {

p=0;

zero=1;

**while**(isdigit(c)) {

**if**(zero && c=='0') {

c=gc();

**continue**;

}

zero=0;

num[p++]=c;

**if**(p > 19) {

error = PARSE\_OVERFLOW;

**return** 0;

}

c=gc();

}

ugc();

**if**(zero) num[p++]='0';

num[p]=0;

*/\* The largest 64 bit long long value. \*/*

ref = "9223372036854775807";

**if**(strlen(num) == strlen(ref) && strcmp(num, ref) > 0) {

error = PARSE\_OVERFLOW;

**return** 0;

}

**if**(sscanf(num,"%lld",&expr) != 1) {

error = PARSE\_OVERFLOW;

**return** 0;

}

**return** expr;

} **else** {

error=PARSE\_UNEXPECTED;

**return** 0;

}

}

TYPE Factor() {

**char** c = gc();

*/\* Unary +/- is handled here. \*/*

**if**(c == '-') {

**return** -Factor();

} **else** **if**(c == '+') {

**return** Factor();

} **else** {

ugc();

**return** Unit();

}

}

TYPE Term() {

TYPE total;

TYPE factor;

**char** c;

total = Factor();

**if**(error) **return** 0;

**while**(1) {

c = gc();

**if**(c == '\*') {

total \*= Factor();

**if**(error) **return** 0;

} **else** **if**(c == '/') {

factor = Factor();

**if**(error) **return** 0;

**if**(factor == 0) {

error = DIV\_BY\_ZERO;

**return** 0;

}

total /= factor;

} **else** {

ugc();

**return** total;

}

}

}

TYPE Expr() {

TYPE total;

**char** c;

total = Term();

**if**(error) **return** 0;

**while**(1) {

c = gc();

**if**(c == '-') {

total -= Term();

**if**(error) **return** 0;

} **else** **if**(c == '+') {

total += Term();

**if**(error) **return** 0;

} **else** {

ugc();

**return** total;

}

}

}

TYPE parse(**char** \*str) {

TYPE res;

s = str;

error = SUCCESS;

pos = 0;

res = Expr();

**if**(!error && gc() != 0) error=PARSE\_TOOLONG;

**return** res;

}

**void** chomp(**char** \*s) {

**int** len = strlen(s);

**if**(s[len-1]=='\n') s[len-1]=0;

}

**int** main() {

**char** line[1000];

TYPE res;

**while**(fgets(line,1000,stdin)) {

res=parse(line);

chomp(line);

printf("Parsed: %s\n",line);

**switch**(error) {

**case** SUCCESS:

printf("The value of the expression is: %lld\n",res);

**break**;

**case** DIV\_BY\_ZERO:

printf("A division by zero occured.\n");

**break**;

**case** PARSE\_UNEXPECTED:

printf("An unexpected character was encountered.\n");

**break**;

**case** PARSE\_TOOLONG:

printf("There was extra junk at the end of the string.\n");

**break**;

**case** PARSE\_OVERFLOW:

printf("An integer was parsed that would not fit in int.\n");

**break**;

**case** PARSE\_NOPAREN:

printf("A closing parenthesis was not found.\n");

**break**;

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Parsing:*

## Converting infix to postfix

*=================================================================*

*Description: Given a infix expression in a string, as well as*

*rules for prescedence of binary operators, returns*

*the postfix version of the expression.*

*Complexity: O(N) where N is the size of the string*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Sept 13, 2002*

*References: www.cs.usyd.edu.au/~loki/cs2ll/infix/InToPost.html*

*-----------------------------------------------------------------*

*Reliability: 1 successful use (Spain Problem 727) Sept 2002*

*Notes: The string should not contain any spaces.*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 1000

**int** pres[300];

**void** setPres(**char** op, **int** p){

pres[(**int**)op] = p;

}

**void** infix\_to\_postfix(**char** \*infix, **char** \*post){

**int** stack[MAXN];

**int** head = 0, len = strlen(infix);

**int** pos = 0, i;

**for**(i = 0; i < len; i++){

**switch**(infix[i]){

**case** '(': stack[head++] = infix[i]; **break**;

**case** ')':

**while**(stack[--head] != '('){

post[pos++] = stack[head];

}

**break**;

**case** '+': **case** '-': **case** '\*': **case** '/':

**while**(head && pres[stack[head-1]] >= pres[(**int**)infix[i]]){

post[pos++] = stack[--head];

}

stack[head++] = infix[i];

**break**;

**default**:

post[pos++] = infix[i];

}

}

**while**(head){

post[pos++] = stack[--head];

}

post[pos] = 0;

}

**int** main(){

**char** infix[MAXN];

**char** postfix[MAXN];

setPres('+', 1);

setPres('-', 1);

setPres('\*', 2);

setPres('/', 2);

**while**(scanf(" %s", infix) == 1){

printf("Infix: [%s] ", infix);

infix\_to\_postfix(infix, postfix);

printf("Postfix: [%s]\n", postfix);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Parsing:*

## Arithmetic Expression Parsing Template

*=================================================================*

*Description: A template for parsing arithmetic expressions.*

*This version handles +,-,\*,/,^, if all values*

*and intermediate values fit within integers, as*

*well as a variable mapping*

*Complexity: O(n)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Oct 22, 2003*

*References: Based on code by Kisman*

*-----------------------------------------------------------------*

*Reliability: 1 (Topcoder 2003 TCO Online Round 2 1000 pt)*

*Notes: - Doesn't handle overflow, but may be fixed by*

*changing types or explicit checking*

*- Doesn't handle division by 0*

*- Doesn't handle whitespace*

*\*/*

#include <stdio.h>

#include <string>

#include <math.h>

#include <map>

**using** **namespace** std;

map<string, **int**> var;

**int** ParseExp(string e){

**int** p, x, a, b;

**for**(x = e.size()-1, p = 0; x >= 0; x--){

p += (e[x] == ')') - (e[x] == '(');

**if**(!p && (e[x] == '+' || e[x] == '-')){

a = ParseExp(e.substr(0,x));

b = ParseExp(e.substr(x+1));

*// Check overflow - underflow here*

**return** e[x] == '+' ? a+b : a-b;

}

}

**for**(x = e.size()-1, p = 0; x >= 0; x--){

p += (e[x] == ')') - (e[x] == '(');

**if**(!p && (e[x] == '\*' || e[x] == '/')){

a = ParseExp(e.substr(0,x));

b = ParseExp(e.substr(x+1));

*// Check overflow - underflow & div/0 here*

**return** e[x] == '\*' ? a\*b : a/b;

}

}

**for**(x = e.size()-1, p = 0; x >= 0; x--){

p += (e[x] == ')') - (e[x] == '(');

**if**(!p && e[x] == '^'){

a = ParseExp(e.substr(0,x));

b = ParseExp(e.substr(x+1));

*// Check overflow here*

**return** (**int**)pow((**double**)a,b);

}

}

**if**(e[0] == '(') **return** ParseExp(e.substr(1,e.size()-2));

**if**(isdigit(e[0])) **return** atoi(e.c\_str());

**return** var[e];

}

**int** main(){

**int** numv, i, x; **char** buff[1000];

*/\* Read in number of variables \*/*

scanf("%d", &numv);

*/\* Read in variables and their assignments into the map \*/*

var.clear();

**for**(i = 0; i < numv; i++){

scanf("%s %d", buff, &x);

var[buff] = x;

}

*/\* Read expression \*/*

**while**(scanf("%s", buff) == 1)

printf("[%s] = %d\n", buff, ParseExp(buff));

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

############################################

# Search

############################################

*/\* Search:*

## Binary Search Template

*=================================================================*

*Description: Given a sorted array A of size n, it tried to find*

*an item x in the array using binary search. The*

*function returns non-zero if x is found, and zero*

*otherwise. Furthermore, if it is found, then*

*A[index] = x. If it is not found, then index is the*

*place x should be inserted into A.*

*ie. A[i] <= x for 0 <= i < index*

*x < A[i] for index <= i < n*

*There is also an insert routine here that will*

*insert the element into the right place after the*

*array has been reallocated (if necessary) to*

*store n+1 elements*

*Complexity:*

*-----------------------------------------------------------------*

*Author: Howard Cheng*

*Date: Nov 10, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: This routine is written for integer arrays, but can*

*be adapted to other types by changing the*

*comparison operator.*

*\*/*

#include <stdio.h>

**int** bin\_search(**int** \*A, **int** n, **int** x, **int** \*index){

**int** low, up, mid;

**if** (n <= 0 || x < A[0]) {

\*index = 0;

**return** 0;

}

**if** (A[n-1] < x) {

\*index = n;

**return** 0;

}

**if** (x == A[n-1]) {

\*index = n-1;

**return** 1;

}

**for**(low = 0, up = n-1; low + 1 < up;){

mid = (low+up)/2;

**if** (A[mid] <= x)

low = mid;

**else**

up = mid;

}

**if** (A[low] == x) {

\*index = low;

**return** 1;

} **else** {

\*index = up;

**return** 0;

}

}

**void** insert(**int** \*A, **int** n, **int** x, **int** index){

**int** i;

**for** (i = n-1; i >= index+1; i--)

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

A[index] = x;

}

**int** main(**void**){

**int** A[10000];

**int** n, i, x, index;

n = 0;

**while** (scanf("%d", &x) == 1 && n < 10000) {

**if** (!bin\_search(A, n, x, &index)) {

n++;

insert(A, n, x, index);

}

printf("List:");

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

printf(" %d", A[i]);

**if** (i == index) printf("\*");

}

printf("\n");

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## (r-g)-Cage search template

*=================================================================*

*Description: This is a template for finding minimum order r-g*

*cages.*

*A r-g cage is a graph where each vertex has degree r*

*and the smallest cycle in the graph is of size g.*

*The minimum order cage is the one with the fewest*

*vertices.*

*The search builds a necessary tree, then searchs on*

*the remaining vertices. The final adjaceny list of*

*the cage is stored in adj[][], where where adj[i][j]*

*stores the jth neighbor of vertex i.*

*Complexity: O(exponential)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 12, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: r-3 cages are easy to find - complete cliques*

*r-4 cages are also easy to find.*

*Other cage values that run fast:*

*3-(5,6,7,8), 4-(5,6)*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXR 10 */\* MAXR = max. degree of each vertex \*/*

#define MAXG 100 */\* MAXG = max. girth of cage \*/*

#define MAXN 100 */\* MAXN = max. # of vertices in cage \*/*

**int** adj[MAXN][MAXR]; */\* Adj list of vertex i \*/*

**int** deg[MAXN]; */\* Current degree of vertex i \*/*

**int** r, g, n; */\* Cage parameters \*/*

#define max(x,y) (((x)>(y))?(x):(y))

#define addEdge(x,y) adj[x][deg[x]++]=y;adj[y][deg[y]++]=x

#define remEdge(x,y) deg[x]--;deg[y]--

**void** FillTree(**int** root, **int** depth){

**int** i;

**for**(i = 0; i < r-1; i++){

n++;

addEdge(root, n);

**if**(depth > 1) FillTree(n, depth-1);

}

}

*/\* Builds the tree required by the lower bound \*/*

**void** BuildTree(){

**int** i;

n = 1;

memset(deg, 0, **sizeof**(deg)); */\* Clear graph \*/*

**if**(g % 2){ */\* Odd girth \*/*

**for**(i = 0; i < r; i++, n++){

addEdge(0,n);

**if**(g/2-1) FillTree(n,g/2-1);

}

} **else** { */\* Even girth \*/*

addEdge(0,n);

FillTree(n, g/2-1);

n++;

**for**(i = 1; i < r; i++, n++){

addEdge(0,n);

**if**(g/2-2) FillTree(n,g/2-2);

}

}

}

**void** PrintCage(){

**int** i, j;

printf("Found (%d,%d)-cage of size %d\n", r, g, n);

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

printf("%d:", i);

**for**(j = 0; j < r; j++) printf(" %d", adj[i][j]);

printf("\n");

*/\* Make sure each vertex has degree r \*/*

**if**(deg[i] != r) printf("ERROR!\n");

}

}

*/\* Determines if adding the edge x->y would create a (<g)-cycle \*/*

**int** Cycle(**int** x, **int** y){

**int** q[MAXN], tail = 1, head, node, next, i;

**char** dist[MAXN];

memset(dist, -1, **sizeof**(dist));

q[0] = y;

dist[y] = 0;

**for**(head = 0, tail = 1; head != tail; head++){

node = q[head];

**for**(i = 0; i < deg[node]; i++){

next = adj[node][i];

**if**(dist[next] == -1){

**if**(next == x) **return** 1;

dist[next] = dist[node]+1;

**if**(dist[node] < g-3) q[tail++] = next;

}

}

}

**return** 0;

}

**int** Search(**int** node){

**int** i;

**if**(node == n){ */\* Found cage \*/*

PrintCage();

**return** 1;

}

*/\* Check if vertex has full degree \*/*

**if**(deg[node] == r) **return** Search(node+1);

*/\* Determine the next possibile candidate \*/*

**if**(deg[node] == 0) i = node+1;

**else** i = max(node+1, adj[node][deg[node]-1]+1);

*/\* Try connecting this vertex \*/*

**for**(; i < n; i++){

**if**(deg[i] == r || Cycle(node, i)) **continue**;

addEdge(node, i);

**if**(Search(node)) **return** 1;

remEdge(node, i);

}

**return** 0;

}

**int** main(){

printf("Enter r and g: ");

**while**(scanf("%d %d", &r, &g) == 2){ */\* Read cage parameters \*/*

**if**(r < 2 || r > MAXR || g < 3 || g > MAXG){

printf("Invalid parameters.\n");

**continue**;

}

BuildTree(); */\* Builds the tree first \*/*

**while**(1){ */\* Search for the cage \*/*

printf("Searching for cage of size %d\n", n);

**if**(Search(0)) **break**;

n++;

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## Graph coloring template

*=================================================================*

*Description: Given a graph, this is a template for k-coloring*

*it.*

*Complexity: O(exponential)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Mar 14, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: -The graph is given by the adjacency matrix adj[][].*

*-The coloring of the graph is stored in color[];*

*-Colors run from 0...k-1*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 50

**int** adj[MAXN][MAXN], n, k, total;

**int** color[MAXN];

**void** BackTrack(**int** pos){

**int** i, c;

**if**(pos == n){

total++;

printf("{");

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

printf("%d%c", color[i], i == n-1 ? '}' : ',');

printf("%c", total % 5 ? ' ' : '\n');

**return**;

}

**for**(c = 0; c < k; c++){

**for**(i = 0; i < pos; i++)

**if**(adj[i][pos] && color[i] == c) **break**;

**if**(i == pos){

color[pos] = c;

BackTrack(pos+1);

}

}

}

**int** main(){

**int** i, j;

**while**(scanf("%d %d", &n, &k) == 2){

**for**(i = 0; i < n; i++) **for**(j = 0; j < n; j++)

scanf("%d", &adj[i][j]);

total = 0;

BackTrack(0);

printf("Total %d-colorings: %d\n", k, total);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## Golden section Search

*=================================================================*

*Description: Given an function f(x) with a single local minimum,*

*a lower and upper bound on x, and a tolerance for*

*convergence, this function finds the value of x*

*The function is written globally as f(x)*

*Complexity: Depends on tolerance*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Mar 14, 2003*

*References: www.chemeng.ed.ac.uk/~jwp/MSO/section5/maths/*

*part3/handout2/*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - watch out for -0.000*

*\*/*

#include <stdio.h>

#define GOLD 0.381966

#define move(a,b,c) x[a]=x[b];x[b]=x[c];fx[a]=fx[b];fx[b]=fx[c]

**double** f(**double** x){

**return** x\*x;

}

**double** golden(**double** xlow, **double** xhigh, **double** tol){

**double** x[4], fx[4], L;

**int** iter = 0, left = 0, mini, i;

fx[0] = f(x[0]=xlow);

fx[3] = f(x[3]=xhigh);

**while**(1){

L = x[3]-x[0];

**if**(!iter || left){

x[1] = x[0]+GOLD\*L;

fx[1] = f(x[1]);

}

**if**(!iter || !left){

x[2] = x[3]-GOLD\*L;

fx[2] = f(x[2]);

}

**for**(mini = 0, i = 1; i < 4; i++)

**if**(fx[i] < fx[mini]) mini = i;

**if**(L < tol) **break**;

**if**(mini < 2){

left = 1;

move(3,2,1);

} **else** {

left = 0;

move(0,1,2);

}

iter++;

}

**return** x[mini];

}

**int** main(){

**double** low, high , tol, minx;

**while**(scanf("%lf %lf %lf", &low, &high, &tol) == 3){

minx = golden(low, high, tol);

printf("f(%.6f) = %.6f\n", minx, f(minx));

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## Hamming Code search template / (Independent set)

*=================================================================*

*Description: This is a template for finding a maximum set of*

*binary code words that are all have at least a*

*Hamming distance of d between them.*

*Also has some ideas for finding maximum clique*

*Complexity: O(exponential)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 14, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <string.h>

#include <algorithm>

#include <vector>

#include <iostream>

**using** **namespace** std;

#define MAXN 9

#define MAXS 21

**int** n, d;

**int** bitcount(**int** x){

**int** c = 0;

**while**(x){

c++;

x &= x-1;

}

**return** c;

}

**int** dist(**int** x, **int** y){

**return** bitcount(x^y);

}

**void** PrintBin(**int** x){

**int** i;

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

printf("%d", x & (1<<i) ? 1 : 0);

printf("\n");

}

**int** best;

**int** curr[MAXS];

**int** bestset[MAXS];

vector<**int**> pos[MAXS];

vector<**int**> adj[1<<MAXN];

**void** MaxClique(**int** k){

**int** i; vector<**int**>::iterator it, it2;

**if**(k > best){

best = k;

**for**(i = 0; i < k; i++) bestset[i] = curr[i];

}

**for**(it = pos[k].begin(); it != pos[k].end(); ++it){

curr[k] = \*it;

pos[k+1].clear();

insert\_iterator<vector<**int**> > next(pos[k+1], pos[k+1].begin());

it2 = it; it2++;

set\_intersection(it2, pos[k].end(),

adj[\*it].begin(), adj[\*it].end(),

next);

**if**(k+1+(**int**)pos[k+1].size() <= best) **continue**;

MaxClique(k+1);

}

}

**int** main(){

**int** i, j;

**while**(scanf("%d %d", &n, &d) == 2){

**for**(i = 0; i < MAXS; i++) pos[i].clear();

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

adj[i].clear();

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

**if**(dist(i,j) >= d)

adj[i].push\_back(j);

}

}

best = -1;

curr[0] = 0;

pos[1] = adj[0];

MaxClique(1);

printf("Maximum code length: %d\n", best);

**for**(i = 0; i < best; i++)

PrintBin(bestset[i]);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## Orthogonal-Latin Squares

*=================================================================*

*Description: This is a template for finding orthogonal Latin*

*Squares.*

*A latin square is a N by N matrix which uses the*

*numbers 0..N-1, such that each number occurs exactly*

*once in each column, and each row.*

*Two Latin squares are orthogonal if the square*

*resulting from the overlapping of the two squares*

*has no pair of elements that occurs twice.*

*e.g.*

*1 2 3 1 2 3 [1,1] [2,2] [3,3]*

*2 3 1 3 1 2 -> [2,3] [3,1] [1,2]*

*3 1 2 2 3 1 [3,2] [1,3] [2,1]*

*An k-set of orthogonal Latin squares is a set of*

*k Latin squares such that every pair is orthogonal.*

*Complexity: O(exponential)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Mar 19, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: - The number of subsets of orthogonal Latin squares*

*can be enormous.*

*- If the order is N, there is at most a N-1 set of*

*orthogonal Latin squares*

*- This program finds a maximum set quickly for 3,4,5*

*but not for any larger.*

*\*/*

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXN 10 */\* Maximum order \*/*

#define MAXK 10 */\* Maximum set size \*/*

**int** n, setsize;

**int** curr[MAXK][MAXN][MAXN]; */\* Stores the Latin squares \*/*

**char** row[MAXK][MAXN][MAXN]; */\* Used to mark used elements \*/*

**char** col[MAXK][MAXN][MAXN]; */\* Used to mark used elements \*/*

**char** ort[MAXK][MAXK][MAXN][MAXN]; */\* Used to mark used elements \*/*

*/\* Print procedure. Modify as necessary \*/*

**void** PrintLatin(**int** size){

**int** i, j, k;

**for**(i = 0; i < n; i++) **for**(k = 0; k < size; k++){

printf("[");

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

printf("%d%c", curr[k][i][j], j == n-1 ? ']' : ',');

printf("%c", k == size-1 ? '\n' : ' ');

}

}

*/\* The search goes through the kth square, xth row, yth column,*

*with a special flag called "same" which forces an ordering on*

*the Latin squares, such that permutations are not considered*

*\*/*

**void** Search(**int** k, **int** x, **int** y, **int** same){

**int** i, j;

**if**(!x && !y){

**if**(k >= 2){

*/\* Completed another square \*/*

printf("[Subset %d] %d-set:\n", ++setsize, k);

PrintLatin(k);

}

same = k;

}

**for**(i = same ? curr[k-1][x][y] : 0; i < n; same = 0, i++){

**if**(row[k][x][i] || col[k][y][i]) **continue**;

**for**(j = 0; j < k && !ort[k][j][i][curr[j][x][y]]; j++);

**if**(j != k) **continue**;

row[k][x][i] = col[j][y][i] = 1;

**for**(j = 0; j < k; j++) ort[k][j][i][curr[j][x][y]] = 1;

curr[k][x][y] = i;

**if**(y == n-1){

**if**(x == n-1) Search(k+1, 0, 0, same);

**else** Search(k, x+1, 0, same);

} **else** {

Search(k, x, y+1, same);

}

row[k][x][i] = col[j][y][i] = 0;

**for**(j = 0; j < k; j++) ort[k][j][i][curr[j][x][y]] = 0;

}

}

**int** main(){

**while**(scanf("%d", &n) == 1){

setsize = 0;

Search(0,0,0,0);

printf("Total # of orthogonal squares of order %d: %d\n",

n, setsize);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## N non-attacking Queens

*=================================================================*

*Description: The following is a template for finding valid*

*configurations for the N-Queens problem.*

*Complexity: O(exponential)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: Mar 12, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability:*

*Notes: rows/cols labelled 0..n-1*

*\*/*

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXN 100

**int** S[MAXN], used[MAXN], n, total;

**void** nQueens(**int** pos){

**int** i, j;

**if**(pos == n){

total++;

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

printf("(%d,%d)%c", i, S[i], i == n-1 ? '\n' : ' ');

**return**;

}

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

**if**(used[i]) **continue**;

**for**(j = 0; j < pos; j++)

**if**(pos-j == abs(i-S[j])) **break**;

**if**(j == pos){

S[pos] = i;

used[i] = 1;

nQueens(pos+1);

used[i] = 0;

}

}

}

**int** main(){

**while**(scanf("%d", &n) == 1){

memset(used, 0, **sizeof**(used));

total = 0;

nQueens(0); */\* Call this to start recursion \*/*

printf("Total valid configurations for n=%d: %d\n", n,total);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Searching:*

## Suffix array

*=================================================================*

*Description: Builds a suffix array of a string of N characters*

*Complexity: O(N log N)*

*-----------------------------------------------------------------*

*Author: Howard Cheng*

*Date: Oct 30, 2003*

*References: Manber, U. and Myers, G. "Suffix Arrays: a New*

*Method for On-line String Searches."*

*SIAM Journal on Computing. 22(5) p. 935-948, 1993.*

*T. Kasai, G. Lee, H. Arimura, S. Arikawa, and*

*K. Park. "Linear-time Longest-common-prefix*

*Computation in Suffix Arrays and Its Applications."*

*Proc. 12th Annual Conference on Combinatorial*

*Pattern Matching, LNCS 2089, p. 181-192, 2001*

*-----------------------------------------------------------------*

*Reliability: 1 (Spain 719 - Glass Beads)*

*Notes: The build\_sarray routine takes in a string S of n*

*characters (null-terminated), and constructs two*

*arrays sarray and lcp. The properties are:*

*- If p = sarray[i], then the suffix of str starting at*

*p (i.e. S[p..n-1] is the i-th suffix when all the*

*suffixes are sorted in lexicographical order*

*- NOTE: the empty suffix is not included in this list,*

*so sarray[0] != n.*

*- lcp[i] contains the length of the longest common*

*prefix of the suffixes pointed to by sarray[i-1]*

*and sarray[i]. lcp[0] is defined to be 0.*

*- To see whether a pattern P occurs in str, you can*

*look for it as the prefix of a suffix. This can be*

*done with a binary search in O(|P| log n) time.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include <assert.h>

#define MAXN 100000

**int** bucket[CHAR\_MAX-CHAR\_MIN+1];

**int** prm[MAXN], count[MAXN];

**char** bh[MAXN+1];

**void** build\_sarray(**char** \*str, **int**\* sarray, **int** \*lcp){

**int** n, a, c, d, e, f, h, i, j, x;

n = strlen(str);

*/\* sort the suffixes by first character \*/*

memset(bucket, -1, **sizeof**(bucket));

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

j = str[i] - CHAR\_MIN;

prm[i] = bucket[j];

bucket[j] = i;

}

**for** (a = c = 0; a <= CHAR\_MAX - CHAR\_MIN; a++) {

**for** (i = bucket[a]; i != -1; i = j) {

j = prm[i];

prm[i] = c;

bh[c++] = (i == bucket[a]);

}

}

bh[n] = 1;

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

sarray[prm[i]] = i;

*/\* inductive sort \*/*

x = 0;

**for** (h = 1; h < n; h \*= 2) {

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

**if** (bh[i] & 1) {

x = i;

count[x] = 0;

}

prm[sarray[i]] = x;

}

d = n - h;

e = prm[d];

prm[d] = e + count[e];

count[e]++;

bh[prm[d]] |= 2;

i = 0;

**while** (i < n) {

**for** (j = i; (j == i || !(bh[j] & 1)) && j < n; j++) {

d = sarray[j] - h;

**if** (d >= 0) {

e = prm[d];

prm[d] = e + count[e];

count[e]++;

bh[prm[d]] |= 2;

}

}

**for** (j = i; (j == i || !(bh[j] & 1)) && j < n; j++) {

d = sarray[j] - h;

**if** (d >= 0 && bh[prm[d]] & 2) {

**for** (e = prm[d]+1; bh[e] == 2; e++) ;

**for** (f = prm[d]+1; f < e; f++) {

bh[f] &= 1;

}

}

}

i = j;

}

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

sarray[prm[i]] = i;

**if** (bh[i] == 2) {

bh[i] = 3;

}

}

}

h = 0;

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

e = prm[i];

**if** (e > 0) {

j = sarray[e-1];

**while** (str[i+h] == str[j+h]) {

h++;

}

lcp[e] = h;

**if** (h > 0) {

h--;

}

}

}

lcp[0] = 0;

}

**int** main(){

**char** S[MAXN]; **int** sarray[MAXN], lcp[MAXN], i;

**char** T[MAXN];

**int** n, j;

**while** (scanf("%s", S) == 1) {

n = strlen(S);

**for**(i = 0; i < n; i++) S[n+i] = S[i];

S[n+n] = 0;

build\_sarray(S, sarray, lcp);

**for** (i = 0; S[i]; i++)

**if**(sarray[i] < n){

printf("%3d: %2d [%d]\n", i, lcp[i], n);

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

printf("%c", S[sarray[i]+j]);

}

printf("\n");

}

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Search:*

## Self Avoiding walk template

*=================================================================*

*Description: A template for a self-avoiding walk search.*

*A self avoiding walk is one in which at each turn*

*one walks N, E, S or W, but never visits a place*

*he/she has visited.*

*Complexity: O(exponential)*

*-----------------------------------------------------------------*

*Author: Gilbert Lee*

*Date: March 12, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes:*

*\*/*

#include <stdio.h>

#include <string.h>

#define MAXN 100

**int** Sx[MAXN], Sy[MAXN], n, total;

**char** walk[MAXN];

**char** dir[5] = "NESW";

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

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

**void** Walk(**int** pos){

**int** i, j;

**if**(pos == n){

total++;

walk[pos] = 0;

printf("%s%c", walk, total%10 ? ' ' : '\n');

**return**;

}

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

Sx[pos+1] = Sx[pos]+dx[i];

Sy[pos+1] = Sy[pos]+dy[i];

walk[pos] = dir[i];

**for**(j = 0; j <= pos; j++)

**if**(Sx[j] == Sx[pos+1] && Sy[j] == Sy[pos+1]) **break**;

**if**(j == pos+1) Walk(pos+1);

}

}

**int** main(){

**while**(scanf("%d", &n) == 1){

total = Sx[0] = Sy[0] = 0;

Walk(0); */\* Start recursive call \*/*

printf("\nTotal # of self-avoiding walks of size %d = %d\n\n", n, total);

}

**return** 0;

}

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

*/\* Miscellaneous:*

## Word Search

*=================================================================*

*Description: Given a very long string, sometimes you have to find*

*a list of substrings in the string. This program*

*will be faster than the obvious brute force solution.*

*Complexity: O(N\*M lg N) setup, O(M lg N) each query*

*where N is the length of your string,*

*M is the length of your substring.*

*-----------------------------------------------------------------*

*Author: Ashley Zinyk*

*Date: March 12, 2003*

*References:*

*-----------------------------------------------------------------*

*Reliability: 0*

*Notes: If you want to find out how many instances of the*

*substring B appear in the substring A, find the*

*index of last, subtract the index of the first, and*

*add 1. Remember to return 0 if you couldn't find*

*at least one.*

*The buffer for your long string should be at least*

*sizeof(longest string) + sizeof(longest substring) + 1.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

**int** maxm;

**int** ptrcmp(**const** **void** \*aa, **const** **void** \*bb) {

**char** \*a = \*(**char** \*\*)aa;

**char** \*b = \*(**char** \*\*)bb;

**int** res = strncmp(a,b,maxm);

**return** (res) ? res : a-b;

*/\* if two substrings are the same, keep them in their*

*original order \*/*

}

*/\* to find the first instance of the key, make fl = 0*

*to find the last instance of the key, make fl = 1 \*/*

**int** getindex(**char** \*key, **char** \*\*ptrs, **int** n, **int** fl) {

**int** lng, res, mid, upp = n, low = 0;

lng = strlen(key);

**while** (1) {

mid = (upp+low)/2;

res = strncmp(key,ptrs[mid],lng);

**if** (!res) {

**if** (fl) {

**if** (mid != n-1 && !strncmp(ptrs[mid+1],key,lng)) low = mid;

**else** **return** mid;

} **else** {

**if** (mid != 0 && !strncmp(ptrs[mid-1],key,lng)) upp = mid;

**else** **return** mid;

}

} **else** **if** (res < 0) {

**if** (low==mid) **return** -1;

low = mid;

} **else** {

**if** (upp==mid) **return** -1;

upp = mid;

}

}

}

*/\* str is the string you are searching in*

*n is the number of characters in the long string*

*k is the number of characters in the longest substring \*/*

**void** makeptrs(**char** \*str, **int** n, **char** \*\*ptrs, **int** k) {

**int** i;

**for** (i = 0; i < k; i++) str[n+k] = '\0';

**for** (i = 0; i < n; i++) ptrs[i] = &str[i];

maxm = k;

qsort(ptrs[i], n, **sizeof**(**char** \*), ptrcmp);

}

**char** message[1001000];

**char** \*ptrs[1001000];

**char** word[1000];

**int** main() {

**int** i, n, m, first;

scanf("%d",&n);

n \*= 64; */\* The SETI problem had n 64-byte lines \*/*

**for** (i = 0; i < n; i+=64)

scanf("%s",&message[i]); */\* input the string \*/*

makeptrs(message,n,ptrs,30);

scanf("%d",&m);

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

scanf("%s",word);

first = getindex(word,ptrs,n,0);

**if** (first < 0) {

printf("%s not found\n",word);

} **else** {

printf("There are %d instances of %s in the message\n",

getindex(word,ptrs,n,1)-first+1,word);

printf("The first instance was found at index %d\n",first);

}

}

**return** 0;

}