//2d Seg Tree max + sum

int a[1003][1003];

int t[3\*1005][3\*1005];

int tmn[3\*1005][3\*1005];

int r,c; // x y

inline int max(int a,int b)

{

if(a>b)

return a;

return b;

}

inline int min(int a,int b)

{

if(a>b)

return b;

return a;

}

void build\_y (int vx, int lx, int rx, int vy, int ly, int ry)

{

if (ly == ry)

if (lx == rx)

t[vx][vy] = a[lx][ly];

else

t[vx][vy] = t[vx\*2][vy] + t[vx\*2+1][vy];

else

{

int my = (ly + ry) / 2;

build\_y (vx, lx, rx, vy\*2, ly, my);

build\_y (vx, lx, rx, vy\*2+1, my+1, ry);

t[vx][vy] = t[vx][vy\*2] + t[vx][vy\*2+1];

}

}

void build\_x (int vx, int lx, int rx)

{

if (lx != rx)

{

int mx = (lx + rx) / 2;

build\_x (vx\*2, lx, mx);

build\_x (vx\*2+1, mx+1, rx);

}

build\_y (vx, lx, rx, 1, 1, c);

}

int sum\_y (int vx, int vy, int tly, int try\_, int ly, int ry)

{

if (ly > ry)

return 0;

if (ly == tly && try\_ == ry)

return t[vx][vy];

int tmy = (tly + try\_) / 2;

return sum\_y (vx, vy\*2, tly, tmy, ly, min(ry,tmy))

+ sum\_y (vx, vy\*2+1, tmy+1, try\_, max(ly,tmy+1), ry);

}

int sum\_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry)

{

if (lx > rx)

return 0;

if (lx == tlx && trx == rx)

return sum\_y (vx, 1, 1, c, ly, ry);

int tmx = (tlx + trx) / 2;

return sum\_x (vx\*2, tlx, tmx, lx, min(rx,tmx), ly, ry)

+ sum\_x (vx\*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry);

}

//For RMQ

void build\_y\_mn (int vx, int lx, int rx, int vy, int ly, int ry)

{

if (ly == ry)

if (lx == rx)

tmn[vx][vy] = a[lx][ly];

else

tmn[vx][vy] = max(tmn[vx\*2][vy] , tmn[vx\*2+1][vy]);

else

{

int my = (ly + ry) / 2;

build\_y\_mn (vx, lx, rx, vy\*2, ly, my);

build\_y\_mn (vx, lx, rx, vy\*2+1, my+1, ry);

tmn[vx][vy] = max(tmn[vx][vy\*2] , tmn[vx][vy\*2+1]);

}

}

void build\_x\_mn (int vx, int lx, int rx)

{

if (lx != rx)

{

int mx = (lx + rx) / 2;

build\_x\_mn (vx\*2, lx, mx);

build\_x\_mn (vx\*2+1, mx+1, rx);

}

build\_y\_mn (vx, lx, rx, 1, 1, c);

}

int min\_y (int vx, int vy, int tly, int try\_, int ly, int ry)

{

if (ly > ry)

return -INF;

if (ly == tly && try\_ == ry)

return tmn[vx][vy];

int tmy = (tly + try\_) / 2;

return max(min\_y (vx, vy\*2, tly, tmy, ly, min(ry,tmy))

, min\_y (vx, vy\*2+1, tmy+1, try\_, max(ly,tmy+1), ry));

}

int min\_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry)

{

if (lx > rx)

return -INF;

if (lx == tlx && trx == rx)

return min\_y (vx, 1, 1, c, ly, ry);

int tmx = (tlx + trx) / 2;

return max(min\_x (vx\*2, tlx, tmx, lx, min(rx,tmx), ly, ry)

, min\_x (vx\*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry));

}

//Usage

build\_x(1,1,r);

build\_x\_mn(1,1,r);

cur\_sum = sum\_x(1,1,r,i,i+a-1,j,j+b-1);

cur\_min = min\_x(1,1,r,i,i+a-1,j,j+b-1);

//2D Sparse Table

//0 based indexes for everything

//2d matrix 0 based row columns

inline int max(int a,int b)

{

if(a>b)

return a;

return b;

}

inline int max(int a,int b,int c,int d)

{

if(a>=b && a>=c && a>=d)

return a;

else if(b>=a && b>=c && b>=d)

return b;

else if(c>=a && c>=b && c>=d)

return c;

return d;

}

inline int min(int a,int b)

{

if(a>b)

return b;

return a;

}

int n,m; //dimension of the original matrix

int M[1002][1002][11][11]; //sparse table

int matrix[1005][1005]; // contains the original 2d matrix

int cum\_matrix[1005][1005];

void sparse\_table\_init()

{

for (int i = 0 ; (1<<i) <= n; i += 1)

{

for(int j = 0 ; (1<<j) <= m ; j += 1)

{

for (int x = 0 ; x + (1<<i) -1 < n; x+= 1)

{

for (int y = 0 ; y + (1<<j) -1 < m; y+= 1)

{

if (i == 0 and j == 0)

M[x][y][i][j] = matrix[x][y]; // store x, y

else if (i == 0)

M[x][y][i][j] = max(M[x][y][i][j-1], M[x][y+(1<<(j-1))][i][j-1]);

else if (j == 0)

M[x][y][i][j] = max(M[x][y][i-1][j], M[x+ (1<<(i-1))][y][i-1][j]);

else

M[x][y][i][j] = max(M[x][y][i-1][j-1], M[x + (1<<(i-1))][y][i-1][j-1], M[x][y+(1<<(j-1))][i-1][j-1], M[x + (1<<(i-1))][y+(1<<(j-1))][i-1][j-1]);

}

}

}

}

}

inline int clz(int N) {

return N ? 32 - \_\_builtin\_clz(N) : -INF;

}

inline int max\_rn(int x, int y, int x1, int y1)

{

int k = clz(x1 - x + 1) - 1; //O(log2(N))

int l = clz(y1 - y + 1) - 1; //O(log2(N))

int ans = max(M[x][y][k][l], M[x1 - (1<<k) + 1][y][k][l], M[x][y1 - (1<<l) + 1][k][l], M[x1 - (1<<k) + 1][y1 - (1<<l) + 1][k][l]);

return ans;

}

//Bit Manipulation

bool Check\_ON(int mask,int pos) //Check if pos th bit (from right) of mask is ON

{

if( (mask & (1<<pos) ) == 0 )return false;

return true;

}

int SET(int mask,int pos) //Save the returned mask into some var //Turn on pos th bit in mask

{

return (mask | (1<<pos));

}

int RESET(int mask,int pos) //Save the returned mask into some var //Turn off pos th bit in mask

{

return (mask & ~(1<<pos));

}

int FLIP(int mask,int pos) //Save the returned mask into some var //Toggle/Flip pos th bit in mask

{

return (mask ^ (1<<pos));

}

int LSB(int mask) // The actual LSB mask

{

return (mask & (-mask));

}

int LSB\_pos(int mask) // 0 based position

{

int mask\_2 = (mask & (-mask));

for(int pos = 0;pos<=15;pos++)

{

if(Check\_ON(mask\_2,pos))

return pos;

}

return -1;//

}

int ON\_Bits(int mask)

{

return \_\_builtin\_popcount(mask);

}

inline int clz(int N) { // O(1) way to calculate log2(X) (int s only)

return N ? 32 - \_\_builtin\_clz(N) : -INF;

}

Taking integer input from a single line string

char buff[100000];

gets(buff);

stringstream ss(buff);

int i = 1;

while(ss>>Arr[i++]); // The string is copied to Arr

N = i-2;

Bars and stars

1) Sum of k tuples adding upto N (all positive)

N-1 C K-1

2) Sum of k tuples adding upto N (all non-negative)

N+K-1 C N or N+K-1 C K-1

3)Dearrangement Formula :

d(1) = 0 d(2) = 0;

d(n) = (n-1)\*( d(n-1) + d(n-2))

// To compute x^y under modulo m

ll power(ll base,ll pw,ll mod)

{

if (pw == 0)

return 1;

ll p12 = power(base, pw/2, mod) % mod;

p12 = (p12 \* p12) % mod;

if(pw%2==0)

return p12;

else

return ((base%mod)\*(p12))%mod;

}

ll modInverse(ll a, ll m)

{

return power(a, m-2, m);

}

//Using Extended Euclid

ll gcdExtended(ll a, ll b, ll \*x, ll \*y)

{

// Base Case

if (a == 0)

{

\*x = 0, \*y = 1;

return b;

}

ll x1, y1; // To store results of recursive call

ll gcd = gcdExtended(b%a, a, &x1, &y1);

// Update x and y using results of recursive

// call

\*x = y1 - (b/a) \* x1;

\*y = x1;

return gcd;

}

ll modinv(ll a, ll m)

{

ll x, y;

ll g = gcdExtended(a, m, &x, &y);

// m is added to handle negative x

ll res = (x%m + m) % m;

return res;

}

char strt[1000009];

char strp[1000009];

int lps[1000009];

void lpscalc()

{

int j = 0; // length of the previous longest prefix suffix

int i;

lps[0] = 0; // lps[0] is always 0

i = 1;

int plen = strlen(strp);

// the loop calculates lps[i] for i = 1 to M-1

while (i < plen)

{

if (strp[j] == strp[i])

{

lps[i] = j+1;

i +=1;

j+=1;

}

else // (pat[i] != pat[j])

{

if (j != 0)

{

j = lps[j-1];

}

else // if (j == 0)

{

lps[i] = 0;

i++;

}

}

}

}

int nummatch()

{

int cnt = 0;

int pat\_index = 0, text\_index = 0;

int plen = strlen(strp);

int tlen = strlen(strt);

if(plen == 0)

{

return 0;

}

while(text\_index < tlen)

{

// if characters match, look for next character match

if(strp[(pat\_index)] == strt[(text\_index)])

{

pat\_index++;

text\_index++;

// indicates that complete pattern has matched

if(pat\_index == plen)

{

cnt++;

pat\_index = lps[pat\_index-1];

}

}

// if the characters do not match, don't go back in the text. Just adjust the pattern\_index

else

{

if(pat\_index != 0)

{

pat\_index = lps[pat\_index-1];

}

else

{

text\_index++;

}

}

}

return cnt;

}

package root;

import java.io.BufferedOutputStream;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

//import java.math.BigInteger;

import java.util.StringTokenizer;

public class NS\_1\_69A {

public static void main(String[] args) {

fastScanner fs = new fastScanner();

out = new PrintWriter(new BufferedOutputStream(System.out));

// Usage-------------------------------------

/\*\*

int n = fs.nextInt(); // read input as integer

long k = fs.nextLong(); // read input as long

double d = fs.nextDouble(); // read input as double

String str = fs.next(); // read input as String

String s = fs.nextLine(); // read whole line as String

out.println(); // print from PrintWriter

\*\*/

// Stop writing your solution here. -------------------------------------

out.close();

}

//-----------PrintWriter for faster output---------------------------------

public static PrintWriter out;

//-----------FastScanner class for faster input----------

public static class fastScanner {

BufferedReader BuffRead;

StringTokenizer StrToc;

public fastScanner() {

BuffRead = new BufferedReader(new InputStreamReader(System.in));

}

String next() {

while (StrToc == null || !StrToc.hasMoreElements()) {

try {

StrToc = new StringTokenizer(BuffRead.readLine());

} catch (IOException e) {

e.printStackTrace();

}

}

return StrToc.nextToken();

}

int nextInt() {

return Integer.parseInt(next());

}

long nextLong() {

return Long.parseLong(next());

}

double nextDouble() {

return Double.parseDouble(next());

}

/\*

BigInteger nextBigInteger(){

return new BigInteger(next().toString());

}

\*/

String nextLine(){

String str = "";

try {

str = BuffRead.readLine();

} catch (IOException e) {

e.printStackTrace();

}

return str;

}

}

//--------------------------------------------------------

}

#include <bits/stdc++.h>

#define loop(i,s,e) for(int i = s;i<=e;i++) //including end point

#define pb(a) push\_back(a)

#define sqr(x) ((x)\*(x))

#define CIN ios\_base::sync\_with\_stdio(0); cin.tie(0);

#define ll long long

#define ull unsigned long long

#define SZ(a) int(a.size())

#define read() freopen("input.txt", "r", stdin)

#define write() freopen("output.txt", "w", stdout)

#define ms(a,b) memset(a, b, sizeof(a))

#define all(v) v.begin(), v.end()

#define PI acos(-1.0)

#define pf printf

#define sfi(a) scanf("%d",&a);

#define sfii(a,b) scanf("%d %d",&a,&b);

#define sfl(a) scanf("%lld",&a);

#define sfll(a,b) scanf("%lld %lld",&a,&b);

#define sful(a) scanf("%llu",&a);

#define sfulul(a,b) scanf("%llu %llu",&a,&b);

#define sful2(a,b) scanf("%llu %llu",&a,&b); // A little different

#define sfc(a) scanf("%c",&a);

#define sfs(a) scanf("%s",a);

#define getl(s) getline(cin,s);

#define mp make\_pair

#define paii pair<int, int>

#define padd pair<dd, dd>

#define pall pair<ll, ll>

#define vi vector<int>

#define vll vector<ll>

#define mii map<int,int>

#define mlli map<ll,int>

#define mib map<int,bool>

#define fs first

#define sc second

#define CASE(t) printf("Case %d: ",++t) // t initialized 0

#define cCASE(t) cout<<"Case "<<++t<<": ";

#define D(v,status) cout<<status<<" "<<v<<endl;

#define INF 1000000000 //10e9

#define EPS 1e-9

#define flc fflush(stdout); //For interactive programs , flush while using pf (that's why \_\_c )

#define CONTEST 1

using namespace std;

//CONTEST MATRIX LIB

#define GB 0

#define dim 4

#define mat vector<vector<int>>

mat GBv;

int idmat[] = //Each row

{

1,0,1,1 ,

1,0,0,0 ,

0,1,0,0 ,

0,0,0,1

};

mat assImat(int arr[]) // assign identity matrix

{

mat X;

int arridx = 0;

vi rows;

if(!rows.empty())

{

rows.clear();

}

loop(r,0,dim-1)

{

loop(c,0,dim-1)

{

rows.pb(arr[arridx]);

arridx++;

}

X.pb(rows);

rows.clear();

}

return X;

}

mat matmul(mat A,mat B,int ra,int ca,int rb,int cb)

{

if(ca!=rb)

{

cout<<"ERR dim"<<endl;

return GBv;

}

mat res;

vi rows;

loop(amr,0,ra-1) //ans matrix row

{

loop(amc,0,rb-1)

{

int rowi = 0;

loop(crc,0,ca-1) //common row column

{

rowi+=A[amr][crc]\*B[crc][amc];

}

rows.pb(rowi);

}

res.pb(rows);

rows.clear();

}

return res;

}

mat expo(mat A, int row,int col,int p)

{

if(p==1)

return A;

else if(p==2)

{

mat res = matmul(A,A,row,col,row,

col);

return res;

}

else if(p%2==0)

{

mat halfp = expo(A,row,col,p/2);

mat res = matmul(halfp,halfp,

row,col,row,col);

return res;

}

else if(p%2==1)

{

mat halfp = expo(A,row,col,p/2);

mat resp = matmul(halfp,halfp,

row,col,row,col);

mat finres = matmul(resp,A,

row,col,row,col);

return finres;

}

}

void showmat(mat A,int row,int col)

{

loop(r,0,row-1)

{

loop(c,0,col-1)

cout<<A[r][c]<<" ";

cout<<endl;

}

}

int main()

{

mat TT = assImat(idmat);

showmat(TT,dim,dim);

mat ans = matmul(TT,TT,dim,dim,dim,dim);

cout<<"----------"<<endl;

showmat(ans,dim,dim);

mat ans2 = expo(TT,dim,dim,2);

cout<<"----------"<<endl;

showmat(ans2,dim,dim);

return 0;

}

int left[max],right[max],vis[mx];

//left[x] e rekhechi left set er x tomo node er shathe kar matching korechi

//zodi left[x]=-1 tahole ekhono karo shathe matching korate parini

vi adj[max];

bool kuhn(int u)

{

//Idea of kuhn function :

/\*

Initially karo shathe karo matching hoy ni . cnt = 0 . shob left[x] = -1 , shob right[x] = -1;

ekhon ami shob gulo left node er shathe kno 1 ta right node er matching korte chai.(tae bpm function e m ta left node er upori loop chaliyechi)

Ekhon kuhn function e ami oi node theke zeshob node e zaoa zay shegulate zacchi ebong zokhoni ekta possible matching pacchi, shei 2 ta match koriye left right update kore dicchi(ekhon ar era -1 nei).

ekhon matching 2 vabe ghotate pari ami,

1) connected kno ekta right node ekhono khali ache (-1) tahole ami easily eder matching koriye dite pari.

2) ami age kno ekta vul (non-optimal) decision nisilam zokhon left er matching koriyechi , orthat amar right node ta ekhon ze left node tar shathe matching koriyechi,hoyto oi left node take ami onno arekta right node er shathe matching korate partam ete amar matching 1 ta barto. eta korar jonno ami amar current right node visited kore dilam(porer bar ar ete zabo na karon er shathe already matching koriye felechi). tarpor ami ze left node er shathe matching koriyechi otake abar kuhn function e pathabo zodi amar current right node(occupied) chara onno karo shathe eke matching korano zeto.

zodi zay, tahole ami abar ekta matching korate parchi.

\*/

loop(x,0,SZ(adj[u])-1)

{

int v = adj[u][x];

if(vis[v]) continue;

vis[v] = 1;

if(right[v]==-1 || kuhn(right[v]))

{

right[v]=u;

left[u]=v;

return true;

}

}

return false;

}

int bpm()

{

ms(left,-1);

ms(right,-1);

int cnt = 0;

loop(x,1,m)

{

ms(vis,0);

if(kuhn(i))

cnt++:

}

return cnt;

}

//Miller Robin

#include<bits/stdc++.h>

#define ll long long int

#define mod 1000000007

#define MAX 10000007

using namespace std;

ll mulmod(ll a , ll b , ll mo)

{

ll q = ((long double) a \* (long double) b / (long double) mo);

ll res = a \* b - mo \* q;

return ((res % mo) + mo) % mo;

}

/\*

ll mulmod(ll a,ll b,ll c)

{

///this function calculates (a\*b)%c taking into account that a\*b might overflow

ll x = 0,y=a%c;

while(b > 0)

{

if(b%2 == 1)

{

x = (x+y)%c;

}

y = (y\*2)%c;

b /= 2;

}

return x%c;

}

\*/

ll bigmod (ll a, ll b, ll c)

{

ll res = 1;

a=a%c;

while (b > 0)

{

if (b % 2 == 1)

{

res=mulmod(res,a,c);

}

a=mulmod(a,a,c);

b=b/2;

}

return res;

}

bool miller(ll a, ll d, ll p)

{

ll x = bigmod(a,d,p);

if(x == 1 || x == p - 1)

return true;

while(d != p - 1)

{

x=mulmod(x,x,p);

d \*= 2;

if(x == 1)

{

return false;

}

if(x == p - 1)

{

return true;

}

}

return false;

}

bool isPrimes(ll p)

{

if(p<2)

{

return false;

}

if(p==2)

return true;

if(p!=2 && p%2==0)

{

return false;

}

ll d=p-1;

while(d%2==0)

d=d/2;

for(ll i=1; i<20; i++)

{

ll a=abs(rand()%(p-2))+2;

if(!miller(a,d,p))

return false;

}

return true;

}

int main()

{

ll t,n,q,i,j,ans,people,y,x,f,k;

scanf("%lld",&t);

while(t--)

{

scanf("%lld",&n);

for(i=n-1;; i--)

{

if(isPrimes(i))

{

printf("%lld\n",i);

break;

}

}

}

}

//Bitwise Sieve

#define mx 2147483700

int prm[(mx/32)+5];

bool Check(int N,int pos){return (bool)(N & (1<<pos));}

int Set(int N,int pos){ return N=N | (1<<pos) ;}

void BWsieve(int N)

{

int i, j, sqrtN;

sqrtN = int( sqrt( N ) );

for( i = 3; i <= sqrtN; i += 2 )

{

if( Check(prm[i>>5],i&31)==0)

{

for( j = i\*i; j <= N; j += (i<<1) )

{

prm[j>>5]=Set(prm[j>>5],j & 31) ;

}

}

}

}

Usage :

input

if( input is even ) Not prime

if( input is odd )

{

if( Check(status[input>>5],input&31 ) == 0 ) Prime

else

Not Prime

}

bool isprime(int input)

{

if(input&1)

{

if( Check(prm[input>>5],input&31 ) == 0 )

return true;

else

return false;

}

else

{

return false;

}

}

//Binary GCD

int gcd(int a, int b)

{

while(b) b ^= a ^= b ^= a %= b;

return a;

}

//EXTENDED EUCLID

int xGCD(int a, int b, int &x, int &y) {

if(b == 0) {

x = 1;

y = 0;

return a;

}

int x1, y1, gcd = xGCD(b, a % b, x1, y1);

x = y1;

y = x1 - (a / b) \* y1;

return gcd;

}

#include <iostream>

#include <float.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

// A structure to represent a Point in 2D plane

struct Point

{

int x, y;

};

/\* Following two functions are needed for library function qsort().

Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/

// Needed to sort array of points according to X coordinate

int compareX(const void\* a, const void\* b)

{

Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

return (p1->x - p2->x);

}

// Needed to sort array of points according to Y coordinate

int compareY(const void\* a, const void\* b)

{

Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

return (p1->y - p2->y);

}

// A utility function to find the distance between two points

float dist(Point p1, Point p2)

{

return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +

(p1.y - p2.y)\*(p1.y - p2.y)

);

}

// A Brute Force method to return the smallest distance between two points

// in P[] of size n

float bruteForce(Point P[], int n)

{

float min = FLT\_MAX;

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

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

if (dist(P[i], P[j]) < min)

min = dist(P[i], P[j]);

return min;

}

// A utility function to find minimum of two float values

float min(float x, float y)

{

return (x < y)? x : y;

}

// A utility function to find the distance beween the closest points of

// strip of given size. All points in strip[] are sorted accordint to

// y coordinate. They all have an upper bound on minimum distance as d.

// Note that this method seems to be a O(n^2) method, but it's a O(n)

// method as the inner loop runs at most 6 times

float stripClosest(Point strip[], int size, float d)

{

float min = d; // Initialize the minimum distance as d

// Pick all points one by one and try the next points till the difference

// between y coordinates is smaller than d.

// This is a proven fact that this loop runs at most 6 times

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

for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)

if (dist(strip[i],strip[j]) < min)

min = dist(strip[i], strip[j]);

return min;

}

// A recursive function to find the smallest distance. The array Px contains

// all points sorted according to x coordinates and Py contains all points

// sorted according to y coordinates

float closestUtil(Point Px[], Point Py[], int n)

{

// If there are 2 or 3 points, then use brute force

if (n <= 3)

return bruteForce(Px, n);

// Find the middle point

int mid = n/2;

Point midPoint = Px[mid]

// Divide points in y sorted array around the vertical line.

// Assumption: All x coordinates are distinct.

Point Pyl[mid+1]; // y sorted points on left of vertical line

Point Pyr[n-mid-1]; // y sorted points on right of vertical line

int li = 0, ri = 0; // indexes of left and right subarrays

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

{

if (Py[i].x <= midPoint.x)

Pyl[li++] = Py[i];

else

Pyr[ri++] = Py[i];

}

// Consider the vertical line passing through the middle point

// calculate the smallest distance dl on left of middle point and

// dr on right side

float dl = closestUtil(Px, Pyl, mid);

float dr = closestUtil(Px + mid, Pyr, n-mid);

// Find the smaller of two distances

float d = min(dl, dr);

// Build an array strip[] that contains points close (closer than d)

// to the line passing through the middle point

Point strip[n];

int j = 0;

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

if (abs(Py[i].x - midPoint.x) < d)

strip[j] = Py[i], j++;

// Find the closest points in strip. Return the minimum of d and closest

// distance is strip[]

return min(d, stripClosest(strip, j, d) );

}

// The main functin that finds the smallest distance

// This method mainly uses closestUtil()

float closest(Point P[], int n)

{

Point Px[n];

Point Py[n];

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

{

Px[i] = P[i];

Py[i] = P[i];

}

qsort(Px, n, sizeof(Point), compareX);

qsort(Py, n, sizeof(Point), compareY);

// Use recursive function closestUtil() to find the smallest distance

return closestUtil(Px, Py, n);

}

// Driver program to test above functions

int main()

{

Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};

int n = sizeof(P) / sizeof(P[0]);

cout << "The smallest distance is " << closest(P, n);

return 0;

}

// Implementation of Andrew's monotone chain 2D convex hull algorithm.

// Asymptotic complexity: O(n log n).

// Practical performance: 0.5-1.0 seconds for n=1000000 on a 1GHz machine.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

typedef double coord\_t; // coordinate type

typedef double coord2\_t; // must be big enough to hold 2\*max(|coordinate|)^2

struct Point {

coord\_t x, y;

Point()

{

this->x = 0.0000000f;

this->y = 0.0000000f;

}

Point(coord\_t x,coord\_t y)

{

this->x = x;

this->y = y;

}

bool operator <(const Point &p) const {

return x < p.x || (x == p.x && y < p.y);

}

};

// 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.

// Returns a positive value, if OAB makes a counter-clockwise turn,

// negative for clockwise turn, and zero if the points are collinear.

coord2\_t cross(const Point &O, const Point &A, const Point &B)

{

return (long)(A.x - O.x) \* (B.y - O.y) - (long)(A.y - O.y) \* (B.x - O.x);

}

// Returns a list of points on the convex hull in counter-clockwise order.

// Note: the last point in the returned list is the same as the first one.

vector<Point> convex\_hull(vector<Point> P)

{

int n = P.size(), k = 0;

vector<Point> H(2\*n);

// Sort points lexicographically

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

// Build lower hull

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

while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;

H[k++] = P[i];

}

// Build upper hull

for (int i = n-2, t = k+1; i >= 0; i--) {

while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;

H[k++] = P[i];

}

H.resize(k);

return H;

}

int main()

{

vector<Point>in;

Point p(-3.4,50);

Point p1(33.4,51);

Point p2(30.4,15);

Point p3(31.4,45);

Point p4(3.4,55);

Point p5(-33.4,15);

Point p6(-31.4,75);

in.push\_back(p);

in.push\_back(p1);

in.push\_back(p2);

in.push\_back(p3);

in.push\_back(p4);

in.push\_back(p5);

in.push\_back(p6);

vector<Point>out = convex\_hull(in);

for(int a=0;a<out.size();a++)

{

Point pp = out[a];

cout<<pp.x<<" "<<pp.y<<endl;

}

}

#include <algorithm>

#include <cstdio>

#include <cmath>

#include <vector>

using namespace std;

#define INF 1e9

#define EPS 1e-9

#define PI acos(-1.0) // important constant; alternative #define PI (2.0 \* acos(0.0))

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

// struct point\_i { int x, y; }; // basic raw form, minimalist mode

struct point\_i { int x, y; // whenever possible, work with point\_i

point\_i() { x = y = 0; } // default constructor

point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} }; // user-defined

struct point { double x, y; // only used if more precision is needed

point() { x = y = 0.0; } // default constructor

point(double \_x, double \_y) : x(\_x), y(\_y) {} // user-defined

bool operator < (point other) const { // override less than operator

if (fabs(x - other.x) > EPS) // useful for sorting

return x < other.x; // first criteria , by x-coordinate

return y < other.y; } // second criteria, by y-coordinate

// use EPS (1e-9) when testing equality of two floating points

bool operator == (point other) const {

return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) { // Euclidean distance

// hypot(dx, dy) returns sqrt(dx \* dx + dy \* dy)

return hypot(p1.x - p2.x, p1.y - p2.y); } // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)

point rotate(point p, double theta) {

double rad = DEG\_to\_RAD(theta); // multiply theta with PI / 180.0

return point(p.x \* cos(rad) - p.y \* sin(rad),

p.x \* sin(rad) + p.y \* cos(rad)); }

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

// the answer is stored in the third parameter (pass by reference)

void pointsToLine(point p1, point p2, line &l) {

if (fabs(p1.x - p2.x) < EPS) { // vertical line is fine

l.a = 1.0; l.b = 0.0; l.c = -p1.x; // default values

} else {

l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);

l.b = 1.0; // IMPORTANT: we fix the value of b to 1.0

l.c = -(double)(l.a \* p1.x) - p1.y;

} }

// not needed since we will use the more robust form: ax + by + c = 0 (see above)

struct line2 { double m, c; }; // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {

if (abs(p1.x - p2.x) < EPS) { // special case: vertical line

l.m = INF; // l contains m = INF and c = x\_value

l.c = p1.x; // to denote vertical line x = x\_value

return 0; // we need this return variable to differentiate result

}

else {

l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);

l.c = p1.y - l.m \* p1.x;

return 1; // l contains m and c of the line equation y = mx + c

} }

bool areParallel(line l1, line l2) { // check coefficients a & b

return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) { // also check coefficient c

return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect

bool areIntersect(line l1, line l2, point &p) {

if (areParallel(l1, l2)) return false; // no intersection

// solve system of 2 linear algebraic equations with 2 unknowns

p.x = (l2.b \* l1.c - l1.b \* l2.c) / (l2.a \* l1.b - l1.a \* l2.b);

// special case: test for vertical line to avoid division by zero

if (fabs(l1.b) > EPS) p.y = -(l1.a \* p.x + l1.c);

else p.y = -(l2.a \* p.x + l2.c);

return true; }

struct vec { double x, y; // name: `vec' is different from STL vector

vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) { // convert 2 points to vector a->b

return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) { // nonnegative s = [<1 .. 1 .. >1]

return vec(v.x \* s, v.y \* s); } // shorter.same.longer

point translate(point p, vec v) { // translate p according to v

return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line

void pointSlopeToLine(point p, double m, line &l) {

l.a = -m; // always -m

l.b = 1; // always 1

l.c = -((l.a \* p.x) + (l.b \* p.y)); } // compute this

void closestPoint(line l, point p, point &ans) {

line perpendicular; // perpendicular to l and pass through p

if (fabs(l.b) < EPS) { // special case 1: vertical line

ans.x = -(l.c); ans.y = p.y; return; }

if (fabs(l.a) < EPS) { // special case 2: horizontal line

ans.x = p.x; ans.y = -(l.c); return; }

pointSlopeToLine(p, 1 / l.a, perpendicular); // normal line

// intersect line l with this perpendicular line

// the intersection point is the closest point

areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line

void reflectionPoint(line l, point p, point &ans) {

point b;

closestPoint(l, p, b); // similar to distToLine

vec v = toVec(p, b); // create a vector

ans = translate(translate(p, v), v); } // translate p twice

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

double norm\_sq(vec v) { return v.x \* v.x + v.y \* v.y; }

// returns the distance from p to the line defined by

// two points a and b (a and b must be different)

// the closest point is stored in the 4th parameter (byref)

double distToLine(point p, point a, point b, point &c) {

// formula: c = a + u \* ab

vec ap = toVec(a, p), ab = toVec(a, b);

double u = dot(ap, ab) / norm\_sq(ab);

c = translate(a, scale(ab, u)); // translate a to c

return dist(p, c); } // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by

// two points a and b (still OK if a == b)

// the closest point is stored in the 4th parameter (byref)

double distToLineSegment(point p, point a, point b, point &c) {

vec ap = toVec(a, p), ab = toVec(a, b);

double u = dot(ap, ab) / norm\_sq(ab);

if (u < 0.0) { c = point(a.x, a.y); // closer to a

return dist(p, a); } // Euclidean distance between p and a

if (u > 1.0) { c = point(b.x, b.y); // closer to b

return dist(p, b); } // Euclidean distance between p and b

return distToLine(p, a, b, c); } // run distToLine as above

double angle(point a, point o, point b) { // returns angle aob in rad

vec oa = toVec(o, a), ob = toVec(o, b);

return acos(dot(oa, ob) / sqrt(norm\_sq(oa) \* norm\_sq(ob))); }

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

//// another variant

//int area2(point p, point q, point r) { // returns 'twice' the area of this triangle A-B-c

// return p.x \* q.y - p.y \* q.x +

// q.x \* r.y - q.y \* r.x +

// r.x \* p.y - r.y \* p.x;

//}

// note: to accept collinear points, we have to change the `> 0'

// returns true if point r is on the left side of line pq

bool ccw(point p, point q, point r) {

return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq

bool collinear(point p, point q, point r) {

return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int main() {

point P1, P2, P3(0, 1); // note that both P1 and P2 are (0.00, 0.00)

printf("%d\n", P1 == P2); // true

printf("%d\n", P1 == P3); // false

vector<point> P;

P.push\_back(point(2, 2));

P.push\_back(point(4, 3));

P.push\_back(point(2, 4));

P.push\_back(point(6, 6));

P.push\_back(point(2, 6));

P.push\_back(point(6, 5));

// sorting points demo

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

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

printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

// rearrange the points as shown in the diagram below

P.clear();

P.push\_back(point(2, 2));

P.push\_back(point(4, 3));

P.push\_back(point(2, 4));

P.push\_back(point(6, 6));

P.push\_back(point(2, 6));

P.push\_back(point(6, 5));

P.push\_back(point(8, 6));

/\*

// the positions of these 7 points (0-based indexing)

6 P4 P3 P6

5 P5

4 P2

3 P1

2 P0

1

0 1 2 3 4 5 6 7 8

\*/

double d = dist(P[0], P[5]);

printf("Euclidean distance between P[0] and P[5] = %.2lf\n", d); // should be 5.000

// line equations

line l1, l2, l3, l4;

pointsToLine(P[0], P[1], l1);

printf("%.2lf \* x + %.2lf \* y + %.2lf = 0.00\n", l1.a, l1.b, l1.c); // should be -0.50 \* x + 1.00 \* y - 1.00 = 0.00

pointsToLine(P[0], P[2], l2); // a vertical line, not a problem in "ax + by + c = 0" representation

printf("%.2lf \* x + %.2lf \* y + %.2lf = 0.00\n", l2.a, l2.b, l2.c); // should be 1.00 \* x + 0.00 \* y - 2.00 = 0.00

// parallel, same, and line intersection tests

pointsToLine(P[2], P[3], l3);

printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2)); // no

printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3)); // yes, l1 (P[0]-P[1]) and l3 (P[2]-P[3]) are parallel

pointsToLine(P[2], P[4], l4);

printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); // no

printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); // yes, l2 (P[0]-P[2]) and l4 (P[2]-P[4]) are the same line (note, they are two different line segments, but same line)

point p12;

bool res = areIntersect(l1, l2, p12); // yes, l1 (P[0]-P[1]) and l2 (P[0]-P[2]) are intersect at (2.0, 2.0)

printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res, p12.x, p12.y);

// other distances

point ans;

d = distToLine(P[0], P[2], P[3], ans);

printf("Closest point from P[0] to line (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d);

closestPoint(l3, P[0], ans);

printf("Closest point from P[0] to line V2 (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

d = distToLineSegment(P[0], P[2], P[3], ans);

printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to A (or P[2]) = (2.00, 4.00)

d = distToLineSegment(P[1], P[2], P[3], ans);

printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to midway between AB = (3.20, 4.60)

d = distToLineSegment(P[6], P[2], P[3], ans);

printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to B (or P[3]) = (6.00, 6.00)

reflectionPoint(l4, P[1], ans);

printf("Reflection point from P[1] to line (P[2]-P[4]): (%.2lf, %.2lf)\n", ans.x, ans.y); // should be (0.00, 3.00)

printf("Angle P[0]-P[4]-P[3] = %.2lf\n", RAD\_to\_DEG(angle(P[0], P[4], P[3]))); // 90 degrees

printf("Angle P[0]-P[2]-P[1] = %.2lf\n", RAD\_to\_DEG(angle(P[0], P[2], P[1]))); // 63.43 degrees

printf("Angle P[4]-P[3]-P[6] = %.2lf\n", RAD\_to\_DEG(angle(P[4], P[3], P[6]))); // 180 degrees

printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0], P[2], P[3])); // no

printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0], P[3], P[2])); // yes

printf("P[0], P[2], P[3] are collinear? %d\n", collinear(P[0], P[2], P[3])); // no

printf("P[0], P[2], P[4] are collinear? %d\n", collinear(P[0], P[2], P[4])); // yes

point p(3, 7), q(11, 13), r(35, 30); // collinear if r(35, 31)

printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left" : "right"); // right

/\*

// the positions of these 6 points

E<-- 4

3 B D<--

2 A C

1

-4-3-2-1 0 1 2 3 4 5 6

-1

-2

F<-- -3

\*/

// translation

point A(2.0, 2.0);

point B(4.0, 3.0);

vec v = toVec(A, B); // imagine there is an arrow from A to B (see the diagram above)

point C(3.0, 2.0);

point D = translate(C, v); // D will be located in coordinate (3.0 + 2.0, 2.0 + 1.0) = (5.0, 3.0)

printf("D = (%.2lf, %.2lf)\n", D.x, D.y);

point E = translate(C, scale(v, 0.5)); // E will be located in coordinate (3.0 + 1/2 \* 2.0, 2.0 + 1/2 \* 1.0) = (4.0, 2.5)

printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

// rotation

printf("B = (%.2lf, %.2lf)\n", B.x, B.y); // B = (4.0, 3.0)

point F = rotate(B, 90); // rotate B by 90 degrees COUNTER clockwise, F = (-3.0, 4.0)

printf("F = (%.2lf, %.2lf)\n", F.x, F.y);

point G = rotate(B, 180); // rotate B by 180 degrees COUNTER clockwise, G = (-4.0, -3.0)

printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

return 0;

}

#include <algorithm>

#include <cstdio>

#include <cmath>

#include <stack>

#include <vector>

using namespace std;

#define EPS 1e-9

#define PI acos(-1.0)

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

struct point { double x, y; // only used if more precision is needed

point() { x = y = 0.0; } // default constructor

point(double \_x, double \_y) : x(\_x), y(\_y) {} // user-defined

bool operator == (point other) const {

return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

struct vec { double x, y; // name: `vec' is different from STL vector

vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) { // convert 2 points to vector a->b

return vec(b.x - a.x, b.y - a.y); }

double dist(point p1, point p2) { // Euclidean distance

return hypot(p1.x - p2.x, p1.y - p2.y); } // return double

// returns the perimeter, which is the sum of Euclidian distances

// of consecutive line segments (polygon edges)

double perimeter(const vector<point> &P) {

double result = 0.0;

for (int i = 0; i < (int)P.size()-1; i++) // remember that P[0] = P[n-1]

result += dist(P[i], P[i+1]);

return result; }

// returns the area, which is half the determinant

double area(const vector<point> &P) {

double result = 0.0, x1, y1, x2, y2;

for (int i = 0; i < (int)P.size()-1; i++) {

x1 = P[i].x; x2 = P[i+1].x;

y1 = P[i].y; y2 = P[i+1].y;

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

}

return fabs(result) / 2.0; }

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

double norm\_sq(vec v) { return v.x \* v.x + v.y \* v.y; }

double angle(point a, point o, point b) { // returns angle aob in rad

vec oa = toVec(o, a), ob = toVec(o, b);

return acos(dot(oa, ob) / sqrt(norm\_sq(oa) \* norm\_sq(ob))); }

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

// note: to accept collinear points, we have to change the `> 0'

// returns true if point r is on the left side of line pq

bool ccw(point p, point q, point r) {

return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq

bool collinear(point p, point q, point r) {

return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn while examining

// all the edges of the polygon one by one

bool isConvex(const vector<point> &P) {

int sz = (int)P.size();

if (sz <= 3) return false; // a point/sz=2 or a line/sz=3 is not convex

bool isLeft = ccw(P[0], P[1], P[2]); // remember one result

for (int i = 1; i < sz-1; i++) // then compare with the others

if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)

return false; // different sign -> this polygon is concave

return true; } // this polygon is convex

// returns true if point p is in either convex/concave polygon P

bool inPolygon(point pt, const vector<point> &P) {

if ((int)P.size() == 0) return false;

double sum = 0; // assume the first vertex is equal to the last vertex

for (int i = 0; i < (int)P.size()-1; i++) {

if (ccw(pt, P[i], P[i+1]))

sum += angle(P[i], pt, P[i+1]); // left turn/ccw

else sum -= angle(P[i], pt, P[i+1]); } // right turn/cw

return fabs(fabs(sum) - 2\*PI) < EPS; }

// line segment p-q intersect with line A-B.

point lineIntersectSeg(point p, point q, point A, point B) {

double a = B.y - A.y;

double b = A.x - B.x;

double c = B.x \* A.y - A.x \* B.y;

double u = fabs(a \* p.x + b \* p.y + c);

double v = fabs(a \* q.x + b \* q.y + c);

return point((p.x \* v + q.x \* u) / (u+v), (p.y \* v + q.y \* u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b

// (note: the last point must be the same as the first point)

vector<point> cutPolygon(point a, point b, const vector<point> &Q) {

vector<point> P;

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

double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;

if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));

if (left1 > -EPS) P.push\_back(Q[i]); // Q[i] is on the left of ab

if (left1 \* left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line ab

P.push\_back(lineIntersectSeg(Q[i], Q[i+1], a, b));

}

if (!P.empty() && !(P.back() == P.front()))

P.push\_back(P.front()); // make P's first point = P's last point

return P; }

point pivot;

bool angleCmp(point a, point b) { // angle-sorting function

if (collinear(pivot, a, b)) // special case

return dist(pivot, a) < dist(pivot, b); // check which one is closer

double d1x = a.x - pivot.x, d1y = a.y - pivot.y;

double d2x = b.x - pivot.x, d2y = b.y - pivot.y;

return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; } // compare two angles

vector<point> CH(vector<point> P) { // the content of P may be reshuffled

int i, j, n = (int)P.size();

if (n <= 3) {

if (!(P[0] == P[n-1])) P.push\_back(P[0]); // safeguard from corner case

return P; // special case, the CH is P itself

}

// first, find P0 = point with lowest Y and if tie: rightmost X

int P0 = 0;

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

if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))

P0 = i;

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

// second, sort points by angle w.r.t. pivot P0

pivot = P[0]; // use this global variable as reference

sort(++P.begin(), P.end(), angleCmp); // we do not sort P[0]

// third, the ccw tests

vector<point> S;

S.push\_back(P[n-1]); S.push\_back(P[0]); S.push\_back(P[1]); // initial S

i = 2; // then, we check the rest

while (i < n) { // note: N must be >= 3 for this method to work

j = (int)S.size()-1;

if (ccw(S[j-1], S[j], P[i])) S.push\_back(P[i++]); // left turn, accept

else S.pop\_back(); } // or pop the top of S until we have a left turn

return S; } // return the result

int main() {

// 6 points, entered in counter clockwise order, 0-based indexing

vector<point> P;

P.push\_back(point(1, 1));

P.push\_back(point(3, 3));

P.push\_back(point(9, 1));

P.push\_back(point(12, 4));

P.push\_back(point(9, 7));

P.push\_back(point(1, 7));

P.push\_back(P[0]); // loop back

printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // 31.64

printf("Area of polygon = %.2lf\n", area(P)); // 49.00

printf("Is convex = %d\n", isConvex(P)); // false (P1 is the culprit)

//// the positions of P6 and P7 w.r.t the polygon

//7 P5--------------P4

//6 | \

//5 | \

//4 | P7 P3

//3 | P1\_\_\_ /

//2 | / P6 \ \_\_\_ /

//1 P0 P2

//0 1 2 3 4 5 6 7 8 9 101112

point P6(3, 2); // outside this (concave) polygon

printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // false

point P7(3, 4); // inside this (concave) polygon

printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

// cutting the original polygon based on line P[2] -> P[4] (get the left side)

//7 P5--------------P4

//6 | | \

//5 | | \

//4 | | P3

//3 | P1\_\_\_ | /

//2 | / \ \_\_\_ | /

//1 P0 P2

//0 1 2 3 4 5 6 7 8 9 101112

// new polygon (notice the index are different now):

//7 P4--------------P3

//6 | |

//5 | |

//4 | |

//3 | P1\_\_\_ |

//2 | / \ \_\_\_ |

//1 P0 P2

//0 1 2 3 4 5 6 7 8 9

P = cutPolygon(P[2], P[4], P);

printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // smaller now 29.15

printf("Area of polygon = %.2lf\n", area(P)); // 40.00

// running convex hull of the resulting polygon (index changes again)

//7 P3--------------P2

//6 | |

//5 | |

//4 | P7 |

//3 | |

//2 | |

//1 P0--------------P1

//0 1 2 3 4 5 6 7 8 9

P = CH(P); // now this is a rectangle

printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // precisely 28.00

printf("Area of polygon = %.2lf\n", area(P)); // precisely 48.00

printf("Is convex = %d\n", isConvex(P)); // true

printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // true

printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

return 0;

}

#include <cstdio>

#include <cmath>

using namespace std;

#define EPS 1e-9

#define PI acos(-1.0)

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

struct point\_i { int x, y; // whenever possible, work with point\_i

point\_i() { x = y = 0; } // default constructor

point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} }; // constructor

struct point { double x, y; // only used if more precision is needed

point() { x = y = 0.0; } // default constructor

point(double \_x, double \_y) : x(\_x), y(\_y) {} }; // constructor

double dist(point p1, point p2) {

return hypot(p1.x - p2.x, p1.y - p2.y); }

double perimeter(double ab, double bc, double ca) {

return ab + bc + ca; }

double perimeter(point a, point b, point c) {

return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {

// Heron's formula, split sqrt(a \* b) into sqrt(a) \* sqrt(b); in implementation

double s = 0.5 \* perimeter(ab, bc, ca);

return sqrt(s) \* sqrt(s - ab) \* sqrt(s - bc) \* sqrt(s - ca); }

double area(point a, point b, point c) {

return area(dist(a, b), dist(b, c), dist(c, a)); }

//====================================================================

// from ch7\_01\_points\_lines

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

// the answer is stored in the third parameter (pass by reference)

void pointsToLine(point p1, point p2, line &l) {

if (fabs(p1.x - p2.x) < EPS) { // vertical line is fine

l.a = 1.0; l.b = 0.0; l.c = -p1.x; // default values

} else {

l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);

l.b = 1.0; // IMPORTANT: we fix the value of b to 1.0

l.c = -(double)(l.a \* p1.x) - p1.y;

} }

bool areParallel(line l1, line l2) { // check coefficient a + b

return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

// returns true (+ intersection point) if two lines are intersect

bool areIntersect(line l1, line l2, point &p) {

if (areParallel(l1, l2)) return false; // no intersection

// solve system of 2 linear algebraic equations with 2 unknowns

p.x = (l2.b \* l1.c - l1.b \* l2.c) / (l2.a \* l1.b - l1.a \* l2.b);

// special case: test for vertical line to avoid division by zero

if (fabs(l1.b) > EPS) p.y = -(l1.a \* p.x + l1.c);

else p.y = -(l2.a \* p.x + l2.c);

return true; }

struct vec { double x, y; // name: `vec' is different from STL vector

vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) { // convert 2 points to vector a->b

return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) { // nonnegative s = [<1 .. 1 .. >1]

return vec(v.x \* s, v.y \* s); } // shorter.same.longer

point translate(point p, vec v) { // translate p according to v

return point(p.x + v.x , p.y + v.y); }

//====================================================================

double rInCircle(double ab, double bc, double ca) {

return area(ab, bc, ca) / (0.5 \* perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {

return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written

// returns 1 if there is an inCircle center, returns 0 otherwise

// if this function returns 1, ctr will be the inCircle center

// and r is the same as rInCircle

int inCircle(point p1, point p2, point p3, point &ctr, double &r) {

r = rInCircle(p1, p2, p3);

if (fabs(r) < EPS) return 0; // no inCircle center

line l1, l2; // compute these two angle bisectors

double ratio = dist(p1, p2) / dist(p1, p3);

point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));

pointsToLine(p1, p, l1);

ratio = dist(p2, p1) / dist(p2, p3);

p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));

pointsToLine(p2, p, l2);

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

return 1; }

double rCircumCircle(double ab, double bc, double ca) {

return ab \* bc \* ca / (4.0 \* area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {

return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written

// returns 1 if there is a circumCenter center, returns 0 otherwise

// if this function returns 1, ctr will be the circumCircle center

// and r is the same as rCircumCircle

int circumCircle(point p1, point p2, point p3, point &ctr, double &r){

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

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

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

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

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

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

ctr.x = (d\*e - b\*f) / g;

ctr.y = (a\*f - c\*e) / g;

r = dist(p1, ctr); // r = distance from center to 1 of the 3 points

return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c

int inCircumCircle(point a, point b, point c, point d) {

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

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

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

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

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

(a.x - d.x) \* ((b.x - d.x) \* (b.x - d.x) + (b.y - d.y) \* (b.y - d.y)) \* (c.y - d.y) > 0 ? 1 : 0;

}

bool canFormTriangle(double a, double b, double c) {

return (a + b > c) && (a + c > b) && (b + c > a); }

int main() {

double base = 4.0, h = 3.0;

double A = 0.5 \* base \* h;

printf("Area = %.2lf\n", A);

point a; // a right triangle

point b(4.0, 0.0);

point c(4.0, 3.0);

double p = perimeter(a, b, c);

double s = 0.5 \* p;

A = area(a, b, c);

printf("Area = %.2lf\n", A); // must be the same as above

double r = rInCircle(a, b, c);

printf("R1 (radius of incircle) = %.2lf\n", r); // 1.00

point ctr;

int res = inCircle(a, b, c, ctr, r);

printf("R1 (radius of incircle) = %.2lf\n", r); // same, 1.00

printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y); // (3.00, 1.00)

printf("R2 (radius of circumcircle) = %.2lf\n", rCircumCircle(a, b, c)); // 2.50

res = circumCircle(a, b, c, ctr, r);

printf("R2 (radius of circumcircle) = %.2lf\n", r); // same, 2.50

printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y); // (2.00, 1.50)

point d(2.0, 1.0); // inside triangle and circumCircle

printf("d inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, d));

point e(2.0, 3.9); // outside the triangle but inside circumCircle

printf("e inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, e));

point f(2.0, -1.1); // slightly outside

printf("f inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, f));

// Law of Cosines

double ab = dist(a, b);

double bc = dist(b, c);

double ca = dist(c, a);

double alpha = RAD\_to\_DEG(acos((ca \* ca + ab \* ab - bc \* bc) / (2.0 \* ca \* ab)));

printf("alpha = %.2lf\n", alpha);

double beta = RAD\_to\_DEG(acos((ab \* ab + bc \* bc - ca \* ca) / (2.0 \* ab \* bc)));

printf("beta = %.2lf\n", beta);

double gamma = RAD\_to\_DEG(acos((bc \* bc + ca \* ca - ab \* ab) / (2.0 \* bc \* ca)));

printf("gamma = %.2lf\n", gamma);

// Law of Sines

printf("%.2lf == %.2lf == %.2lf\n", bc / sin(DEG\_to\_RAD(alpha)), ca / sin(DEG\_to\_RAD(beta)), ab / sin(DEG\_to\_RAD(gamma)));

// Phytagorean Theorem

printf("%.2lf^2 == %.2lf^2 + %.2lf^2\n", ca, ab, bc);

// Triangle Inequality

printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 5, canFormTriangle(3, 4, 5)); // yes

printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 7, canFormTriangle(3, 4, 7)); // no, actually straight line

printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 8, canFormTriangle(3, 4, 8)); // no

return 0;

}

#include <cstdio>

#include <cmath>

using namespace std;

#define INF 1e9

#define EPS 1e-9

#define PI acos(-1.0)

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

struct point\_i { int x, y; // whenever possible, work with point\_i

point\_i() { x = y = 0; } // default constructor

point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} }; // constructor

struct point { double x, y; // only used if more precision is needed

point() { x = y = 0.0; } // default constructor

point(double \_x, double \_y) : x(\_x), y(\_y) {} }; // constructor

int insideCircle(point\_i p, point\_i c, int r) { // all integer version

int dx = p.x - c.x, dy = p.y - c.y;

int Euc = dx \* dx + dy \* dy, rSq = r \* r; // all integer

return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside

bool circle2PtsRad(point p1, point p2, double r, point &c) {

double d2 = (p1.x - p2.x) \* (p1.x - p2.x) +

(p1.y - p2.y) \* (p1.y - p2.y);

double det = r \* r / d2 - 0.25;

if (det < 0.0) return false;

double h = sqrt(det);

c.x = (p1.x + p2.x) \* 0.5 + (p1.y - p2.y) \* h;

c.y = (p1.y + p2.y) \* 0.5 + (p2.x - p1.x) \* h;

return true; } // to get the other center, reverse p1 and p2

int main() {

// circle equation, inside, border, outside

point\_i pt(2, 2);

int r = 7;

point\_i inside(8, 2);

printf("%d\n", insideCircle(inside, pt, r)); // 0-inside

point\_i border(9, 2);

printf("%d\n", insideCircle(border, pt, r)); // 1-at border

point\_i outside(10, 2);

printf("%d\n", insideCircle(outside, pt, r)); // 2-outside

double d = 2 \* r;

printf("Diameter = %.2lf\n", d);

double c = PI \* d;

printf("Circumference (Perimeter) = %.2lf\n", c);

double A = PI \* r \* r;

printf("Area of circle = %.2lf\n", A);

printf("Length of arc (central angle = 60 degrees) = %.2lf\n", 60.0 / 360.0 \* c);

printf("Length of chord (central angle = 60 degrees) = %.2lf\n", sqrt((2 \* r \* r) \* (1 - cos(DEG\_to\_RAD(60.0)))));

printf("Area of sector (central angle = 60 degrees) = %.2lf\n", 60.0 / 360.0 \* A);

point p1;

point p2(0.0, -1.0);

point ans;

circle2PtsRad(p1, p2, 2.0, ans);

printf("One of the center is (%.2lf, %.2lf)\n", ans.x, ans.y);

circle2PtsRad(p2, p1, 2.0, ans); // we simply reverse p1 with p2

printf("The other center is (%.2lf, %.2lf)\n", ans.x, ans.y);

return 0;

}