A Short Note On Basic Computational Geometry

Masum Billal

October 15, 2012

Abstract

A point with integer coordinates is called a *lattice point*. A segment $L(A, B)$ means its starting point is A and ending point is B. The order doesn't matter unless we are dealing with vectors. We will denote by X the X-axis and by Y, the Y-axis(otherwise stated). We will use structures such as

```
struct Point{
    int x, y;
    //if all functions can be done be in integers somehow ↩
        without overflow
    //write all functions here such as
    bool operator < (Point p){
    //compares two points by their x and y values
        return (x < p.x) || ((x == p.x) && (y < p.y));
    }
};
```

for coding. This makes the code a lot easier to read and write. We access the x and y values by $P.x$ and $P.y$. And it can be assured that for this coding, not much knowledge about structure or class is required. But this may be boring sometimes.

1 Line & Points

1. (Polar Coordinate System) A point $P(x, y)$ in Cartesian coordinate system is $P(r, \theta)$ in polar coordinate system where r is the distance of P from origin $O(0, 0)$ and θ is the angle created by the point along with the positive $X-$axis. Then, we can find r and θ from x and y from the relations:

$$r = x^2 + y^2$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

The reverse is:

$$x = r\cos\theta$$

$$y = r\sin\theta$$

2. Distance between two points $A(x_1, y_1)$ and $B(x_2, y_2)$ is

$$AB = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

$$AB^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$$

```
int SquaredDist(const Point &p1, const Point &p2){
    int dx = (p1.x-p2.x);
    int dy = (p1.y-p2.y);
    return (dx*dx)+(dy*dy);
}

double Dist(const Point &p1, const Point &p2){
    return sqrt(1.0*SquaredDist(p1, p2));
}
```

Sometimes, we can use the square of distance instead of the distance itself if the conditions still holds the same and does not overflow. But the advantage is we don't need double comparison and can avoid precision error. Often, double precision becomes an issue of getting WA.

3. The distance of $P(x, y)$ from X is y and the distance from Y is x.

4. (Different forms of equation of line)

   (a) The most general equation, and convenient for coding.

   $$ax + by = c$$

   This is default form used, unless stated.

   (b) Slope form

   $$y = mx + c$$

   m is the slope of the line and c is the height from the origin along the Y-axis(when $x = 0$, $y = c$).

(c) Assume a segment does not cross the origin. Then it must cut X and Y. Let the cut be a and b for X and Y respectively. The equation now becomes

$$\frac{x}{a} + \frac{y}{b} = 1$$

Here a and b has significance for its sign. If a is positive, it means the line cut X on the positive side whose length is $abs(a)$, otherwise on the negative side. And the same for b with Y.

5. Slope of a line going through two points $A(x_1, y_1)$ and $B(x_2, y_2)$ is

$$m_{AB} = \frac{y_2 - y_1}{x_2 - x_1}$$

6. Perpendicular distance of a line $L(A, B)$ from a point $P(x, y)$,

$$d = \frac{ax + by - c}{\sqrt{a^2 + b^2}}$$

in the first form.

7. Equation of a line going through two points $L(A, B)$ where $A(x_1, y_1)$ and $B(x_2, y_2)$.

$$\frac{x - x_1}{x_1 - x_2} = \frac{y - y1}{y_1 - y_2}$$

Re-writing this in the first form,

$$x(y_2 - y_1) + y(x_1 - x_2) = x_1y_2 - x_2y_1$$

So, equating corresponding coefficients,

$$a = y_2 - y_1, b = x_1 - x_2, c = x_1y_2 - x_2y_1$$

8. Two lines are parallel if their slope are equal. So, it is enough to check if $a_1b_2 = a_2b_1$(why?) or not?

9. Two lines are perpendicular if the product of their slope is -1(why?). So, it's enough to check if $a_1b_2 + a_2b_1 = 0$ or not(again why? You can explain this too if you can do the previous).

10. (Intersection Of two lines) The intersection point of two lines can be found if they are not parallel. Say, the equations of two lines are

$$a_1x + b_1y = c_1$$

$$a_2x + b_2y = c_2$$

Then, using determinant, the solution of them is given by

$$x = \frac{b_2c_1 - b_1c_2}{a_1b_2 - a_2b_1}, y = \frac{c_2a_1 - c_1a_2}{a_1b_2 - a_2b_1}$$

Here, determinant coefficient of x, $Dx = b_2c_1 - b_1c_2$ and $Dy = c_2a_1 - c_1a_2$. And $D = a_1b_2 - a_2b_1$ which is the determinant of the coefficients. Then, the solution to the equations is

$$x = \frac{Dx}{D}, y = \frac{Dy}{D}$$

11. (Converting into equation from $L(A, B)$) We can build a class called *Line* containing all functions related.

```
struct point{
    double x, y;
};//used for double points

class Line {
    public:
    Point A, B;

    Line (Point P, Point Q){
        A = P, B = Q;
    }

    Point MidPoint(Point A, Point B){
    //returns the midpoint of the segment
        Point p;
        p.x = (A.x+B.x)>>1;
        p.y = (A.y+B.y)>>1;
        return p;
    }

    int a(){
        return B.y-A.y;//corresponding to y2-y1
    }

    int b(){
        return A.x-B.x;//corresponding to x1-x2
    }

    int c(){
        return (A.x*B.y)-(B.x*A.y);//corresponding to x1y2-x2y1
    }

    //Meaning: The equation of the line - ax+by = c

    //a_1*x+b_1*y=c_1
    //a_2*x+b_2*y=c_2
    //Solving using Determinant
    int Dt(Line l){
        int a1 = a();
        int a2 = l.a();
        int b1 = b();
        int b2 = l.b();
        return ((a1*b2)-(a2*b1));
    }

    int DX(Line l){
        return l.b()*c()-b()*l.c();
    }

    int DY(Line l){
        return l.c()*a()-l.a()*c();
    }

    point Intersection(Line l2){
        point p;
        double d = 1.0*Dt(l2);
        double dx = 1.0*DX(l2);
        double dy = 1.0*DY(l2);
        p.x = dx/d;
        p.y = dy/d;
        return p;
    }

    //Slope form i.e. y = mx+c
    double m(){//returns m
        double dy = 1.0*(B.y-A.y);
        double dx = 1.0*(B.x-A.x);
        return dy/dx;
    }
```

```
70
71      double slc(){//returns c
72          return (1.0*c())/(1.0*a());
73      }
74
75      double DistanceFromPoint(Point P){
76          //returns distance of this line from P
77          double m = a();
78          double n = b();
79          double t = c();
80          double x = 1.0*P.x;
81          double y = 1.0*P.y;
82          return (m*x+n*y-t)/sqrt(m*m+n*n);
83      }
84
85      bool IsParallel(Line l){
86          //if they are parallel
87          int res = Dt(l);
88          if (!res)   return true;
89          return false;
90      }
91
92      bool IsPerp(Line l){
93          //if they are perpendicular to each other
94          int res = a()*l.a()+b()*l.b();
95          if (!res)   return true;
96          return false;
97      }
98  };
```

Clearly, a, b and c are the coefficients and constant as stated above.

12. Area of a triangle with points $A(x_0, y_0), B(x_1, y_1), C(x_2, y_2)$ is

$$2 * Area = x_0(y_1 - y_2) + x_1(y_2 - y_0) + x_2(y_0 - y_1)$$

```
1  int TriArea (const Point &A, const Point &B, const Point &C){
2      //area of triangle ABC
3      return abs(A.x*(B.y-C.y)+B.x*(C.y-A.y)+C.x*(A.y-B.y));
4  }
```

Here x and y values are such that it does not overflow integers. And note that, we don't use double area dividing by 2 unless it's necessary. Avoid double as far as you can! So, if the checks can be validated without double comparison, that's better, like we are doing here. Also, if you need to return the area only, use absolute value. Otherwise, the sign of this area may matter a lot.

13. The formula above can be generalized in this way:

    A simple polygon $P(P[0], P[1], P[2], \ldots, P[n-1])$ has an area

$$2 * Area = x_0(y_1 - y_2) + x_1(y_2 - y_0) + \ldots + x_{n-1}(y_0 - y_1)$$

```
1  int PolArea_twice(Point P[], int n){
2      //returns the area of a polygon, n is the number of points
3      int A = 0;
4      for (int i = 0; i < n; i++){
5          int j = (i+1)%n;
6          A += (P[i].x*P[j].y)-(P[j].y*P[i].x);
7      }
8      A = abs(A);
9      return A;
10 }
```

We have to return its absolute value because this quantity can be negative whereas area is always positive. But being the area has its significance. If the area is less than 0 that means the points of the polygon were in clockwise order. Otherwise, they were in counter-clockwise order.

14. (Testing if a polygon has points in counterclockwise order). We need not find the area of the whole polygon. Since the triangle formed by the first 3 points will be also in the same order as the polygon, we can just test the area of that triangle. If you need to change the rotation, just reverse the array of P.

```
bool isCounterclockwise(Point P[]){
    return TriArea(P[0], P[1], P[2]) > 0;
}
```

We assume no 3 consecutive points are co-linear and the number of points of any polygon must be greater than or equal to 3. This can also determine if three points make a counterclockwise rotation or clockwise rotation, which we call **ccw** in short.

15. Why is the above procedure correct? Any line divides the whole plane in 3 parts. The line itself, the part below the line and above the line, see the figure. Now, say the equation of the line is

$$(x - x_1)(y_1 - y_2) - (x_1 - x_2)(y - y_1) = 0$$

This can be used as a discriminant for determining whether a point lies on the line, or above the line or below the line. Say, the point is $P(x_3, y_3)$. Set the point into the equation of the line:

$$D = (x_3 - x_1)(y_1 - y_2) - (x_1 - x_2)(y_3 - y_1)$$

If $D = 0$ it means we have the point on the line. If $D > 0$, it is above the line. If $D < 0$, it lies below the line. If noticed carefully, we can see that D is actually the area of the points ABP.

There is another approach if you can't sense this properly. In figure 1, slope of BC is greater than AB, and slope of BD is less than AB and the slope of BE is equal to slope of AB. For the first case,

$$m_{BC} > m_{AB}$$

Setting the expression for slope,

$$\frac{y_3 - y_2}{x_3 - x_2} > \frac{y_1 - y_2}{x_1 - x_2}$$

Re-writing,

$$(x_3 - x_1)(y_1 - y_2) - (x_1 - x_2)(y_3 - y_1) > 0$$

again the same expression. And so the same for the two cases latter too.

16. How do we test if 3 points are co-linear? If some points are co-linear, then they form a polygon with area 0. Here is the special case for 3 points.

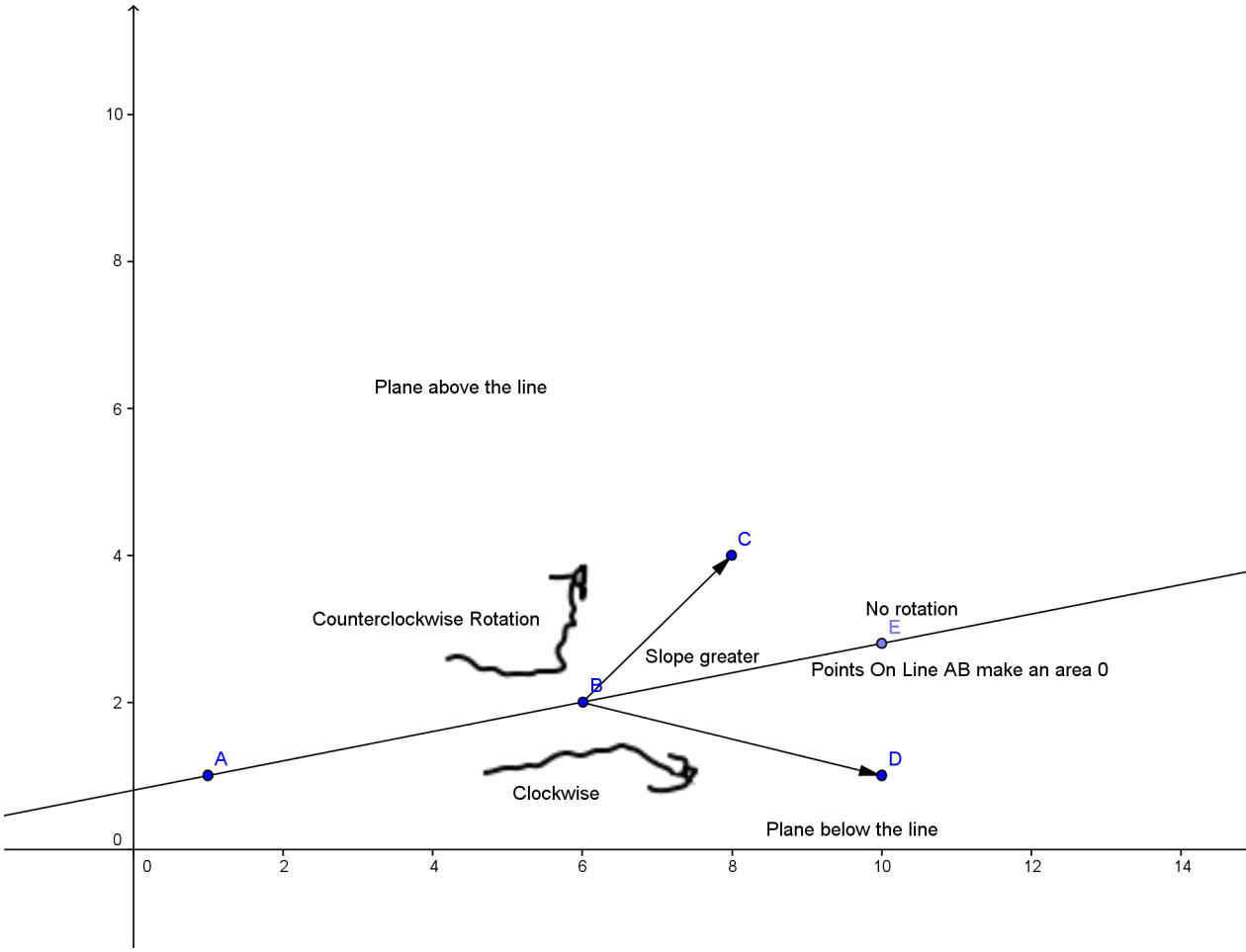
Plane above the line
C
Counterclockwise Rotation
No rotation
E
Slope greater
Points On Line AB make an area 0
B
A
D
Clockwise
Plane below the line
0
2
4
6
8
10
12
14

Figure 1: ccw

```
bool IsCollinear(const Point &A, const Point &B, const Point &C){
    return !TriArea(A, B, C);
}
```

17. A simple[1] polygon is called **convex** iff all the angles of the polygon are less than 180^0, assuming no 3 points are co-linear. Again, how to determine if a polygon is convex? If it is convex, then we need to check if the rotation will be the same taking each consecutive 3 vertex. If there is a different rotation, it is concave i.e. not convex.

```
int Sign(int x){
    if (x < 0)  return -1;
    else    return 1;
}

int Area_Of_Tr(Point A, Point B, Point C){
    int res = A.x*(B.y-C.y)+B.x*(C.y-A.y)+C.x*(A.y-B.y);
    return res;
}

bool IsConvex(Point p[], int n){
    int sgn;
    for (int i = 0; i < n; i++){
        int j = (i+1)%n;
        int k = (i+2)%n;
        int Area = Area_Of_Tr(p[i], p[j], p[k]);
        int test = Sign(Area);
        if (!i) sgn = test;
        else {
            if (test != sgn)    return false;
            //check if the direction is same as before
        }
    }
    return true;
}
```

18. If we want to check if two segments intersect or not, it can be done in two ways.

    (a) Find the intersection point of them. Then check if it lies on both segment. To test it, find the sum of distances of two endpoints of a segment from the intersection point. It lies in the segment if this is equal to the length of the segment i.e. distance of two endpoints. If this check is true for both segments, then they intersect.

    (b) There are some problems with this check. We have to find the intersection point, which can be a double coordinate. Also, there are double precision error problem. We can avoid this using ccw, see figure 2. It becomes pretty obvious now, when two segments intersect. Consider two segments $L_1(A, B)$ and $L_2(C, D)$. If A, B both lie on different side of CD and C, D both lie on different side of AB(figure 2.(a)), then we can say for sure that they both intersect. It can again, be done checking ccw. If a segment AB is co-linear with a point of another segment, then we need to check if that point C is between A and B. If it is between them, they intersect(2.b(2)). If it is not

[1] A polygon is simple if it has no edge which intersects the polygon. Otherwise, it is complex.

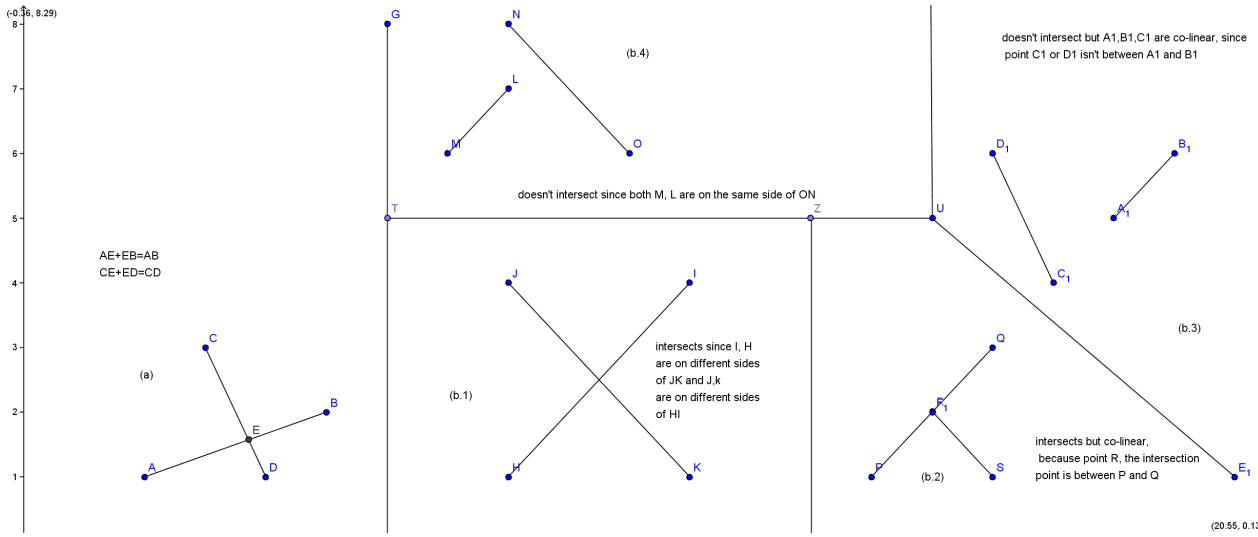

Figure 2: Segment-segment Intersection

between them(2.b(3)) or both endpoints of a segment lie on one side of another segment(2.b(4)) then they don't intersect.

```
bool Between(Point A, Point B, Point C){
    //checks if C is between A, B
    if (A > B)  swap(A, B);
    return (C.x >= A.x) && (C.x <= B.x) &&
    (C.y >= A.y) && (C.y <= B.y);

}

bool IsLeft(const Point &A, const Point &B, const Point &C){
    return TriArea(A, B, C) > 0;
}

bool XOR(bool x, bool y){
//false if both true or both false
    return !x ^ !y;
}

bool ifIntersects(Line l1, Line l2){
    Point A = l1.A, B = l1.B, C = l2.A, D = l2.B;
    if (XOR(IsLeft(A, B, C), IsLeft(A, B, D)) &&
    XOR(IsLeft(C, D, A), IsLeft(C, D, B)))
        return true;
    if (!TriArea(A, B, C) && Between(A, B, C))  return true;
    //if C is co-linear with AB and between A, B
    if (!TriArea(A, B, D) && Between(A, B, D))  return true;
    //else if D is co-linear with AB and between A, B
    if (!TriArea(C, D, A) && Between(C, D, A))  return true;
    //else if A is co-linear with CD and between C, D
    if (!TriArea(C, D, B) && Between(C, D, B))  return true;
    //else if B is co-linear with CD and between C, D
    return false;
}
```

1.1 Pick's Theorem

A polygon with lattice points may have some lattice points inside, and there are some lattice points on its boundary. Say, the number of inner lattice points is I and the number of boundary lattice points is B. Then, the area of this polygon can be computed by,

$$A = I + \frac{B}{2} - 1$$

This is a pretty useful formula. Using the area formula, run a program find the area of the polygon in the figure. Then, do the same using Pick's theorem. See if it works. You may, as well, think why it works. But its proof is not the motivation of this note. :)

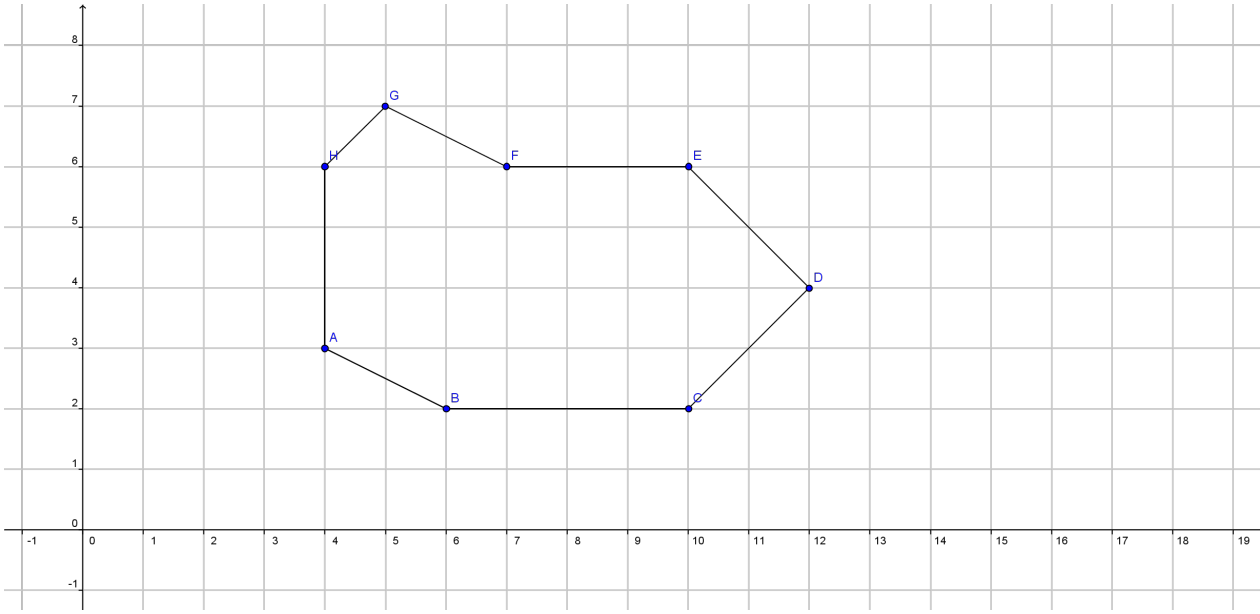

Figure 3: Pick's Theorem

Count Boundary Points How to count the number of lattice points on it's boundary? For this, we need to see another problem. Say, you are given two lattice end points of a segment. How many lattice points are there on the segment? More easily, just count the number of lattice points on a line joining O and $P(x, y)$. Take an example, $P(12, 8)$. From the figure, it is clear that there are 4 lattice points except the origin. It can be sensed in a systematic way. Also $(6, 4)$ is a lattice point on this line, then so is $(3, 2)$. Note that, the first lattice point on this line is $(3, 2)$. And their gcd is 1. Indeed, since $\gcd(12, 8) = 4$, we can divide them both by 4 and get a lattice point $(3, 2)$ on this line. This is the first one. Now, go 3 on right X and 2 on upper Y. We have the point $(6, 4)$. After that, $(9, 6)$ and finally $(12, 8)$. So, how many are there? Of-course $\frac{12}{3} = \frac{8}{2} = 4$. In general, if the point was (m, n) it would be $\gcd(m, n)$.

Back to the main problem. Let the two points be (x_1, y_1) and (x_2, y_2). We can count the number of points from the origin. So, let's say we have

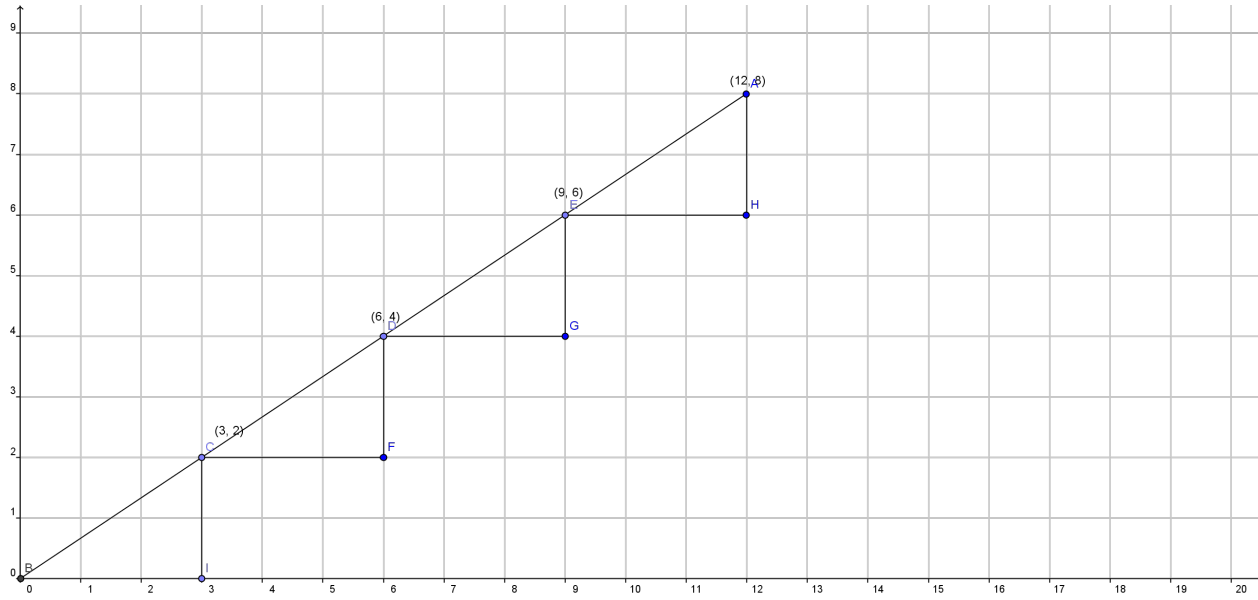

Figure 4: Number Of Lattice Points

transferred the origin to (x_1, y_1). Then the coordinate of the other point will be $(x_2 - x_1, y_2 - y_1)$. Now the number of points in this segment is $G(P_1, P_2) = \gcd(abs(x_1 - x_2), abs(y_1 - y_2))$. So, if the polygon is

$$P(P_0, P_1, \ldots, p_{n-1})$$

then the number of boundary points can be found summing all the consecutive $G(P_i, P_{i+1})$.

```
int NOfBoundaryPoints(Point P[], int n){
    int cnt = 0;
    for (int i = 0; i < n; i++){
        int j = (i+1)%n;
        int dx = abs(P[i].x-P[j].x);
        int dy = abs(P[i].y-P[j].y);
        cnt += gcd(dx, dy);
    }
    return cnt;
}
```

Check for figure 3 if this works?

1.2 Convex Hull

Convex Hull may be the most famous part of computational geometry. A **convex hull** of some lattice points is defined as the smallest convex polygon with lattice points which contains all the points inside the polygon. Several algorithms are known as finding the convex hull. *Graham Scan algorithm* is the most common. So, it is the best to discuss.

First see the pictures of the book given here: `http://www.mediafire.com/?5k4tb3j028eyaem`

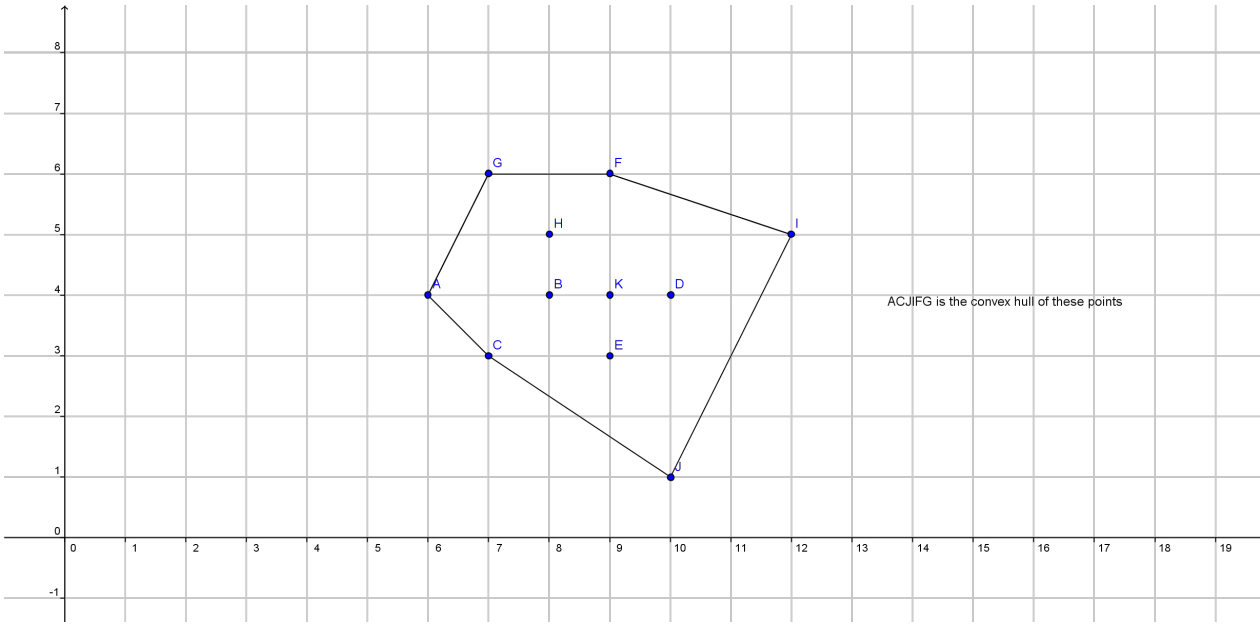

Figure 5: Convex Hull

1.3 Creating Polygon

Given some points of a polygon. Order them so that they form a polygon when connected consecutively.

Well, how to determine the order? The answer is actually simple, even though you find it hard to find after thinking sometime. This kind of sorting is known as **angular sorting**. First, we can determine the leftmost and lowermost point, say it P_0. We shall sort the points in counterclockwise order to the angle it makes with the positive X and P_0. And note that, just after the sorting, they become oriented so that they are the consecutive vertices. If it's not clear, see the simulation of Convex hull on the above link again. The question is how to compare the angles. Actually, isn't the question is to determine if the consecutive 3 points are making a left turn or right turn with P_0? Think carefully, I am leaving this for you to understand. Therefore, it reduces to the fact that, we just need to change the compare function in order that, it returns whether a point B is on the left of AP_0 or not. Again, the ccw function.

Now, back to the original problem. We need to run a $O(n)$ loop now to determine which points are on the hull. What we know currently is, if we can determine which points are on the hull, we find the convex hull. Because of angular sorting, they would be the consecutive points of a polygon. How to decide which points to take. As per the simulation says, you can understand while running through the points whenever a point makes a clockwise turn, we discard the point. And after the operation, we are left with the convex hull.

Here is a code, you can see that almost every Graham-Scan code is almost same. This code is due to Zobayer vai.
P[]: holds all the points
C[]: holds points on the hull

np: number of points in P[]
nc: number of points in C[]
to handle duplicate, call makeUnique() before calling convexHull()
call convexHull() if you have np >= 3
to remove co-linear points on hull, call compress() after convexHull()

```
struct point {
    int x, y;
} P[MAX], C[MAX], P0;

int triArea2(const point &a, const point &b, const point &c) {
    return (a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.y-b.y));
}

int sqDist(const point &a, const point &b) {
    return ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

bool comp(const point &a, const point &b) {
    int d = triArea2(P0, a, b);
    if(d < 0) return false;
    if(!d && sqDist(P0, a) > sqDist(P0, b)) return false;
    return true;
}

bool normal(const point &a, const point &b) {
    return ((a.x==b.x) ? a.y < b.y : a.x < b.x);
}

bool issame(const point &a, const point &b) {
    return (a.x == b.x && a.y == b.y);
}

void makeUnique(int &np) {
    sort(&P[0], &P[np], normal);
    np = unique(&P[0], &P[np], issame) - P;
}

void convexHull(int &np, int &nc) {
    int i, j, pos = 0;
    for(i = 1; i < np; i++)
        if(P[i].y<P[pos].y || (P[i].y==P[pos].y && P[i].x<P[pos].x))
            pos = i;
    swap(P[0], P[pos]);
    P0 = P[0];
    sort(&P[1], &P[np], comp);
    for(i = 0; i < 3; i++) C[i] = P[i];
    for(i = j = 3; i < np; i++) {
        while(triArea2(C[j-2], C[j-1], P[i]) < 0) j--;
        C[j++] = P[i];
    }
    nc = j;
}

void compress(int &nc) {
    int i, j, d;
    C[nc] = C[0];
    for(i=j=1; i < nc; i++) {
        d = triArea2(C[j-1], C[i], C[i+1]);
        if(d || (!d && issame(C[j-1], C[i+1]))) C[j++] = C[i];
    }
    nc = j;
}
```