Two-Dimensional Geometry

Vectors

Basic Definition and Explanation

A vector is recording a relative change in position, but isn't fixed to a particular point in space. In two dimensions, a vector simply records a change in x and a change in y. For example, the vector (2, 5) represents moving in a straight line from your current position (x, y) to the position (x+2, y+5). Note that in many times, the difference between a fixed point in space and a vector isn't readily obvious and context has to be used to determine which is which. In the previous explanation, (x, y) is a fixed point and (2, 5) is a vector. Often times, we'll write the vector (2, 5) as 2i + 5j, as i is often referred to as the unit vector in the direction of the positive x axis and j is referred to as the unit vector in the positive y axis. A unit vector is simply a vector with magnitude 1. The magnitude of a vector is simply its length. Thus, the magnitude of the vector ai + bj is $\sqrt{a^2 + b^2}$.

Given two points A(x_1, y_1) and B(x_2, y_2), we can compute the vector AB as ($x_2 - x_1$, $y_2 - y_1$). This is the change from pt A to pt B. In essence this vector describes the position of B, relative to A.

Dot Product

The dot product between two vectors returns a scalar (a number). By definition, the dot product between vectors $v_1 = x_1i + y_1j$ and $v_2 = x_2i + y_2j$ is $x_1x_2 + y_1y_2$. It turns out that we can prove that this dot product is ALSO equal to $|v_1||v_2|\cos\theta$, where θ is the angle between the two vectors. In short, if we want to know the angle between two known vectors, we can easily find it via the dot product.

Example: What is the angle between u = 3i + 4j and v = 12i – 5j?

$$u \cdot v = 3(12) + 4(-5) = 16$$
$$u \cdot v = |u||v|cos\theta = \sqrt{3^2 + 4^2}\sqrt{12^2 + 5^2}cos\theta = 65cos\theta$$
$$16 = 65cos\theta, \text{ so } cos\theta = \frac{16}{65}, \text{ and } \theta = cos^{-1}\left(\frac{16}{65}\right) \sim 1.32\ radians$$

Cross Product

The cross product between two vectors returns another vector. By definition, it returns a vector perpendicular to both input vectors with a magnitude equal to the area of the parallelogram defined by both vectors. When we are dealing with 2D geometry, the direction of the cross product is always in the positive or negative z-axis. Thus, this cross product is always (0, 0, z). If z > 0, the vector points in the positive z-axis. If z < 0, then it points in the negative z-axis. For two dimensions, with vectors $v_1 = x_1i + y_1j$ and $v_2 = x_2i + y_2j$, the cross product is (0, 0, $x_1y_2 - x_2y_1$). Note that this is also written as $0i + 0j + (x_1y_2 - x_2y_1)k$, since k is the name given to the unit vector in the direction of the positive z-axis. Geometrically, a positive value for the k coefficient of the dot product means that the second vector is counter-clockwise from the first vector by an angle less than π radians. Often times in physics, this is referred to as the "right-hand rule". Basically, in determining the direction of the cross product, you use your right hand, sweeping across from the first vector towards the second and the direction of your thumb is the direction of the resultant

vector. We can use the cross product to determine the signed area of a triangle (the sign just tells you which direction you're going in, from side to side), as well as testing if three points are collinear.

Example 1: What is the area of the triangle formed by the following three points: (3, 7), (12, -8), and (9, 16)?

> Create two vectors from (3, 7) to the other two points. These are
> $9i - 15j$ and $6i + 9j$. The cross product of these is $(9(9) - (-15)(6))k = 171k$.
> The absolute value of the k coefficient is the area of the defined parallelogram.
> Thus, the area of the defined triangle is just half of this, or 85.5.

Example 2: Are the following three points collinear: (3, 8), (19, 40), and (-2, -2)?

> Create two vectors from (3, 8) to the other two points. These are
> $16i + 32j$ and $-5i - 10j$. Their cross product is $16(-10) - (32)(-5) = 0$.
> The points are collinear because sin of the angle between the vectors
> is 0, consequently that angle is either 0 or π radians!

Line Equation in Two Dimensions

Now that we have a basis in vectors (sorry, bad joke…), we can define the vector equation of a line, which tends to be better to use for programming contest problems than Cartesian equations for lines. A line is defined by a point and a direction. Thus, the vector equation of a line looks like this:

$$r = p_0 + \lambda v$$

where p_0 is a point on the line and v is the directional vector in the direction of the line. Lambda is a parameter such that for any point on the line, there exists a value of lambda that, when plugged in, sets r equal to that point. Consider the following equation of a line:

$$r = (3, -5) + \lambda(2, 1)$$

This essentially means that (3, -5) is on the line and the direction of movement on the line is (2, 1). So, to reach an arbitrary point on the line, start at (3, -5) and move in the direction (2, 1) as far as you want. Logically, this means we can set of a pair of equations solving for the x and y coordinates of any point on the line in terms of lambda:

$$x = 3 + 2\lambda$$
$$y = -5 + 1\lambda$$

These equations are known as parametric equations for the line. For example, if we plug in $\lambda = 3$, we obtain $x = 9$ and $y = -2$. This means that (9, -2) is on the line as well. Each unique value we

plug in for lambda will produce a different point on the line. For each point on the line, there's a unique value of lambda that creates it.

Line Intersection in Two Dimensions

Now that we know how to express a line in both vector and parametric equations, we can learn how to find the intersection of two lines as follows:

Write out each vector equation. Below r_1 is a line containing (x_1, y_1) in the direction (u_x, u_y) and r_2 is a line containing (x_2, y_2) in the direction (v_x, v_y). Note that we use different parameters because simultaneously, we may care about points on r_1 and r_2 that are created with different parameter values. The only time we would use the same parameters if the parameter represented a time and both equations described the movement of objects in time and the value of the parameter produced the location of each particle at that point in time.

$$r_1 = (x_1, y_1) + \lambda(u_x, u_y)$$
$$r_2 = (x_2, y_2) + \mu(v_x, v_y)$$

Parametrically, for both equations we get:

$$x = x_1 + u_x\lambda \qquad x = x_2 + v_x\mu$$
$$y = y_1 + u_y\lambda \qquad y = y_2 + v_y\mu$$

In order for these lines to intersect, there must exist a value for λ and a value for μ, that when plugged into both equations, produces the same point. Thus, for any intersection point, we must have:

$$x_1 + u_x\lambda = x_2 + v_x\mu$$
$$y_1 + u_y\lambda = y_2 + v_y\mu$$

The only unknown quantities are λ and μ. We can rewrite this as a typical pair of linear equations in two variables as follows:

$$u_x\lambda - v_x\mu = x_2 - x_1$$
$$u_y\lambda - v_y\mu = y_2 - y_1$$

By hand, we use many different techniques to solve a system like this and choose the one that minimizes arithmetic. In code, it's probably best to use Kramer's Rule. If our system has a unique solution and is

$$ax + by = c$$
$$dx + ey = f$$

then are solutions are: $x = \frac{\begin{vmatrix} c & b \\ f & e \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}}$ and $y = \frac{\begin{vmatrix} a & c \\ d & f \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}}$. Recall that absolute value bars on the matrices indicates a determinant and for the 2 x 2 case, we have $\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$.

If the denominator of these expressions is zero, this means the system does NOT have unique solutions. Specifically, either the lines are coincidental (describing the same line), or they are parallel and non-intersecting. To determine which is the case, pick two points on one line and one line on the other, and test to see if they are collinear or not.

Line Segment Intersection in Two Dimensions

Do all the work described for regular line intersection and make sure that the vector of movement is the vector of movement from the start point to the end point of the line segment. (You can arbitrarily choose the start and end point unless the problem given specifies a direction of motion where the time variable is meaningful.)

We can quickly see that the intersection point is on one segment if the corresponding parameter of the solution is in between 0 and 1, inclusive. Thus, for the segments to actually intersect, BOTH parameters must be in between 0 and 1 inclusive.

Finally, we have the tricky case of line segments both on coincidental lines. We must take one line and test both endpoints of the other line against the first line. If both endpoints correspond to a parameter greater than 1 or less than 0 on the first line, then there's no intersection. Otherwise, there is.

Example: Do the line segments (3, 9) to (18, -6) and (-4, 16) to (2, 10) intersect?

In solving the appropriate set of equations, we find that the lines are coincidental. Now, our first line, written parametrically is:

$$x = 3 + 15\lambda$$
$$y = 9 - 15\lambda$$

Find the value of λ that corresponds to the point (-4, 16): Just set $3 + 15\lambda = -4$, yielding $\lambda = -7/15$.
Find the value of λ that corresponds to the point (2, 10): Just set $3 + 15\lambda = 2m$ yielding $\lambda = -1/15$.

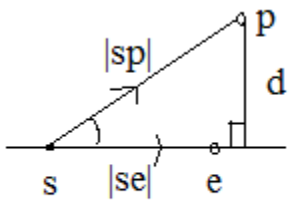
Since both of these are less than 0, there is no intersection.

If we move the second point of the second segment to be (4, 8), then its $\lambda = 1/15$ and there would be an intersection.

Alternatively, if we move the second point of the second segment to be (20, -8), then its $\lambda = 17/15$ would still indicate an intersection because both values aren't either less than 0 or greater than 1.

Point Line Distance

Let the line be formed with points s and e, so the direction vector of the line is e - s, and let the point in question be the point p. Draw a vector from s to p. Then label the angle formed by points p, s and e to be theta.



sin(theta)=d/|sp|

The desired distance is the line drawn from p down to se that is perpendicular to se. This line segment is also the height of a right triangle with points s and p. Label the desired distance d, then $sin\theta = \frac{d}{|sp|}$, so the desired distance is $d = |sp|sin\theta$.

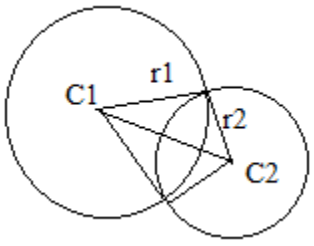
But, notice that this looks very similar to the magnitude of a cross product of the vectors se and sp. In fact, the only thing missing in it is the magnitude of se. Recall that $|sp \times se| = |sp||se|sin\theta$. Dividing this quantity by |se| yields the expression above.

It follows that the distance between a point p, and a line defined by the points s and e is:

$$d = \frac{|sp||se|sin\theta}{|se|} = \frac{|sp \times se|}{|se|}$$

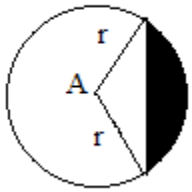
Circle-Circle Intersection

Let circles C_1 and C_2 have radii r_1 and r_2, respectively. If the distance between C_1 and C_2 exceeds $r_1 + r_2$, there is no intersection. If this distance is equal, then there's one intersection, which is on both circles and the line segment between both centers. If this distance is less then we have two intersections. Here is a picture:



Since we know r_1 and r_2 and the distance between the two centers, we know all three of sides of the two congruent triangles in the picture. This allows for us to solve for all three angles in these triangles. Using the atan2 function, we can get the directional angle from C_1 to C_2, add/subtract to it the appropriate angle in the triangle to get the new directional angle from C_1 to each intersection point. We can move along each of these vectors a distance of r_1 to get the solution points.

If we want to fine the area of that "eye-shaped" region, we can calculate the sum of two sector minus triangle calculations. Here is an illustration of one such calculation:



The area of the sector is $\frac{A}{2}r^2$ while the area of the triangle is $\frac{1}{2}r^2 sin\ A$. Thus, the shaded area is the difference of these two: $\frac{r^2}{2}(A - sinA)$. This difference of area also cleverly proves that A > sin A for $0 < A < \pi$. Note that once we know angle A, we can compute the length of the chord on the circle using the law of cosines. In particular, this length is equal to $r\sqrt{2(1 - cosA)}$.

Circle-Line Intersection

Given an equation of a circle $(x - c_x)^2 + (y - c_y)^2 = r^2$ and a line in parametric form with the equation

$$x = x_1 + \lambda d_x$$
$$y = y_1 + \lambda d_y$$

we can simply plug in our arbitrary expression for x and y for any point on the line into the circle equation:

$$(x_1 + \lambda d_x - c_x)^2 + (y_1 + \lambda d_y - c_y)^2 = r^2$$

Noting that the only thing unknown in this equation is λ, we see that once we simplify, we can rewrite this equation as a quadratic in λ:

$$(d_x^2 + d_y^2)\lambda^2 - 2(\ d_x(x_1 - c_x) + d_y(y_1 - c_y))\lambda + (x_1 - c_x)^2 + (y_1 - c_y)^2 - r^2 = 0$$

If this equation has no solutions, there is no intersection. If it has one solution there is one solution, which is a point of tangency, otherwise, there are two solutions. Once we get the appropriate values of λ, we can plug these back into the line equation to get the corresponding points of intersection on the line with the circle. If our input line was a segment, we would have check to see if either parameter for the intersection was in between 0 and 1.

Polygon Area

The area of a polygon can be summed up as the signed area of several triangles, each of which we add using the magnitude of the cross product (discussed earlier). Let the vertices of a polygon in counter-clockwise order be (x_0, y_0), (x_1, y_1), … (x_{n-1}, y_{n-1}). Imagine forming triangles with the origin to each pair of vertices of the form (x_i, x_{i+1}), as well as (x_{n-1}, x_0). The signed area of each of these triangles precisely equals the area enclosed by the polygon. Adding these magnitudes of cross products (and dividing by two since we're dealing with triangles), we get the desired formula $\frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{(i+1)\%n} - x_{(i+1)\%n} y_i)$, for the area of a polygon.

Pick's Theorem

If we are given a polygon with vertices on lattice points (integer coordinates), there's a nice relationship between the number of lattice points on the boundary of the polyon, the interior of the polygon and the area of the polygon. The formula is as follows: $A = I + \frac{B}{2} - 1$, where A is the area of the polygon, B is the number of boundary lattice points, and I is the number of interior lattice points. Since we know how to find polygon area (above), this formula is useful for determine the number of lattice points in the interior of a polygon.

Point in Polygon

Given a not necessarily convex polygon, one method to determine if a point is in a polygon is to shoot a ray from the point in an arbitrary direction (better if it's not horizontal or vertical) and count the number of times the ray crosses one of the line segments of the polygon. If this number is odd, the point is in the polygon, if it's even it's not in the polygon. The key to this test is that you don't want one of the points the ray intersects to be an actual end point of any edge, since accidentally this will count twice. This is the reason for the arbitrary direction. The run time of this is O(n), where n is the number of vertices/sides to the polygon.

Alternatively, what we can do if the vertices (in clockwise order) are $v_1v_2\ldots v_n$ and the point in question is p, we can calculate the angles v_1pv_2, v_2pv_3, v_3pv_4, …, v_npv_1, and add up all of these angles. If this sum is equal to 2π, then the point is in the polygon. Otherwise it's outside the polygon. (Note: when we check for equality with doubles, we'll check to see if our answer is within a tolerance of what we expect. More details are given below in the Note about Precision.) If these angles add up to something less, then the point is outside of the polygon. The angles should be signed angles.

Notes about Solving Triangles

When solving a triangle, law of cosines yields an unambiguous answer always. If you can't use it, keep in mind that law of sines has an ambiguous case. In particular, if you solve an equation and get sin x = 0.5,where x is an angle in a triangle, x could be 30 degrees or 150 degree. Sometimes you can rule out 150 if one of the other angles in the triangle is greater than 30. Other times, you can't. Know a couple of the different formulas for the area of a triangle. (Some of these are $Area = \frac{1}{2}bh$, $Area = \frac{1}{2}bcsin(A)$, $Area = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = \frac{a+b+c}{2}$.)

Note about Precision

Many geometry problems ask contestants to solve for quantities that are non-integral, or have intermediate calculations that are necessarily important and non-integral. When dealing with doubles, it's extremely important to take care with precision issues. Whenever checking for equality between doubles, always use a tolerance. For example,

```
if (Math.abs(a-b) < 1e-9)
```

is preferable to

```
if (a == b)
```

Even in questions that ask for output rounded to some number of decimal places, due to the nature of numbers, it's often the case that a test case could have a result of 0.175 exactly. If the output is to two decimal places, if you have 0.174999999998 stored, when printed to 2 decimal places, 0.17 will print instead of the correct 0.18. Thus, it's typical, before printing a rounded value to add a small tolerance to it. The reason this doesn't usually change right answers to wrong answers is that it's very difficult to make a test case where the correct answer is 0.17499999998, but easy to make one where the right answer is 0.175.

Efficient Convex Hull Algorithm: Graham Scan

1. Find point with minimum y coordinate. If there's a tie, break the tie with minimum x coordinate. (Takes O(n) time for n points.) Call this the "starting point."

2. Sort the rest of the points based on angle from the starting point.

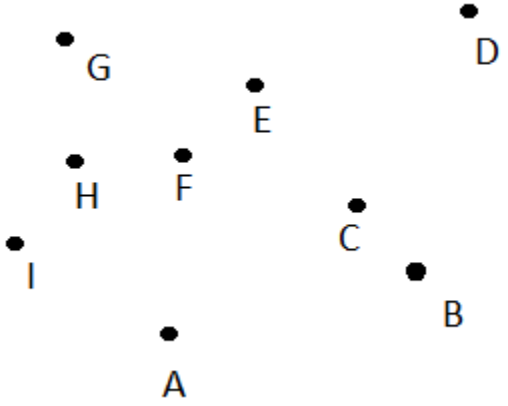
3. Maintain a stack of points, which starts empty. This stack will store the convex hull. Add to it the starting point and the next point in the array.

4. For each subsequent point (from index 2 on…) do the following: (This runs in O(n) time since no point can be involved in more than two stack operations and there are only n points.)

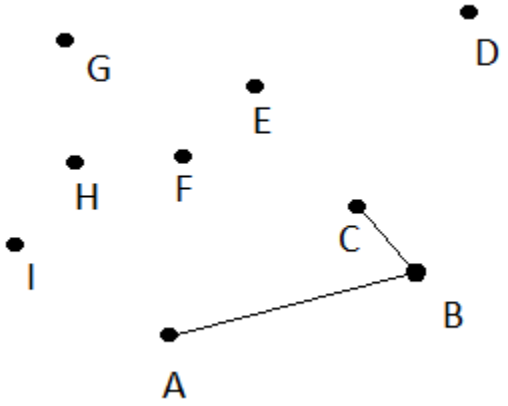
    a. Determine the angle formed with the last two points on the stack and the current point.
    b. While this angle represents a right turn, pop the top point off the stack.
    c. Push the current point onto the stack.

5. The items in the stack after we consider all points is the convex hull, in the order they are in the stack.

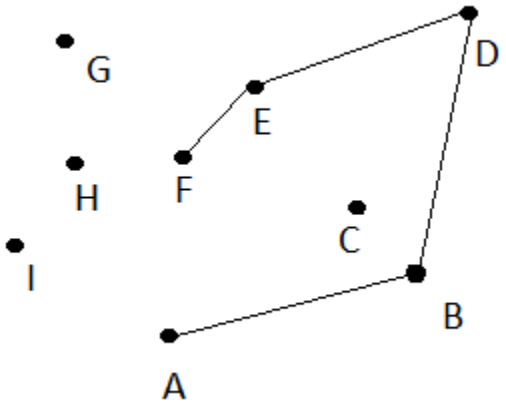
Consider this example:



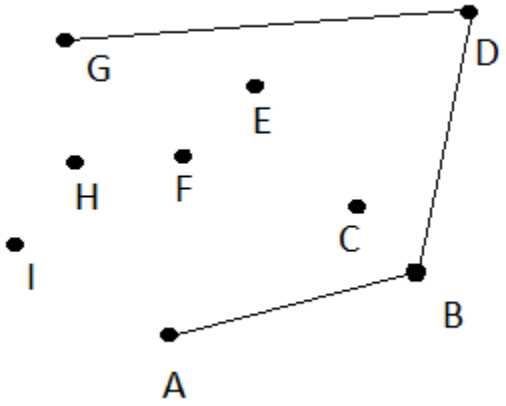
The points are already sorted in alphabetical order based on the criteria given. Point A is the starting point. Initially, the stack contains A, B and C and our hull looks like:



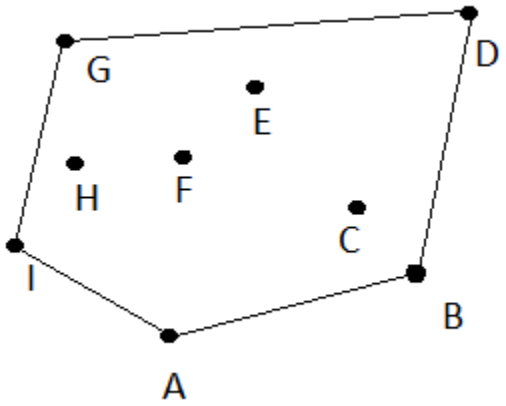
Now, when we consider point D, we see that BCD is a right turn, not a left turn. So, we pop off C from the stack and then look at the turn ABD, since AB are the top two items in the stack. This is a left turn, so our hull changes to ABD. Fast forward and we get to when F gets added to the hull and our picture looks like:



Now, let's consider G. The turn EFG is a right hand turn, so we pop F off the stack. The turn DEG is ALSO a right hand turn, so we pop E off the stack. Finally, we find that the turn BDG is a left hand turn, so our hull is as follows:

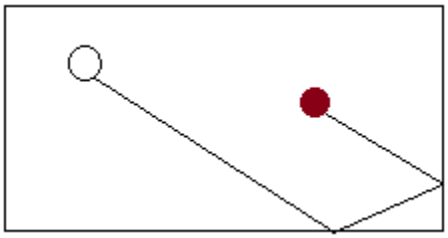


In the second to last iteration, H will get added onto the stack. In the last iteration, H gets popped off the stack and then I gets pushed onto it. Then, the hull is completed by adding the edge IA:

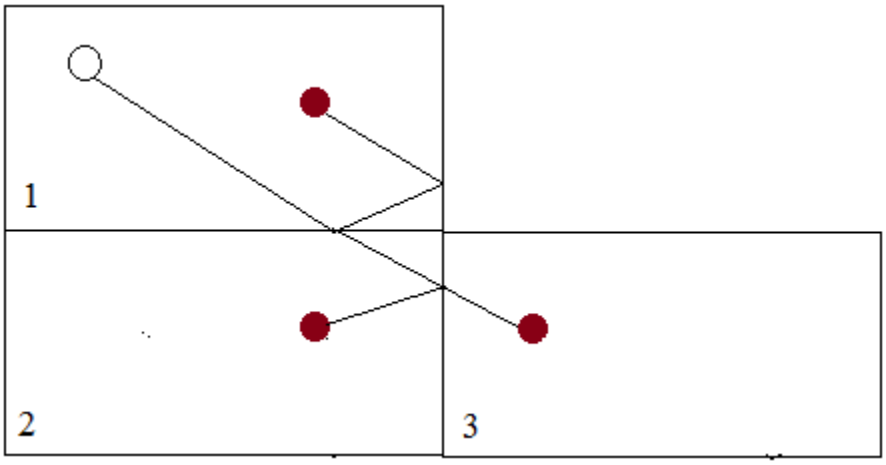


Reflection Trick - Pool Table

Some geometry problems deal with calculating reflections of an object bouncing around some sort of area, and then trying to determine if a collision will occur, after some number of bounces. While we can simulate such an action using some math, it turns out to be much easier to solve these sorts of problems by "reflecting" the area over each boundary line. Consider the 2009 South East Regional Problem Pool Table. In this problem you are given the location of a cue ball and a target ball on a rectangular pool table, as well as the number of times the pool cue must bounce off the walls before hitting the target ball. Given this information, you have to calculate the *minimum* distance the cue ball has to travel to satisfy the requirements. Consider the following picture of a cue bouncing twice and then hitting a target ball:



Another way of looking at this path is to reflect the table along each boundary wall that it bounces. Consider this picture:



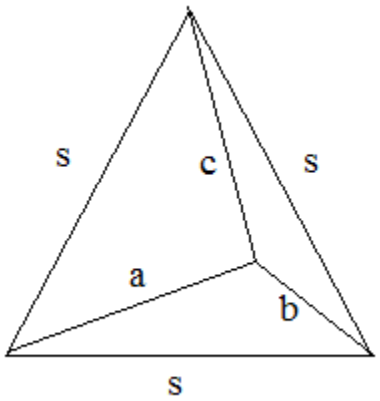
We start in frame one. Then the first bounce is just like what happens in frame two, if we reflect the pool table over the bottom horizontal line. Then the second bounce is shown in frame 3, and this is a picture of the reflection of frame 2 over the right vertical line. What we see here is calculating where to shoot the ball to hit the target in two bounces is just line calculating the straight line to the target ball in frame 3, where we place it in frame 3 by doing the appropriate reflections show. So, to solve this problem, we do every possible set of k reflections (where we need k bounces) and reflect the target ball in these locations. (These "mirror" pool tables form a diamond similar to where you can get if you travel exactly k steps via Manhattan distance via unit movements.) Then, just calculate the shortest distance to any of these reflected balls…EXCEPT when on the path to a reflected ball, a previous reflected ball appears!!! (What this means is that the cue ball would have hit the target earlier than k bounces.)

Binary, Ternary Searches - Geometry

Often times, even if a straight mathematical formula is difficult to derive for a geometric figure, it's much easier to run a binary (or ternary) search to determine a length or area. Consider the following problem called "Carpet" from the 2014 South East Regional:

Problem #5: Carpet (Binary search with geometry)

The problem is you are given 3 lengths a, b, and c. They represent the distance from S in any angle or direction. Find the length, s, such that you can form an equilateral triangle and from some point in the triangle the distance from all 3 points of the triangle is a, b, and c. Here is a diagram:



Since we are given a, b and c, one thing we might imagine is seeing if we can check if a given value L is too big or too small for our equilateral triangle.

We know that law of cosines state that c2 = a2 + b2 - 2abCos(C) where a, b, c are legs of the triangle, unrelated to the above mentioned a, b, and c, and C is the angle formed at the point opposite of the leg c.

So to determine if a current leg length is too big or too small we can determine this by summing up the inner angle formed by the legs using the law of cosine. If the angle sum is less then 360 degrees then we need longer legs to increase the angle sum, if the angle sum is more then 360 we need smaller legs.

There is a small problem we need to consider when using acos in java. It returns a range of 0 to PI. So we have to check if the length of the 3 legs qualify as being a triangle.