# Computational Geometry

HKU ACM ICPC Training 2010

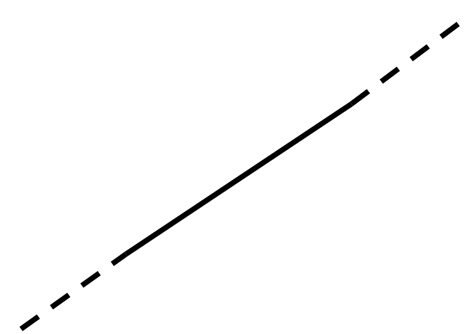
# What is Computational Geometry?

- Deals with geometrical structures
  - Points, lines, line segments, vectors, planes, etc.
- A relatively boring class of problems in ICPC
  - CoGeom problems are usually straightforward
  - Implementation is tedious and error-prone
- In this training session we only talk about 2-dimensional geometry
  - 1-D is usually uninteresting
  - 3-D is usually too hard

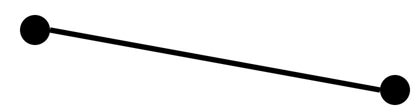
# Basic definitions

- Point
  - Specified by two coordinates (x, y)

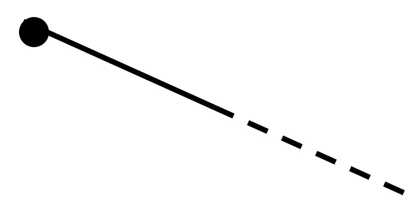
- Line
  - Extends to infinity in both directions



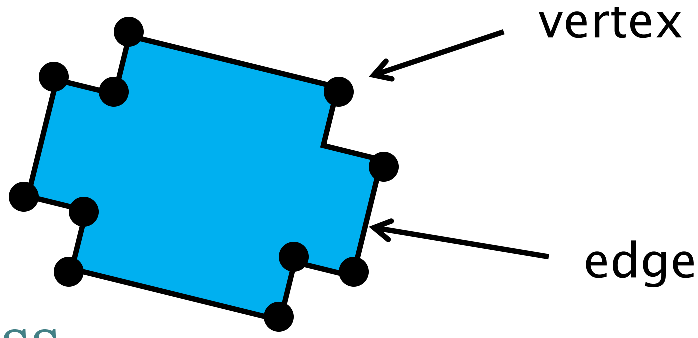
- Line segment
  - Specified by two endpoints



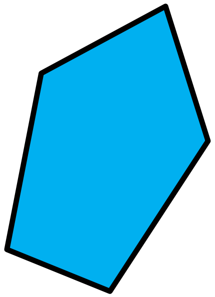
- Ray
  - Extends to infinity in one direction



# Basic definitions

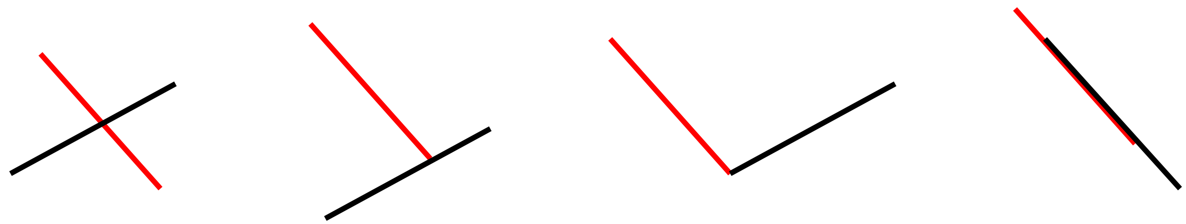



- Polygon
  - We assume edges do not cross
- Convex polygon
  - Every interior angle is at most 180 degrees
  - Precise definition of *convex*: For any two points inside the polygon, the line segment joining them lies entirely inside the polygon



# What makes CoGeom problems so annoying?

- Precision error
  - Avoid floating-point computations whenever possible (see later slides)
- Degeneracy
  - Boundary cases
  - For example, imagine how two line segments can intersect



# I’m bored…

- Do I really need to learn these??

# ACM World Finals 2005

- A: Eyeball Benders
- B: Simplified GSM Network
- C: The Traveling Judges Problem
- D: cNteSahruPfefrlefe
- E: Lots of Sunlight
- F: Crossing Streets
- G: Tiling the Plane
- H: The Great Wall Game
- I: Workshops
- J: Zones

Geometry

Shortest Path

Matching

# I’m bored…

- Do I really need to learn these??
  - It seems that the answer is ‘YES’

# Outline

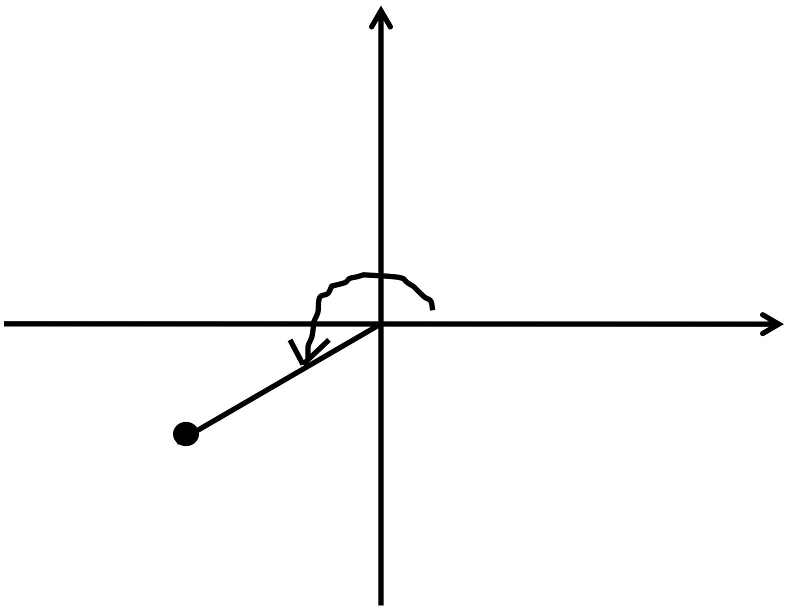
- Basic operations
  - Distance, angle, etc.
  - Cross product
  - Intersection
- Polygons
  - Area
  - Containment
- Convex hull
  - Gift wrapping algorithm
  - Graham scan

# Distance between two points

- Two points with coordinates $(x1, y1)$ and $(x2, y2)$ respectively
- Distance = $\mathrm{sqrt}( (x1-x2)^2 + (y1-y2)^2 )$
- Square root is kind of slow and imprecise
- If we only need to check whether the distance is less than some certain length, say R
- if $( (x1-x2)^2 + (y1-y2)^2 ) < R^2$ …

# Angle

- Given a point (x, y), find its angle about the origin (conventionally counterclockwise)
  - Answer should be in the range (-π, π]



Sorry I'm not an artist

# Angle

- Solution: Inverse trigonometric function
- We use arctan (i.e. $\tan^{-1}$)
- atan(z) in C++
  - need to #include <cmath>
- atan(z) returns a value θ for which tan θ = z
  - Note: all C++ math functions represent angles in radians (instead of degrees)
    - radian = degree * π / 180
    - π = acos(-1)
- Solution(?): θ = atan(y/x)

# Angle

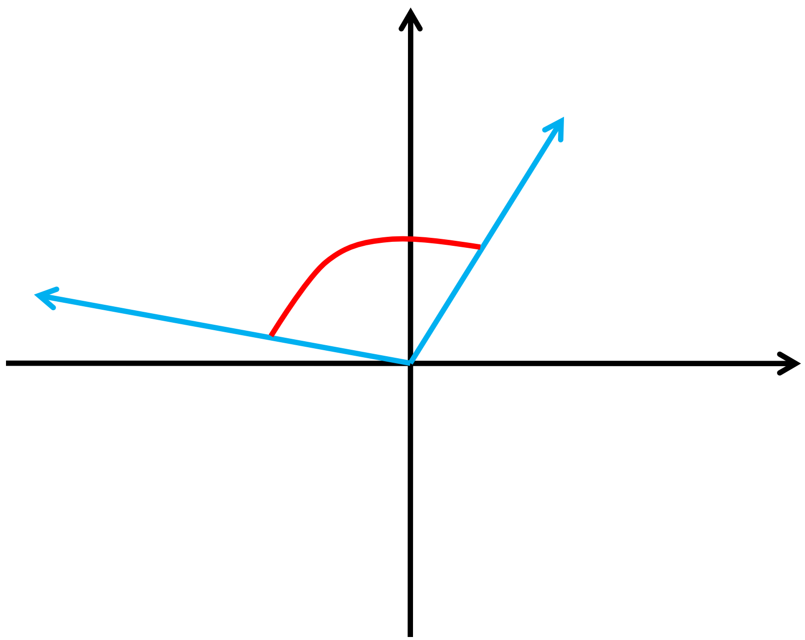
- Solution(?): $\theta = \text{atan}(y/x)$
- Bug #1: Division by zero
  - When $\theta$ is $\pi/2$ or $-\pi/2$
- Bug #2: y/x doesn't give a 1-to-1 mapping
  - $x=1, y=1,$ $\mathbf{y/x=1, \theta=\pi/4}$
  - $x=-1, y=-1,$ $\mathbf{y/x=1, \theta=-3\pi/4}$
- Fix: check sign of x
  - Too much trouble… any better solution?

# Angle

- Solution: $\theta = \text{atan2}(y, x)$
  - #include <cmath>
- That's it
- Returns answer in the range [-π, π]
  - Look at your C++ manual for technical details
- Note: The arguments are **(y, x)**, not (x, y)!!!

# Angle between two vectors

- Find the minor angle (i.e. <= π) between two vectors **a**(x1, y1) and **b**(x2, y2)
- Solution #1: use atan2 for each vector, then subtract

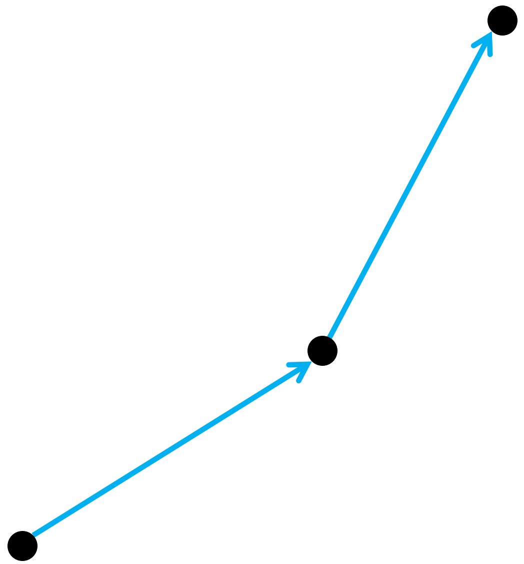


# Angle between two vectors

- Solution #2: Dot product
- Recall: $\mathbf{a}\bullet\mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta$
- Therefore: $\theta = \mathrm{acos}(\mathbf{a}\bullet\mathbf{b} / (|\mathbf{a}||\mathbf{b}|))$
  - Where: $\mathbf{a}\bullet\mathbf{b} = x1*x2+y1*y2$
  - And: $|\mathbf{a}| = \mathrm{sqrt}(x1*x1+y1*y1)$ (similar for $|\mathbf{b}|$)
- Note: acos returns results in the range $[0, \pi]$
- Note: When either vector is zero the angle between them is not well-defined, and the above formula leads to division by zero

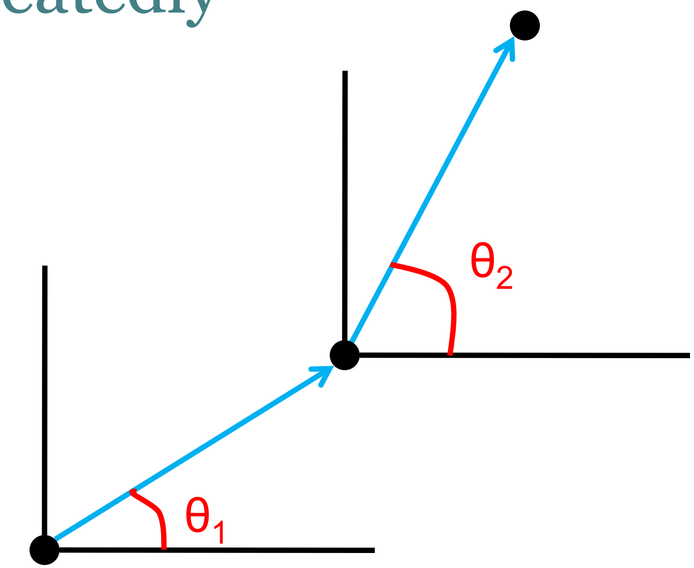
# Left turn or right turn?

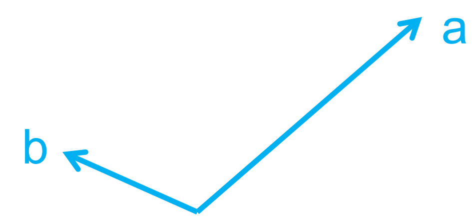
- Are we making a left turn or right turn here?
  - Of course easy for us to tell by inspection
  - How about (121, 21) → (201, 74) → (290, 123) ?



# Left turn or right turn?

- Solution #1: Using angles
- Compute $\theta_2 - \theta_1$
- "Normalize" the result into the range $(-\pi, \pi]$
  - By adding/subtracting $2\pi$ repeatedly
- Positive: left turn
- Negative: right turn
- 0 or $\pi$: up to you



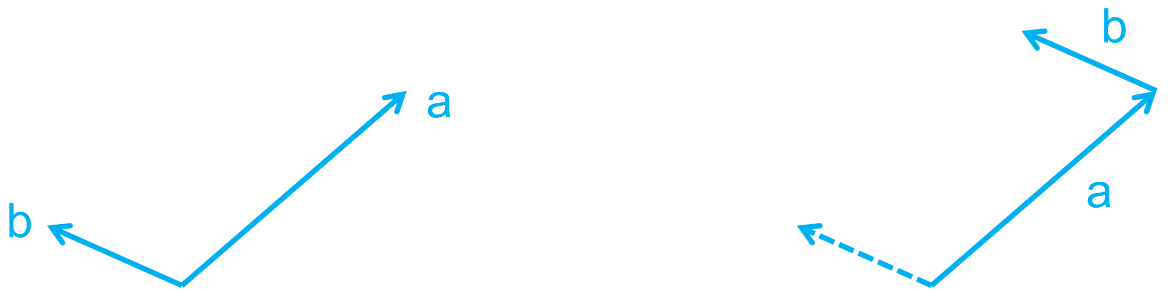



# Cross product

- Solution #2 makes use of cross products (of vectors), so let's review
- The cross product of two vectors $\mathbf{a}(x_a, y_a)$ and $\mathbf{b}(x_b,y_b)$ is $\mathbf{a}\times\mathbf{b} = (x_a*y_b-x_b*y_a)\mathbf{k}$
  - **k** is the unit vector in the positive z-direction
  - **a** and **b** are viewed as 3-D vectors with having zero z-coordinate
  - Note: $\mathbf{a}\times\mathbf{b} \neq \mathbf{b}\times\mathbf{a}$ in general
- Fact: if $(x_a*y_b-x_b*y_a) > 0$, then **b** is to the left of **a**

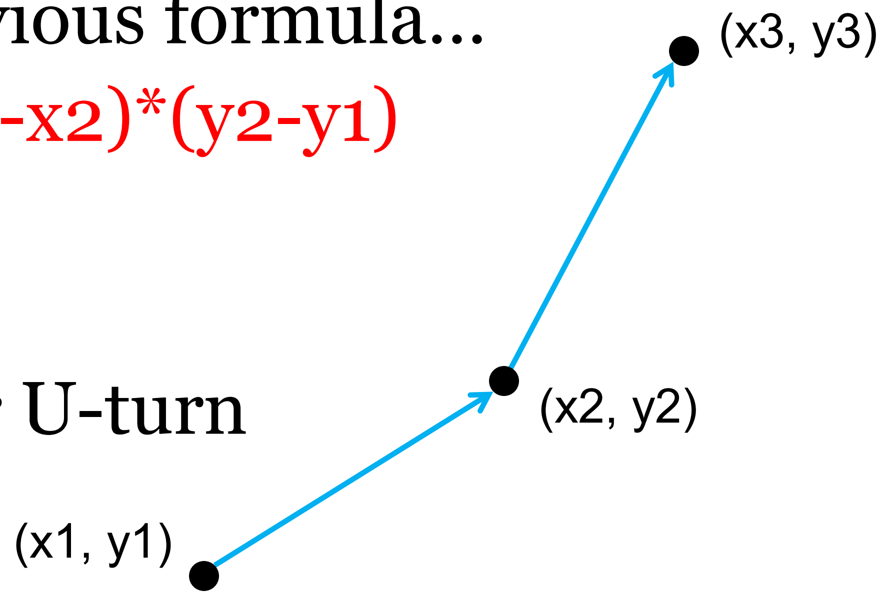
# Left turn of right turn?

- Observation: "**b** is to the left of **a**" is the same as "**a**→**b** constitutes a left turn"



# Left turn or right turn?

- Solution 2: A simple cross product
- Take **a** = (x2-x1, y2-y1)
- Take **b** = (x3-x2, y3-y2)
- Substitute into our previous formula…
- P = (x2-x1)*(y3-y2)-(x3-x2)*(y2-y1)
- P > 0: left turn
- P < 0: right turn
- P = 0: straight ahead or U-turn



# crossProd(p1, p2, p3)

- We need this function later
- `function crossProd(p1, p2, p3: Point)`

```
{
    return (p2.x-p1.x)*(p3.y-p2.y) –
           (p3.x-p2.x)*(p2.y-p1.y);
}
```

- Note: Point is not a predefined data type – you may define it

# Intersection of two lines

- A straight line can be represented as a linear equation in standard form $Ax+By=C$
  - e.g. $3x+4y-7 = 0$
  - We assume you know how to obtain this equation through other forms such as
    - slope-intercept form
    - point-slope form
    - intercept form
    - two-point form (most common)

# Intersection of two lines

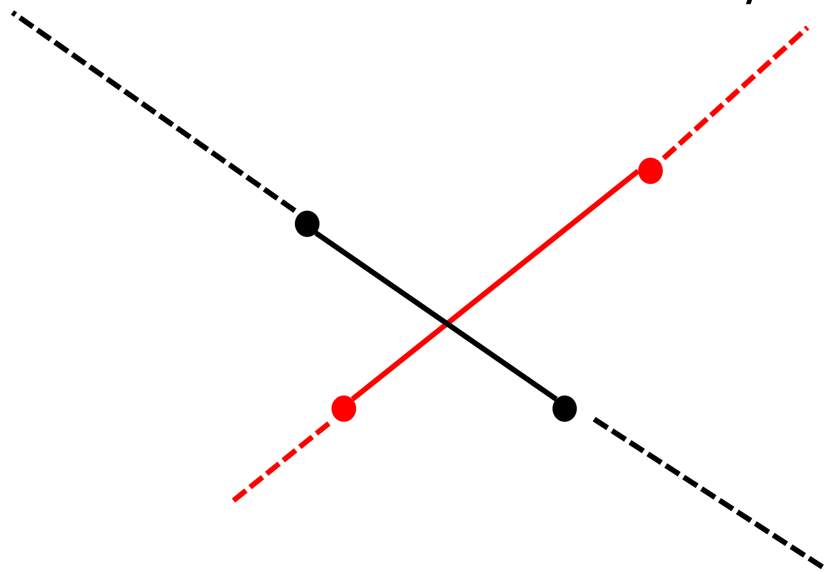
- Given L1: Ax+By=C and L2: Dx+Ey=F
- To find their intersection, simply solve the system of linear equations
  - Using whatever method, e.g. elimination
- Using elimination we get
  - x = (C*E-B*F) / (A*E-B*D)
  - y = (A*F-C*D) / (A*E-B*D)
  - If A*E-B*D=0, the two lines are parallel
    - there can be zero or infinitely many intersections

# Intersection of two line segments

- Method 1:
  - Assume the segments are lines (i.e. no endpoints)
  - Find the intersection of the two lines
  - Check whether the intersection point lies between all the endpoints
- Method 2:
  - Check whether the two segments intersect
    - A lot easier than step 3 in method 1. See next slide
  - If so, find the intersection as in method 1

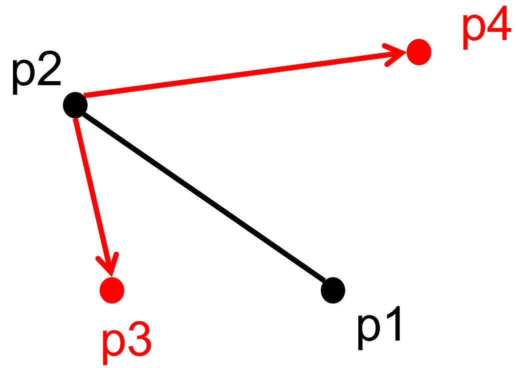
# Do they intersect?

- Observation: If the two segments intersect, the two red points must lie on different sides of the black line (or lie exactly on it)
- The same holds with black/red switched



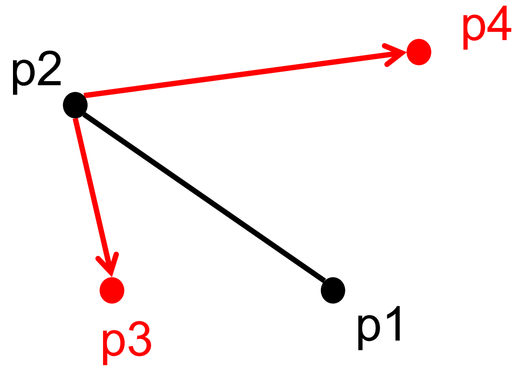
# Do they intersect?

- What does “different sides” mean?
  - one of them makes a left turn (or straight/U-turn)
  - the other makes a right turn (or straight/U-turn)
- Time to use our crossProd function



# Do they intersect?

- turn_p3 = crossProd(p1, p2, p3)
- turn_p4 = crossProd(p1, p2, p4)
- The red points lie on different sides of the black line if (turn_p3 * turn_p4) <= 0
- Do the same for black points and red line

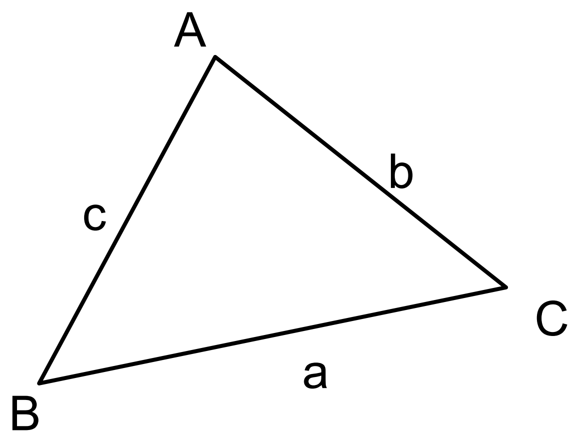


# Outline

- Basic operations
    - Distance, angle, etc.
    - Cross product
    - Intersection
- Polygons
    - Area
    - Containment
- Convex hull
    - Gift wrapping algorithm
    - Graham scan

# Area of triangle

- Area = Base * Height / 2
- Area = a * b * sin(C) / 2
- Heron's formula:
    - Area = sqrt( s(s-a)(s-b)(s-c) )
    - where s = (a+b+c)/2 is the semiperimeter



# Area of triangle

- What if only the vertices of the triangle are given?
- Given 3 vertices (x1, y1), (x2, y2), (x3, y3)
- Area = abs( x1*y2 + x2*y3 + x3*y1 - x2*y1 - x3*y2 - x1*y3 ) / 2
- Note: abs can be omitted if the vertices are in **counterclockwise** order. If the vertices are in clockwise order, the difference evaluates to a negative quantity

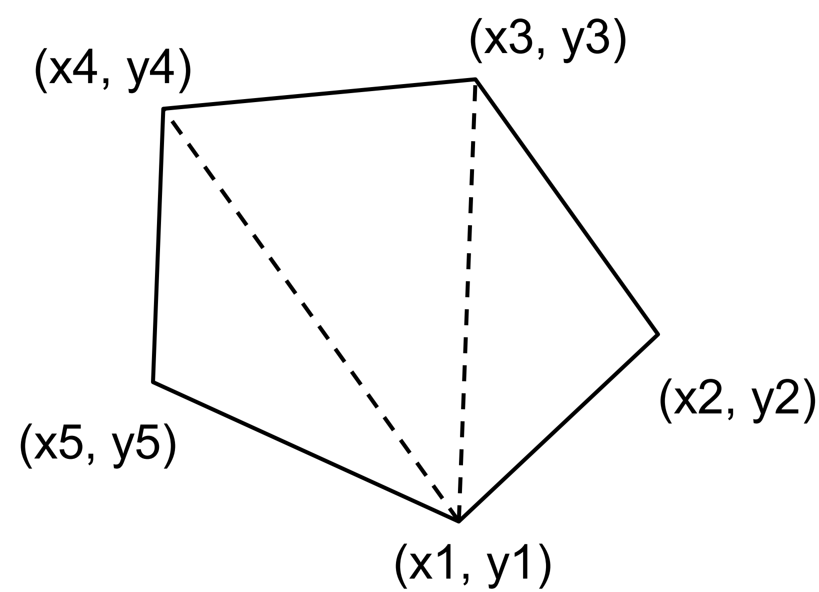
# Area of triangle

- That hard-to-memorize expression can be written this way:

- Area = ½ * $\begin{vmatrix} x1 & y1 \\ x2 & y2 \\ x3 & y3 \\ x1 & y1 \end{vmatrix}$

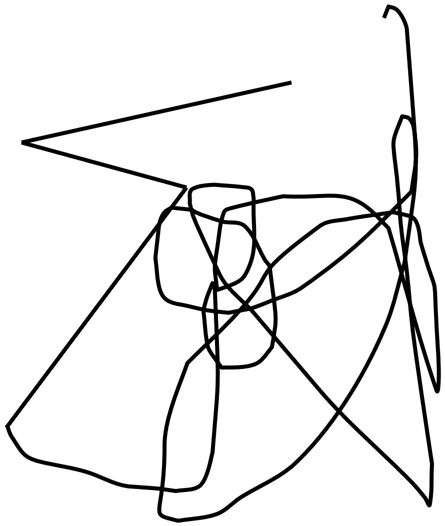
# Area of convex polygon

- It turns out the previous formula still works!
- Area = ½ * $\left|\begin{array}{cc} x1 & y1 \\ x2 & y2 \\ x3 & y3 \\ x4 & y4 \\ x5 & y5 \\ x1 & y1 \end{array}\right|$ $\quad -$ $\quad +$



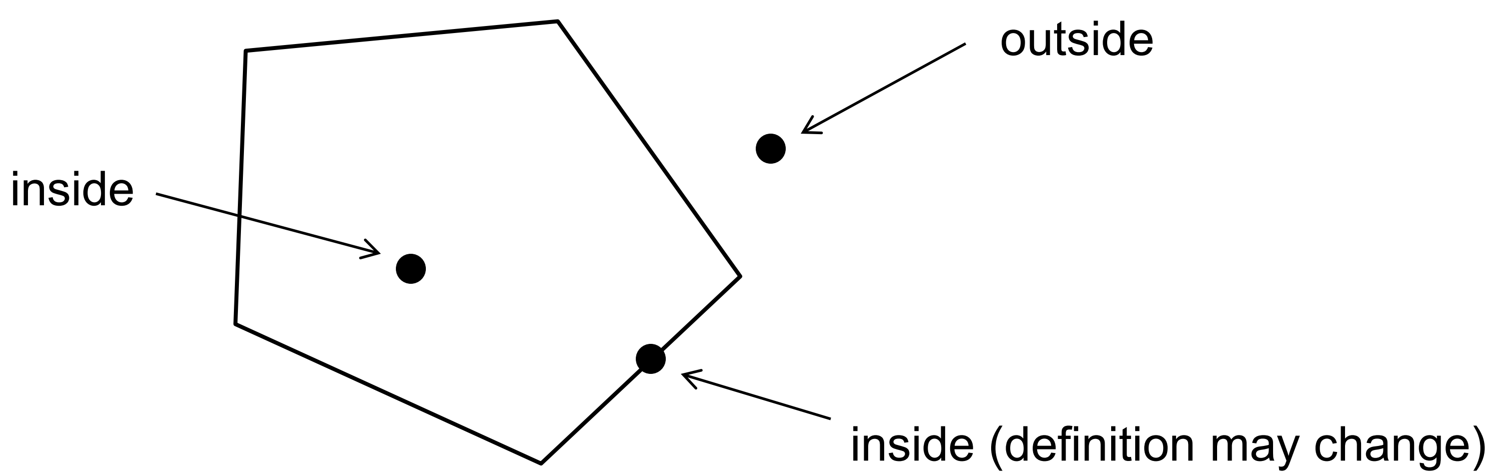
# Area of (non-convex) polygon

- Miraculously, the same formula still holds for non-convex polygons!
- Area = ½ * …
- I don't want to draw anymore



# Point inside convex polygon?

- Given a convex polygon and a point, is the point contained inside the polygon?
  - Assume the vertices are given in **counterclockwise** order for convenience

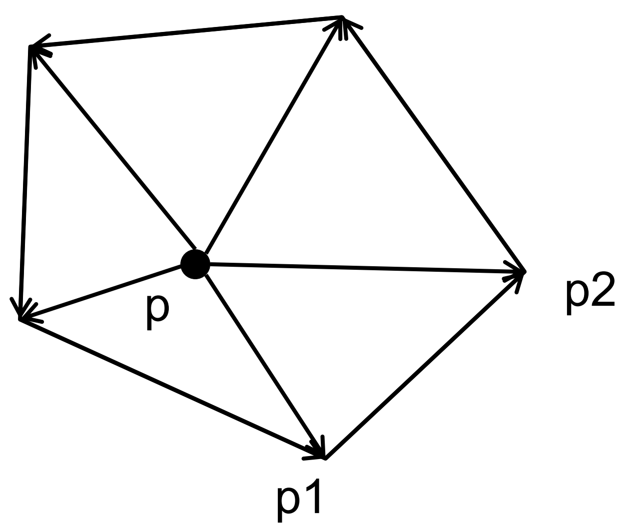


# Detour – Is polygon convex?

- A quick question – how to tell if a polygon is convex?
- Answer: It is convex if and only if every turn (at every vertex) is a left turn
  - Whether a “straight” turn is allowed depends on the problem definition
- Our crossProd function is so useful

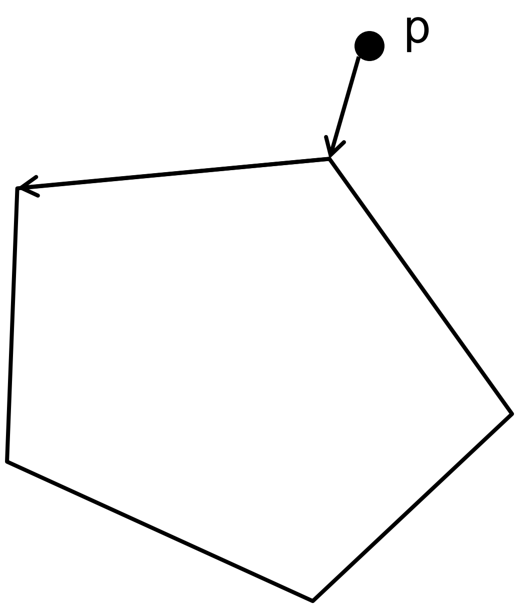
# Point inside convex polygon?

- Consider the turn p → p1 → p2
- If p does lie inside the polygon, the turn must **not** be a right turn
- Also holds for other edges (mind the directions)



# Point inside convex polygon?

- Conversely, if p was outside the polygon, there would be a right turn for some edge

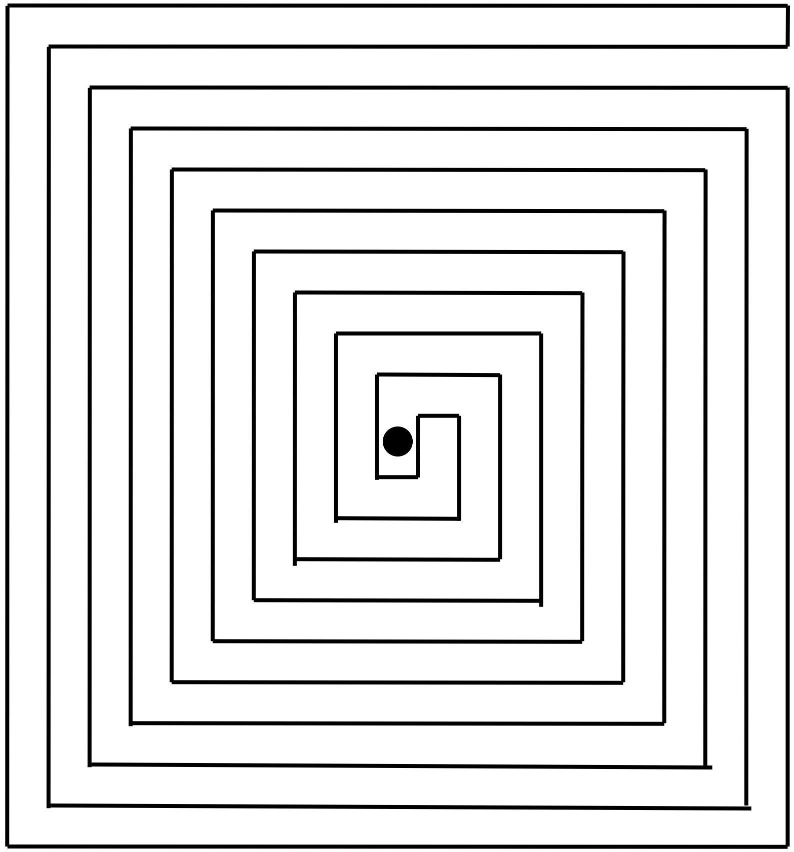


# Point inside convex polygon

- Conclusion: p is inside the polygon if and only if it makes a non-left turn for every edge (in the counterclockwise direction)

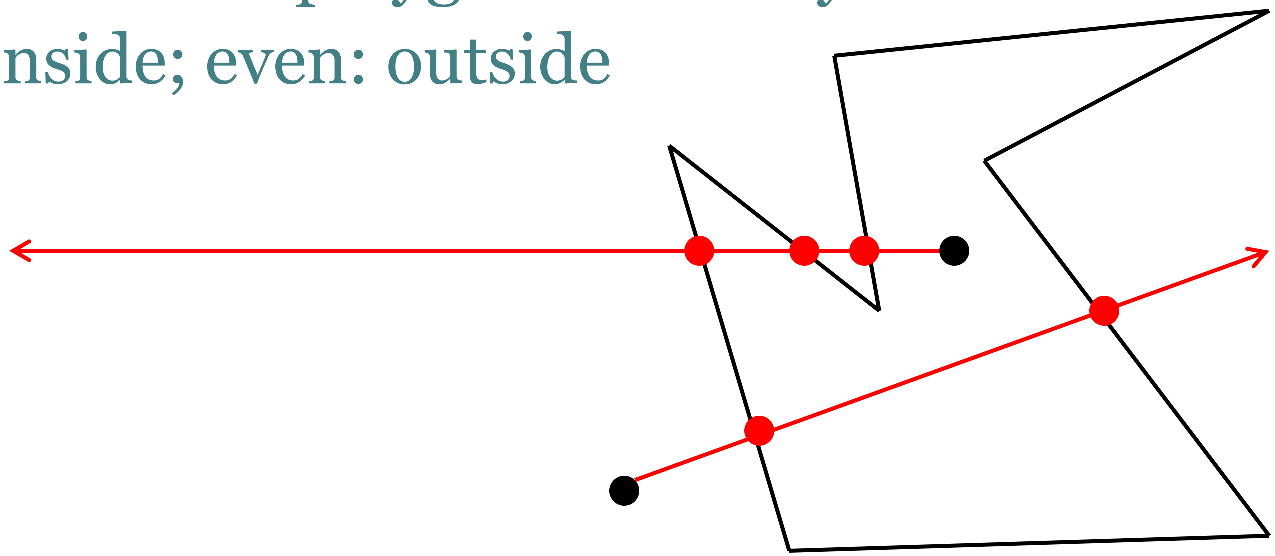
# Point inside (non-convex) polygon

- Such a pain



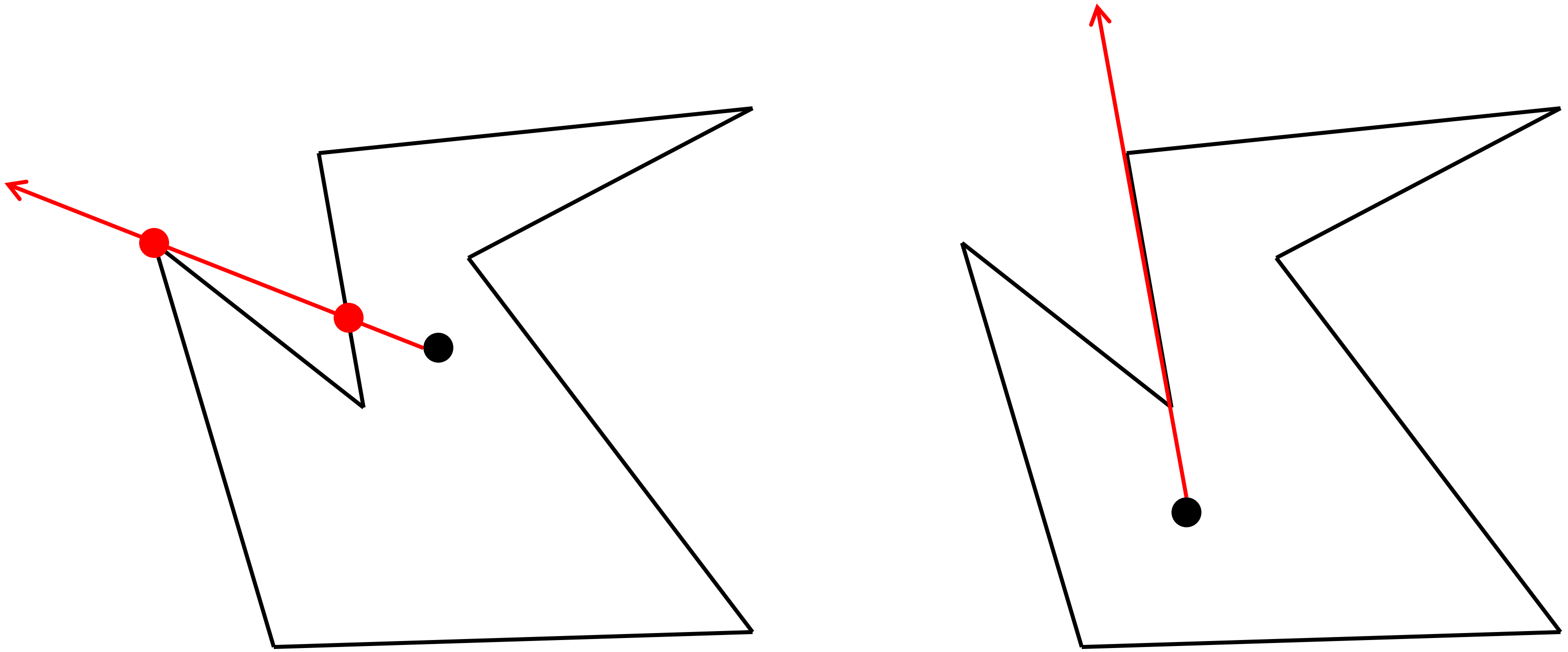
# Point inside polygon

- Ray casting algorithm
  - Cast a ray from the point along some direction
  - Count the number of times it **non-degenerately intersects** the polygon boundary
  - Odd: inside; even: outside



# Point inside polygon

- Problematic cases: Degenerate intersections



# Point inside polygon

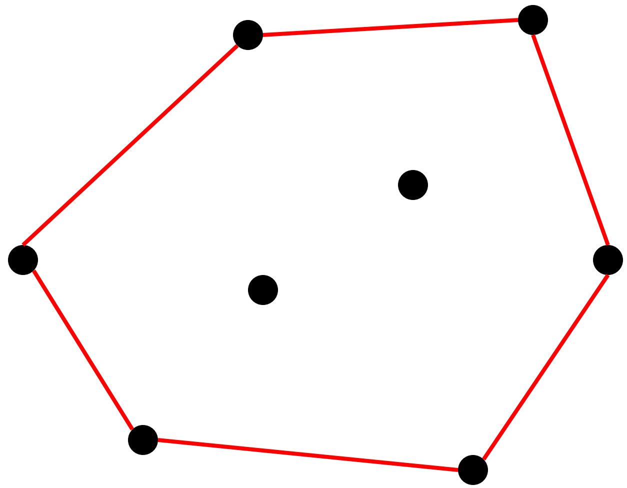
- Solution: Pick a random direction (i.e. random slope). If the ray hits a vertex of the polygon, pick a new direction. Repeat.

# Outline

- Basic operations
  - Distance, angle, etc.
  - Cross product
  - Intersection
- Polygons
  - Area
  - Containment
- Convex hull
  - Gift wrapping algorithm
  - Graham scan

# Convex hulls

- Given N distinct points on the plane, the ***convex hull*** of these points is the smallest convex polygon enclosing all of them



# Application(s) of convex hulls

- To order the vertices of a convex polygon in (counter)clockwise order
- You probably are not quite interested in real-world applications

# Gift wrapping algorithm

- Very intuitive
- Also known as Jarvis March
- Requires crossProd to compare angles
- For details, check out [Google.com]
- Time complexity: O(NH)
  - Where H is the number of points **on** the hull
  - Worst case: $O(N^2)$

# Graham scan

- Quite easy to implement
- Requires a stack
- Requires crossProd to determine turning directions
- For details, check out Google.com
- Time complexity: O(N logN)
    - This is optimal! Can you prove this?

# Circles and curves??

- Circles
  - Tangent points, circle-line intersections, circle-circle intersections, etc.
  - Usually involves equation solving
- Curves
  - Bless you

# Things you may need to know…

- Distance from point to line (segment)
- Great-circle distance
  - Latitudes, longitudes, stuff like that
- Visibility region / visibility polygon
- Sweep line algorithm
- Closest pair of points
  - Given N points, which two of these are closest to each other? A simple-minded brute force algorithm runs in $O(N^2)$. There exists a clever yet simple $O(N \log N)$ divide-and-conquer algorithm

# Practice problems

- Beginner
  - 10242 Fourth Point!!!
- Basic
  - 634 Polygon – point inside (non-convex) polygon
  - 681 Convex Hull Finding – for testing your convex hull code
- Difficult
  - 137 Polygons
  - 11338 Minefield
  - 10078 The Art Gallery
  - 10301 Rings and Glue – circles
  - 10902 Pick-up Sticks
- Expert (Regional Contest level)
  - 361 Cops and Robbers
  - 10256 The Great Divide – coding is easy though
  - 10012 How Big Is It – circles
- Challenge (World Finals level)
  - 10084 Hotter Colder
  - 10117 Nice Milk
  - 10245 The Closest Pair Problem – just for your interest
  - 11562 Hard Evidence – really hard

# References


- Wikipedia. http://www.wikipedia.org/
- Joseph O'Rourke, *Computational Geometry in C*, 2nd edition, Cambridge University Press
  - This book has most of the geometric algorithms you need for ICPC written in C code, and many topics beyond our scope as well, e.g. 3D convex hulls (which is 10 times harder than 2D hulls), triangulations, Voronoi diagrams, etc.