```cpp
//vote.h
#ifndef __VOTE_H_
#define __VOTE_H_


#include <iostream>
#include <vector>
#include <stdio.h>
#include <math.h>


using namespace std;


//extern const int MAX=999;
#define MAX 999
//MAX is the maximum capacity that can test the number of points
extern double DEVIATION;
/*
DEVIATION is the error accuracy for judging
whether a set of points conforms to the situation
*/

extern int AllVotes[MAX][MAX];
//AllVotes store the result of votes to find match points
extern int M;
//Small summary points
extern int N;
//Big summary points
extern int maxVotes;
//Record the maximum number of votes received

struct TreeNode {
    int level;
    //The level for this TreeNode
    int num;
    //Indicates that this point is at the subscript of the input sequence
    double standard;
    /*Record the standard length of this corresponding graphic
    Notice:Take the first edge of this graph as the base
    */
    int votes;
    //Record the number of votes this edge has received
    int ancestor;
    //Record the subscript of the corresponding root node of this tree
    vector<TreeNode*> child;
    //Use vector to store the Node's Child Nodes
    TreeNode* parent;
    //Store the parent node in it
};
```

```cpp
struct Point {
    double x;
    double y;
};
//A struct to indicates the horizontal and vertical coordinates of a point
extern Point Node1[MAX];   //Store the less points
extern Point Node2[MAX];   //Store the larger points

inline void Swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
//An inline function to swap two points

inline int Mod(int x, int y) {
    if(x%y==0) return x;
    else return x%y;
}
/*An inline function to calculate x%y but return x for result 0
Its role is to facilitate the correct execution of subsequent traversal
*/

inline double CalLength(Point a,Point b) {
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}
//Calculate the distance between two points

inline double CalAgle(double l1,double l2, double l3) {
    return acos((l1*l1+l2*l2-l3*l3)/(2*l1*l2));
}
//Calculate the angle of the triangle formed by the three sides

inline double CalDeviation(double x,double y)
{
    if(x>y) {
        return (x-y)/y;
    }
    else {

        return (y-x)/y;
    }
}
//An inline function to calculate relative error


struct JudgeNode {
double length1;
double length2;
```

```
double agle;
};
/*
An indicator used to judge whether a set of points meets the standard
Contains two sides and their included angle
*/

extern vector<TreeNode*> Forest;
//A forest structure to store all eligible trees

extern TreeNode* MaxPath[MAX];
/*
Record the longest path obtained during the voting process
and this is the final result
*/

extern TreeNode* ThePath[MAX];
//Record the currently judged path when finding the longest path

extern JudgeNode* Node1_Judger[MAX];
//Stores the judgment information of the reference graph


inline int Add(int a) {
    a+=1;
    if(a<=N) {
        return a;
    }
    else {
        return 1;
    }
}
//Add 1 to a for each time and make sure a is between 1 and N

inline int Sub(int a) {
    a-=1;
    if(a>=1) {
        return a;
    }
    else {
        return N;
    }
}
//Sub 1 to a for each time and make sure a is between 1 and N

void CopyPath();
//Copy the current path to the maximum path


void DFS(TreeNode* father,int sum);
//Find the path with the longest votes
```

```
TreeNode* MakeRoot(int num);
//Generate a root node for the specified node

void ReadPoint();
//Read test data midpoint

void MakeJudger();
//Generates criteria for making test judgments

TreeNode* InsertNode(TreeNode* parentNode,int level,int num,double standard,int ancesto
//Insert child nodes

bool Judger(TreeNode* child);
//Determine whether the generated node meets the requirements

TreeNode* MakeTree1(TreeNode* root,double standard);
//Create tree in clockwise direction

TreeNode* MakeTree2(TreeNode* root,double standard);
//Create a tree counterclockwise

vector<TreeNode*> MakeForest();
//Build all possible trees

TreeNode* DeleteNode(TreeNode* father);
//Delete nodes that cannot form a complete path

int GetVotes(TreeNode* father);
//Count the number of path votes for nodes

void ForAllVotes(TreeNode* root);
//Count the final voting results

bool JudgeVotes();
//Determine whether the result of the vote has been generated

#endif
```

```cpp
//vote.cpp
#include <iostream>
#include <vector>
#include <stdio.h>
#include <math.h>
#include "vote.h"


using namespace std;

double DEVIATION = 0;
//const int MAX=999;
int AllVotes[MAX][MAX];
int M;
int N;
int maxVotes = 0;
struct TreeNode;
struct Point;
Point Node1[MAX];
Point Node2[MAX];
struct JudgeNode;
vector<TreeNode*> Forest;
TreeNode* MaxPath[MAX];
TreeNode* ThePath[MAX];
JudgeNode* Node1_Judger[MAX];


void CopyPath() {
    for (int i=1; i<=M; i++) {
        MaxPath[i] = ThePath[i];
    }
    //Copy the current path to the maximum path
}

void DFS(TreeNode* father,int sum)
{
    if(father->level==M) {
        //reached the last node of the path
        ThePath[father->level] = father;
        sum+=AllVotes[father->level][father->num];
        if(sum>=maxVotes) {
            maxVotes = sum;
            CopyPath();
            //log this path if it is longer
        }
        return;
    }
    else {
        ThePath[father->level] = father;
        //Record the elements traversed by the current path
```

```
        for (int i=0; i<father->child.size(); i++) {
            DFS(father->child[i],sum+AllVotes[father->level][father->num]);
            //traverse every possible path
        }
    }

}

TreeNode* MakeRoot(int num) {
    TreeNode* root = new TreeNode;
    root->level = 1;
    //The level of the root node is marked as 1
    root->num = num;
    root->standard = num;
    /*
    At this time, the standard length of the sub-graphics is not generated,
    but a simple placeholder processing is performed.
    */
    root->parent = NULL;
    root->votes = 0;
    //Initialize the number of votes to start with
    root->ancestor = num;
    return root;
}
//create a root node


void ReadPoint()
{
    scanf("%d %d",&M,&N);
    if(M>N) {
        for (int i=1; i<=M; i++) {
            scanf("%lf %lf",&Node2[i].x,&Node2[i].y);
        }

        for (int i=1; i<=N; i++) {
            scanf("%lf %lf",&Node1[i].x,&Node1[i].y);
        }
        Swap(M,N);
    }

    else {
        for (int i=1; i<=M; i++) {
            scanf("%lf %lf",&Node1[i].x,&Node1[i].y);
        }

        for (int i=1; i<=N; i++) {
            scanf("%lf %lf",&Node2[i].x,&Node2[i].y);
        }
    }
```

```
}
/*
Read each point, and make sure that
Node1 records the number of nodes with a small number of nodes,
and Node2 records the number of nodes with a large number of nodes
*/


void MakeJudger()
{
    Point pointA;
    Point pointB;
    Point pointC;
    for (int i=1; i<=M; i++) {
        pointA = Node1[i];
        pointB = Node1[Mod(i+1,M)];
        pointC = Node1[Mod(i+2,M)];
        /*
        Take the subscript and start the side length of the triangle
        formed by the three points clockwise
        */
        double l1 = CalLength(pointA,pointB);
        double l2 = CalLength(pointB,pointC);
        double l3 = CalLength(pointC,pointA);
        //Use CalLength to calculate the length
        Node1_Judger[i] = new JudgeNode;
        Node1_Judger[i]->length1 = l1;
        Node1_Judger[i]->length2 = l2;
        Node1_Judger[i]->agle = CalAgle(l1,l2,l3);
        //Use CalAgle to calculate the agle
        //Record the corresponding angle and side length in the judger

    }

}


TreeNode* InsertNode(TreeNode* parentNode,int level,int num,double standard,int ancesto
{
    TreeNode* childNode = new TreeNode;
    //generate a child node
    childNode->level = level;
    childNode->num = num;
    childNode->standard = standard;
    childNode->parent = parentNode;
    //Record incoming parameters
    childNode->votes = 0;
    //initial the vote to zero
    childNode->ancestor = ancestor;
    //points to its ancestor
```

```
    return childNode;
}


bool Judger(TreeNode* child) {

    Point pointA = Node2[child->parent->parent->num];
    Point pointB = Node2[child->parent->num];
    Point pointC = Node2[child->num];
    //Get three consecutive points counterclockwise from this node
    double l1 = CalLength(pointA,pointB);
    double l2 = CalLength(pointB,pointC);
    double l3 = CalLength(pointC,pointA);
    //Calculate the length of the three sides formed by the three points
    double judge_angle = CalAgle(l1,l2,l3);
    //Calculate the angle of the top foot
    JudgeNode* the_judger = Node1_Judger[child->level-2];
    double standard_l1 = l1/child->standard;
    double standard_l2 = l2/child->standard;
    //normalize length

    double judger_l1 = the_judger->length1/Node1_Judger[1]->length1;
    double judger_l2 = the_judger->length2/Node1_Judger[1]->length1;
    /*Screening by judging the relative error of side length and angle
    */
    if(CalDeviation(standard_l1,judger_l1)>DEVIATION) {
        return false;
        //judge the deviation of l1
    }

    if(CalDeviation(standard_l2,judger_l2)>DEVIATION) {
        return false;
        //judge the deviation of l2
    }

    if(CalDeviation(judge_angle,the_judger->agle)>DEVIATION) {
        return false;
        //judge the deviation of agle
    }
    //If the error is within the accuracy return true
    return true;

}


//Build in a clockwise direction
TreeNode* MakeTree1(TreeNode* root,double standard)
{   int this_level = root->level+1;
    if(this_level>M) {
        return root;
        //If the number of layers of the node reaches M, the tree is completed
```

```
    }
    for (int i=Add(root->num); i!=root->ancestor; i=Add(i)) {
        //Iterate over i monotonically
        if(N-i+1+root->ancestor-1>=M-this_level+1) {
            /*If the number of remaining nodes is sufficient, build a tree
            */
            if(this_level<3) {
            TreeNode* childNode = InsertNode(root,this_level,i,standard,root->ancestor)
            double standard_length = CalLength(Node2[childNode->num],Node2[root->num]);
            root->child.push_back(MakeTree1(childNode,standard_length));
            //If the current number of nodes is less than 3, you can directly insert
            }
            else {
                TreeNode* childNode = InsertNode(root,this_level,i,standard,root->ances
                if(Judger(childNode)==true) {

                    root->child.push_back(MakeTree1(childNode,standard));
                    /*
                    When the number of nodes is greater than 3,
                    the insertion operation is performed only if the conditions are met
                    */
                }
                else {
                    delete childNode;
                    //If the conditions are not met, this node will be deleted
                }
            }

        }
        else {
            break;
            /*If the number of remaining nodes is insufficient,
             the tree will not be built
            */
        }
    }
    return root;
}


//Build in a counterclockwise direction
TreeNode* MakeTree2(TreeNode* root,double standard)
{   int this_level = root->level+1;
    if(this_level > M) {
        return root;
        //If the number of layers of the node reaches M, the tree is completed
    }
    for (int i=Sub(root->num); i!=root->ancestor; i=Sub(i)) {
        //Iterate over i monotonically
        if(i+N-root->ancestor>=M-this_level+1) {
            /*If the number of remaining nodes is sufficient, build a tree
```

```
            */
            if(this_level<3) {
            TreeNode* childNode = InsertNode(root,this_level,i,standard,root->ancestor)
            double standard_length = CalLength(Node2[childNode->num],Node2[root->num]);
            root->child.push_back(MakeTree2(childNode,standard_length));
            //If the current number of nodes is less than 3, you can directly insert
            }
            else {
                TreeNode* childNode = InsertNode(root,this_level,i,standard,root->ances
                if(Judger(childNode)==true) {
                    root->child.push_back(MakeTree2(childNode,standard));
                }
                else {
                    delete childNode;
                }
                /*
                    When the number of nodes is greater than 3,
                    the insertion operation is performed only if the conditions are met
                */
            }
        }
        else {
            /*If the number of remaining nodes is insufficient,
             the tree will not be built
            */
            break;
        }
    }
    return root;
}

TreeNode* DeleteNode(TreeNode* father)
{

    if(father->level == M) {
        return father;
        /*
        If you reach the end of the path,
        it means that the path meets the length requirements,
        and you can return directly.
        */
    }

    if(father->child.size()==0) {
        return NULL;
        /*
        If the last layer is not reached,
        and the node has no child nodes, delete the node
        */
    }
```

```
    for (int i=0; i<father->child.size(); i++) {
        if(father->level+1<M) {
            father->child[i] = DeleteNode(father->child[i]);
            //If there is a child node, Judgment processing for its child node
        }
    }

    vector<TreeNode*> :: iterator it;
    //The iterator is used to judge the node after the delete operation
    TreeNode* delete_father = new TreeNode;
    //The result of processing and judging the node after the operation
    delete_father->level = father->level;
    delete_father->num = father->num;
    delete_father->parent = father->parent;
    delete_father->standard = father->standard;
    //Copy the corresponding fixed information
    for (it=father->child.begin(); it!=father->child.end(); it++) {

        if(*it!=NULL) {
            delete_father->child.push_back((*it));
            //Record only if the child node is not NULL
        }
    }
    free(father);
    //clear old nodes
    if(delete_father->child.size()==0) {
        return NULL;
        //If the node has no child nodes after the operation return NULL
    }
    else {
        return delete_father;
        //If there is a child node, return the node
    }
}

vector<TreeNode*> MakeForest()
{   vector<TreeNode*> Forest;
    for (int i=1; i<=N; i++) {

            TreeNode* root1 = MakeRoot(i);
            root1 = MakeTree1(root1,-1);
            //Store the root node of the tree built in a clockwise direction
            Forest.push_back(root1);

            TreeNode* root2 = MakeRoot(i);
            root2 = MakeTree2(root2,-1);
            //Store the nodes that build the tree in a counterclockwise direction
            Forest.push_back(root2);

    }
    return Forest;
```

```
}


int GetVotes(TreeNode* father)
{   if(father->level == M) {
        father->votes = 1;
        //The number of end nodes of the path is 1
        return 1;
    }
    father->votes = 0;
    for (int i=0; i<father->child.size(); i++) {
        father->votes += GetVotes(father->child[i]);
        //The sum of the votes of the parent node and the votes of the child nodes
    }
    return father->votes;
}

void ForAllVotes(TreeNode* root)
{
    if(root->level==M) {
        AllVotes[root->level][root->num] += root->votes;
        //If there is no child node, the number of votes is recorded directly
        return;
    }
    else {
        AllVotes[root->level][root->num] += root->votes;
        //Record the number of votes for the current node
        for (int i=0; i<root->child.size(); i++) {
            ForAllVotes(root->child[i]);
            //Count the votes of child nodes
        }
    }
}

bool JudgeVotes() {
    for (int i=1; i<=M; i++) {
        for (int j=1; j<=N; j++) {
            if(AllVotes[i][j]!=0) {
                return true;
                //Determine whether the result of the vote has been generated
            }
        }
    }
    return false;
}
```

```cpp
//main.cpp
#include <iostream>
#include <vector>
#include <stdio.h>
#include <math.h>
#include "vote.h"
#include <fstream>

using namespace std;

int main()
{
    ReadPoint();
    //Read all information of nodes
    MakeJudger();
    //generate data for judgment
   while(true){
   Forest = MakeForest();
    //Make the forest for result
    for (int i=0; i<Forest.size(); i++) {
        Forest[i] = DeleteNode(Forest[i]);
        //Prune unqualified paths
    }
    for (int i=0; i<Forest.size(); i++) {
        if(Forest[i]!=NULL) {
            Forest[i]->votes = GetVotes(Forest[i]);
            //Statistical processing of the votes of the path
        }
    }


    for (int i=0; i<Forest.size(); i++) {
        if(Forest[i]!=NULL) {
            ForAllVotes(Forest[i]);
            //If the path is not NULL, record the data in the voting table
        }
    }

    if(JudgeVotes()==true) {

        ofstream outfile;
        outfile.open("votes.txt",ios::app);
        for (int i=1; i<=M; i++) {
         for (int j=1; j<=N; j++) {
             //printf("%d ",AllVotes[i][j]);
             outfile<<AllVotes[i][j]<<" ";
         }
        /*If data is generated in the voting table,
         output the voting table and end the loop
        */
```

```
        outfile<<endl;

    }
    outfile<<"_______________________________________"<<endl;
    break;
    }
    else {
        DEVIATION+=0.05;
        /*If the voting table fails to be generated,
        increase the allowable value of the error
        */
        if(DEVIATION>1) break;
        //Avoid the upper limit set by the infinite loop
    }
   }
    cout<<"___________________"<<endl;

    for (int i=0; i<Forest.size(); i++) {
        if(Forest[i]!=NULL) {
            DFS(Forest[i],0);
            //Find the path with the longest total votes
        }
    }

    for (int i=1; i<=M; i++) {
        if(MaxPath[i]!=NULL)
        printf("(%d,%d)\n",i,MaxPath[i]->num);
        //Output the corresponding corresponding point
    }
    cout<<"___________________"<<endl;
}
```