Computational Geometry

- Computational Geometry เป็นอีกหัวข้อหนึ่งที่พบบ่อยๆ ในการแข่งขัน

- สังเกตได้ว่าปัญหาที่เกี่ยวกับรูปทรงเรขาคณิต (Geometry–related problem) จะไม่ถูกลองทำในช่วงต้นของการแข่งขัน เพราะว่าคำตอบจากการแก้ปัญหาทางเรขาคณิตนั้นมีโอกาสต่ำในการที่จะ accept ในระหว่างแข่งขัน เมื่อเทียบกับปัญหาประเภทอื่นๆ อาทิ complete search หรือ dynamic programming

- โดยทั่วไปแล้วปัญหาเกี่ยวกับ geometry มีดังต่อไปนี้
- หลายปัญหาเกี่ยวกับ geometry มักมี 1 หรือหลายๆ test case ที่ล่อลวง เช่น
  - เกิดอะไรขึ้นถ้าเส้นตรงลากเป็นแนวดิ่ง
  - เกิดอะไรขึ้นถ้าจุดหลายจุด อยู่บนแนวเส้นตรงเดียวกัน
  - เกิดอะไรขึ้นถ้า polygon เป็น concave
- ดังนั้นเป็นความคิดที่ดีที่จะทดสอบคำตอบของ geometry ด้วยเทสเคส พวกมุมหรือขอบ ก่อนที่จะส่ง
- มีโอกาสที่ความแม่นยำของจำนวนทศนิยมผิด ทำให้ได้คำตอบผิดได้
- คำตอบของปัญหาเกี่ยวกับ geometry นั้นส่วนใหญ่ code ยาวและช้า

- จากเหตุผลที่กล่าวมาเป็นสาเหตุให้คนแข่งหลายคนเห็นว่าไปใช้เวลากับข้ออื่นดีกว่า
  - ทั้งนี้ผู้เข้าแข่งขันลืมสูตรพื้นฐานที่สำคัญบางอย่างหรือไม่ก็ไม่สามารถแก้สมการ แก้สูตรที่ต้องการจากสูตรอย่างง่าย
  - ผู้แข่งขันไม่ได้เตรียม library function ก่อนแข่งและความพยายาม code ในช่วงเวลาจำกัดที่มีความเครียดทำให้ bug ง่าย ทำให้ส่วนใหญ่แล้วหากแข่ง ACM ICPC ทีมส่วนใหญ่จะเอา hard copy ที่บันทึกสูตรทาง geometry และ library function เข้าไปด้วย

- บทนี้จะเพิ่มโอกาสในการทำโจทย์เกี่ยวกับ geometry–related problems ในการแข่ง มีหลักๆ สองเรื่อง
- เรื่องแรกเกี่ยวกับ English geometric terminologies และสูตรพื้นฐานของวัตถุในมิติต่างๆ 0D, 1D, 2D, and 3D. ในเรื่องนี้ถูกใช้เป็นการอ้างอิงอย่างรวดเร็วเมื่อเจอปัญหา geometry problems
- เรื่องที่สองจะพูดเกี่ยวกับหลายๆ algorithms ใน 2D **polygons** ซึ่งมีหลายอันที่มี library มาให้ทำให้เราสะดวกขึ้น เช่น algorithm ที่บอกว่า polygon นั้นเป็น convex หรือ concave, ตัดสินว่าจุดนี้อยู่ในหรือนอก polygon ที่กำหนดหรือไม่, การตัด polygon ด้วยเส้นตรง, การหา convex hull จากเซตของจุดที่ได้รับ

- จะ highlight special cases ที่สามารถเกิดขึ้นได้และ/หรือ เลือก implementation ที่ลดจำนวน special cases
- ระวัง floating point operations (เช่น division, square root, และอื่นๆ สามารถทำให้เกิดข้อผิดพลาดทางทศนิยมได้) และทำงานกับ integers เมื่อทำได้ (เช่น integer additions, subtractions, multiplications)
- ถ้าเราต้องการทำงานกับ floating point จริงๆ, เราต้องเทียบ floating point ว่าเท่ากันด้วยวิธีนี้ fabs(a – b) < EPS เมื่อ EPS เป็นเลขน้อยๆ เช่น 1e–9 แทนการเทียบ a == b และเมื่อเราต้องการเทียบว่า floating point number $x \geq 0.0$, เราจะใช้ x > –EPS (คล้ายกับการทดสอบว่าถ้า $x \leq 0.0$, เราก็จะใช้ < EPS).

0D Objects: Point

- จุด **Point** เป็น object พื้นฐานในการสร้าง object ที่มีมิติที่สูงขึ้น
- ใน 2D Euclidean space, จุด โดยทั่วไปแล้วแทนด้วย struct in C/C++ ด้วยสมาชิก 2 ตัว: คู่อันดับ **x** และ **y** โดยที่จุดกำเนิดคือคู่อันดับ (0, 0)
- ถ้าปัญหาอธิบายโดยใช้ คู่อันกับที่เป็นจำนวนเต็มให้ใช้ int ถ้าไม่เช่นนั้นก็ใช้ double
- ทั้งนี้ต่อไปจะยกตัวอย่างโดยใช้ struct ที่เป็น floating-point
- ต่อไปเป็น default constructor ที่จะใช้ต่อไป

```
// 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
```

เรียงจุด

- ในบางครั้งเราต้องการเรียงจุด
- เราสามารถทำได้ไม่ยากโดยใช้ overloading the less than operator ภายใน struct point จากนั้นใช้ sort library

ตัวอย่าง

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  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
// in int main(), assuming we already have a populated vector<point> P
sort(P.begin(), P.end()); // comparison operator is defined above
```

จุดเท่ากันไหม

- ในบางครั้งถ้าเราต้องการทดสอบว่าจุดสองจุดนี้เท่ากันไหม
- เราทำได้ไม่ยากโดยเพิ่ม overloading equal operator ภายใน struct point

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
 // 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)); } };
// in int main()
point P1(0, 0), P2(0, 0), P3(0, 1);
printf("%d\n", P1 == P2); // true
printf("%d\n", P1 == P3); // false
```

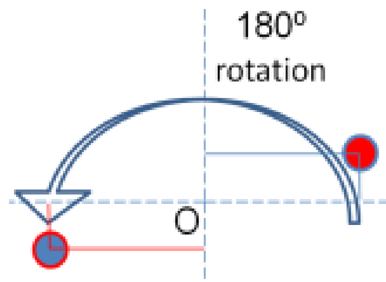
ระยะทางระหว่างสองจุด

- หากเราต้องการวัดระยะทางระหว่างจุดสองจุด คำนวณได้โดยใช้สูตรต่อไปนี้

```
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); }
```

หมุน

- เราสามารถหมุนจุดไป θ องศาทวนเข็มนาฬิกาจากจุดกำเนิด โดยใช้ rotation matrix:
- รูปต่อไปเป็นตัวอย่างการหมุนจุด (10,3) ไป 180 องศาตามเข็มนาฬิกาจากจุดกำเนิด



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

```
// 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)); }
```

-

```
#include <bits/stdc++.h>
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; };    // 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)); } };
```

```
bool areSame(point_i p1, point_i p2) { // integer version
  return p1.x == p2.x && p1.y == p2.y; } // precise comparison

bool areSame(point p1, point p2) { // floating point version
  // use EPS when testing equality of two floating points
  return fabs(p1.x - p2.x) < EPS && fabs(p1.y - p2.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) {
  // rotation matrix R(theta) = [cos(theta) -sin(theta)]
  //                            [sin(theta)  cos(theta)]
  double rad = DEG_to_RAD(theta); // must work in radian
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }
```

```
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++)
    cout<< P[i].x<<" "<< P[i].y<<endl;
```

```
// 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]);
  cout<<"Euclidean distance between P[0] and P[5] = "<< d<<endl;//should be
5.000
  return 0;
}
```

https://bit.ly/2ZpfscZ

1D Objects: Line

- **เส้นตรงหรือ Line** ใน 2D Euclidean space คือเซตของจุดที่คู่อันดับสอดคล้องกับสมการเส้นตรงที่กำหนดให้ $ax + by + c = 0$
- ฟังก์ชันต่อไปในหัวข้อนี้ สมมติว่าสมการเส้นตรงมี $b = 1$ สำหรับเส้นตรงที่ไม่ใช่แนวดิ่งและ $b = 0$ สำหรับเส้นตรงแนวดิ่งถ้าไม่ได้กำหนดเป็นอย่างอื่น
- เส้นตรงโดยทั่วไปแล้วเราจะแทนด้วย struct in C/C++ ด้วยสมาชิกสามตัวนั่นคือ: สัมประสิทธิ์ a, b, และ c ของเส้นตรงนั้นเอง

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

แปลงจุดเป็นเส้น

- เราสามารถคำนวณเส้นที่ต้องการได้จากสมการ ถ้าเราได้รับอย่างน้อย 2 จุดที่ผ่านเส้นตรงผ่านฟังก์ชันต่อไปนี้

```
// 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;

} }
```

- เราสามารถทดสอบว่าเส้นตรง 2 เส้นขนานกันหรือไม่ โดยตรวจสอบว่าถ้าค่าสัมประสิทธิ *a* และ *b* เหมือนกัน

- เราสามารถทดสอบต่ออีกว่าเส้นตรงสองเส้นนั้นเป็นเหมือนกันหรือไม่ จากการตรวจสอบว่ามันขนานกันและค่าสัมประสิทธิ *c* เหมือนกัน (นั่นคือค่าสัมประสิทธิทั้งสามค่า *a*, *b*, *c* เหมือนกัน).

- สังเกตว่าในตอนที่เรา implement เราได้ fixed ค่าของสัมประสิทธิ *b* ให้เป็น 0.0 สำหรับเส้นแนวดิ่งและเป็น 1.0 สำหรับเส้นที่ไม่ใช่เส้นแนวดิ่ง

```
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); }
```

- ถ้าเส้นตรงสองเส้นไม่ขนานกัน (และไม่ใช่เส้นเดียวกัน), พวกมันจะตัดกันที่จุดจุดหนึ่ง

- จุดตัดจุดนั้น (x, y) สามารถหาได้โดยการแก้สมการเส้นตรงสองเส้นสองตัวแปร: $a_1x + b_1y + c_1 = 0$ และ $a_2x + b_2y + c_2 = 0$

```
// 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; }
```

Line Segment

- **ส่วนของเส้นตรง หรือ Line Segment** คือเส้นตรงที่มีจุดปลายสองข้อที่มีความยาวจำกัด
- **เวกเตอร์ Vector** คือส่วนของเส้นตรง (ดังนั้นมันมีจุดปลายสองจุดและมีความยาว) ที่มีทิศทาง(*direction*)
- โดยทั่วไปแล้ว vector ถูกแสดงด้วย struct ในC/C++ ด้วยสมาชิกสองตัว: ขนาดของ **x** และ **y** ของเวกเตอร์ ทั้งนี้ขนาดของเวกเตอร์ถูกปรับขนาดได้หากจำเป็น
- เราสามารถแปลงจุดให้สอดคล้องกับเวกเตอร์ที่กำหนดให้ได้โดยสร้างจุดปลายอีกจุดที่มีขนาดในแกน x และ y สอดคล้องกับเวกเตอร์

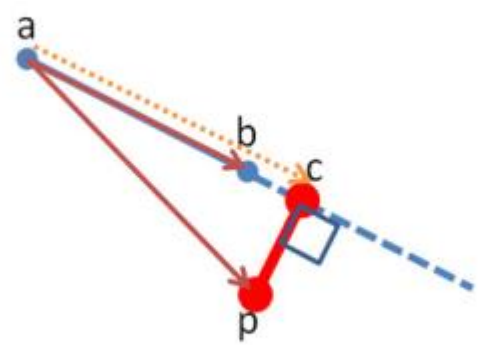
```
struct vec { double x, y; // name: 'vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

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); }
// ย้ายจุด p ตามเวกเตอร์ v
```

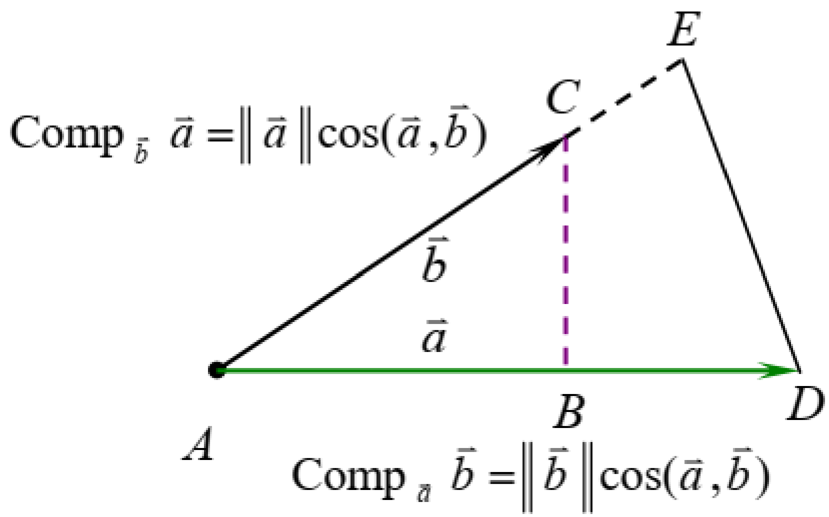
- เมื่อกำหนดจุด p และ**เส้นตรง** l (ที่อธิบายด้วยจุดสองจุด a และ b), เราสามารถคำนวนระยะทางสั้นที่สุดจาก p ไป l
- โดยเริ่มต้นคำนวณตำแหน่งของจุด c ใน l ที่ใกล้กับจุด p มากที่สุด หลังจากนั้นหาระยะทางระหว่างจุด p และ c



- เราสามารถมองจุด c เป็นจุด a ที่ถูกแปลงโดยการย่อ/ขยายขนาดของ u ของเวกเตอร์ ab หรือ $c = a + u \times ab$

- ในการคำนวณ u, เราจะทำ scalar projection ของเวกเตอร์ ap ลงบนเวกเตอร์ ab โดยใช้ dot product (ac = u x ab)

- อธิบาย จากนิยาม dot product ของเวกเตอร์
- $\vec{a} \cdot \overrightarrow{b} = \|\vec{a}\|\|\vec{b}\|\cos(\vec{a},\vec{b})$
- จำนวนจริง $\|\vec{b}\|\cos(\vec{a},\vec{b})$ จะเรียกว่าส่วนประกอบ(Component) ของ $\vec{b}$ บน $\vec{a}$ เขียนแทนด้วย $Comp_{\vec{a}}\ \vec{b}$
- ดังนั้น $Comp_{\vec{a}}\ \vec{b} = \|\vec{b}\|\cos(\vec{a},\vec{b})$
- เช่นเดียวกัน $Comp_{\vec{b}}\ \vec{a} = \|\vec{a}\|\cos(\vec{a},\vec{b})$



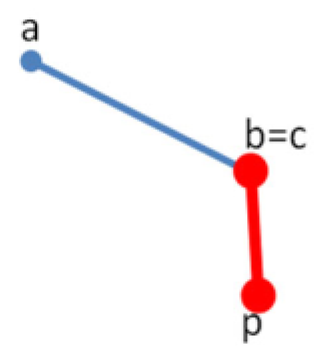
- ถ้า $\vec{a}$และ $\vec{b}$ ไม่ใช้เวกเตอร์ศูนย์ จะได้ว่า

$$\mathrm{Comp}_{\vec{a}}\ \vec{b} = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|}$$

$$\mathrm{Comp}_{\vec{b}}\ \vec{a} = \frac{\vec{a}\cdot\vec{b}}{\|\vec{b}\|}$$

```cpp
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
```

ถ้าเราได้รับ**<u>ส่วนของเส้นตรง</u>** (นิยามด้วยจุดปลายสองจุด *a* และ *b*), แล้วระยะทางสั้นที่สุดจากจุด *p* ไปยังส่วนของเส้นตรง *ab* จะต้องพิจารณา *2* two special case: จุดปลาย a และ b ของส่วนของเส้นตรง code คล้ายกับ distToLine function ก่อนหน้า



```
// 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
```

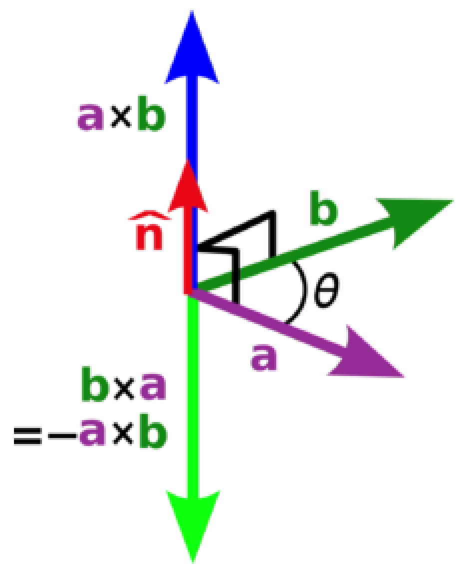
- เราสามารถคำนวณหามุม *aob* เมื่อกำหนดจุด 3 จุดมาให้: *a*, *o*, และ *b*, โดยการใช้ dot product เนื่องจาก $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$, เราจะได้ว่า $theta = \arccos(oa \cdot ob / (|oa| \times |ob|))$

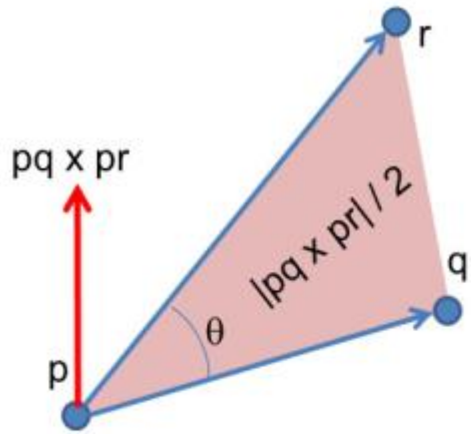
```
double angle(point a, point o, point b) { //returns angle aob in rad
  vec oa = toVector(o, a), ob = toVector(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }
```

- เมื่อกำหนดเส้นตรงที่นิยามด้วยจุดสองจุด *p* และ *q* มาให้, เราต้องการตัดสินว่าจุด *r* อยู่ทางซ้ายหรือขวาของเส้น หรือว่าทั้งสามจุดอยู่บนแนวเดียวกัน ทำได้โดยการใช้ *cross product.*

- กำหนดให้ *pq* และ *pr* เป็นสองเวกเตอร์ที่ได้จากสามจุด cross product *pq* × *pr* ให้ผลลัพธ์เป็นอีกเวกเตอร์ที่ตั้งฉากกับทั้ง *pq* และ *pr.*
- ขนาดของเวกเตอร์นี้จะเท่ากับเส้นทแยงมุมของสี่เหลี่ยมด้านขนานของสองเวกเตอร์

- ผลคูณเชิงเวกเตอร์ (Cross product) เวกเตอร์ $\vec{a}$ และ $\vec{b}$ นิยามด้วย
- $\vec{a} \times \vec{b} = \|\vec{a}\| \|\vec{b}\| \sin(\vec{a}, \vec{b})\, \vec{u}$
- เมื่อ $\vec{u}$ คือเวกเตอร์หนึ่งหน่วยที่ตั้งฉากกับ $\vec{a}$ และ $\vec{b}$





- ถ้าขนาดเป็น positive/zero/negative,แล้วเรารู้ว่า $p \longrightarrow q \longrightarrow r$ is a left turn/collinear/right turn, ตามลำดับ
- ทั้งนี้การทดสอบหมุนซ้ายเรามักเรียกว่าทดสอบหมุนทวนเข็ม **CCW (Counter Clockwise) Test**

```
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; }
```

https://bit.ly/2PlCJb3

แบบฝึกหัด

- จงคำนวณระยะทางแบบยูคลิเดียนระหว่างจุด (2, 2) และ (6, 5)
- จงหมุนจุด (10, 3) ไป 90 องศาทวนเข็มนาฬิการอบจุดกำเนิด พิกัดของจุดใหม่เป็นเท่าใด (ลองเทียบทำมือดู)
- จงหมุนจุด (10, 3) ไป 77 องศาทวนเข็มนาฬิการอบจุดกำเนิด พิกัดของจุดใหม่เป็นเท่าใด
- ลองทำ codejam ข้อ ufo เทสเคสเล็กดู
- คิดว่าเส้นตรงที่อธิบายได้ด้วย y= mx+c เมื่อ m เป็นความชันกับ ax+by+c=0 อันไหนดีกว่ากัน

- คำนวณสมการเส้นตรงที่ผ่านจุด (2,2) กับ (4,3)
- คำนวณสมการเส้นตรงที่ผ่านจุด (2,2) กับ (2,4)
- แปลงจุด c (3,2) ให้สอดคล้องกับเวกเตอร์ ab ที่นิยามด้วยสองจุด a(2,2) และ b(4,3) พิกัดของจุดใหม่คืออะไร
- หมุนจุด c(3,2) ไป 90 องศาทวนเข็มรอบจุดกำเนิด
- หมุนจุด c(3,2) ไป 90 องศาทวนเข็มรอบจุด (2,1) (hint: ต้องแปลงจุด)
- กำหนดให้จุด $a(2, 2)$, $o(2, 4)$ละ $b(4, 3)$ คำนวณมุม aob ในหน่วยองศา
- จุด r(35,30) อยู่ทางด้านไหนของเส้นตรงที่ผ่านจุด (3,7) และ (11,13)