

CHAPTER 1

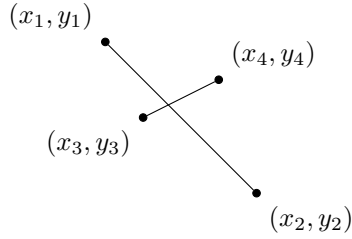
GEOMETRIC ALGORITHMS

1.1 Intersection of Line Segments

In this section, we will look at algorithms to determine whether two line segments intersect, and the geometric interpretation of the algorithm. First, we consider some geometric objects:

- a *point* in 2-dimension is given by (x, y) ;
- a *line* is an (ordered) pair of points p and q , and is denoted by $-p - q-$;
- a *line segment* is a pair of end-points p and q , and is denoted by $p - q$;

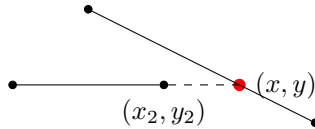
Now, we consider when two line segments p_1 and p_2 intersect- we do not need to compute the point of intersection. This is possible to do using high school mathematics. For instance, let the line segments be the following:



Then, the equations of the two lines are:

$$\begin{aligned}(x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1) &= 0, \\ (x_4 - x_3)(y - y_3) - (y_4 - y_3)(x - x_3) &= 0.\end{aligned}$$

We then find a point of intersection (p, q) by solving the two equations simultaneously, if possible¹. Finally, to check that the intersection of the two lines is within the line segments, we check that $x_1 \leq p \leq x_2$ and $x_3 \leq p \leq x_4$. The following figure illustrates why this is required:



Using the equation of the two lines, we will find that (x, y) is a solution to the equation. However, we have $x > x_2$, so the intersection does not lie on the line segment with (x_2, y_2) .

Although possible, this is not the most optimal way of solving the problem. The rearrangement to find the point of intersection involves division, which is

¹There are two cases when we won't be able to solve this system of equations easily- if the lines are parallel and have different y -offset, then they cannot intersect; if the two lines have the same y -offset, then they can intersect if the two segments overlap (i.e. $x_1 \leq x_3 \leq x_2$ or $x_1 \leq x_4 \leq x_2$, or vice versa).

susceptible to floating point errors. Moreover, we find the point of intersection even though we just need to find whether the line segments intersect- we are doing more work than required.

We will now look at a more efficient algorithm by breaking the problem down. If we have two line segments $p - q$ and $r - s$, then:

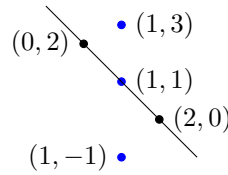
- we first determine whether the two points p and q lie on different sides of the line $r - s$, and vice versa (`onOppositeSides`); and
- we then determine whether the smallest rectangles containing the lines $p - q$ and $r - s$ intersect (`boundingBox`).

The line segments intersect if and only if all the tests pass.

The algorithm `onOppositeSides` takes in two points a, b and a line $p1 - p2$, and determines whether a and b lie on opposite sides of the line. The equation of the line is given by

$$(l.p2.x - l.p1.x)(y - l.p1.y) - (l.p2.y - l.p1.y)(x - l.p1.x) = 0.$$

When we substitute (x, y) as the points a and b , if the value at the LHS of the equation have different signs, then the two points lie on opposite sides. To see this, consider the following example:



Then, the equation of the line is $y + x - 2 = 0$. We can substitute the three blue points to the left hand side of the equation to get a value:

- if $(x, y) = (1, 3)$, then $y + x - 2 = 2 > 0$;
- if $(x, y) = (1, 1)$, then $y + x - 2 = 0$;
- if $(x, y) = (1, -1)$, then $y + x - 2 = -2 < 0$.

In the figure above, all 3 pairs of points are considered to be on opposite sides- we consider $(1, 1)$ to be both above and below the line since it lies on the line. An easy way to check this is if the product of the two values to be non-negative. Note that we are meant to accept the case when both the lines are $(1, 1)$ - this corresponds to the trivial line segment that intersects the line given at $(1, 1)$.

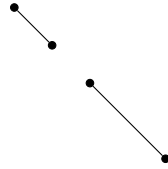
The algorithm `onOppositeSides` is given below.

```

1 bool onOppositeSides(Line line, Point a, Point b) {
2     double v1 = (line.p2.x - line.p1.x) * (a.y - line.p1.y) -
3               (line.p2.y - line.p1.y) * (a.x - line.p1.x);
4     double v2 = (line.p2.x - line.p1.x) * (b.y - line.p1.y) -
5               (line.p2.y - line.p1.y) * (b.x - line.p1.x);
6     return v1 * v2 <= 0;
7 }

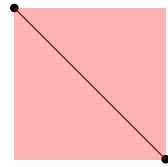
```

The condition `onOppositeSides` is not enough to check that the two line segments intersect. For example, consider the following line segments:



Since the two line segments are part of the same line, the result of `onOppositeSides` in both cases will be `true`. But, since the two line segments do not overlap, they do not intersect. For this reason, we require a third check- the bounding box.

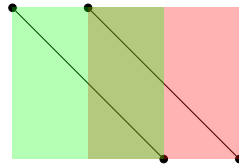
The *bounding box* of a line segment $p - q$ is the smallest rectangle that contains the line segment $p - q$ whose sides are parallel to the x and the y -axes. For example, below is a line segment and its bounding box:



For two line segments to intersect, we require the bounding boxes to intersect. We illustrate this in the examples below:

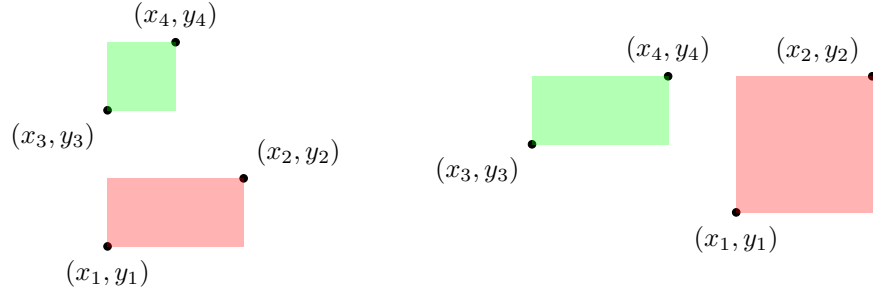


We see that when the two line segments intersect, their bounding boxes intersect. Also, if the bounding boxes do not intersect, then the two lines cannot intersect. Nonetheless, it is possible for two line segments to intersect and there to still not be an intersection- consider the following example.



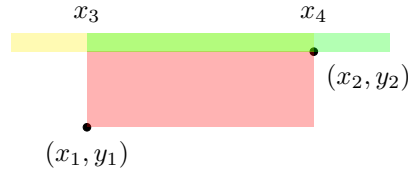
In this case, although the two lines do not intersect, their bounding boxes still intersect. The issue here is that in both line segments, the points are both on the same side of the other line segment.

Now, we consider how to define the `boundingBox` algorithm. The problem is essentially about when two rectangles intersect. So, consider the following two pairs of rectangles:



The rectangles on the left do not intersect since $y_3 > y_2$ and $x_3 > x_2$. On the other hand, the rectangles on the right do not intersect since $x_1 > x_4$ and $y_1 > y_4$. These give us four potential conditions required for rectangles to intersect- $x_3 \leq x_2, y_3 \leq y_2, x_1 \leq x_4, y_1 \leq y_4$.

Assume now we fix one of the rectangles and consider what happens when the x -coordinates satisfy this condition:



The value x_3 comes to the left of x_2 , while the value x_4 comes to the right of x_1 . In particular, x_3 can be at any point within the yellow region, and x_4 on the green region. This ensures that the two rectangles do intersect with respect to the x -coordinate. So, if we further require $y_3 \leq y_2$ and $y_1 \leq y_4$, the rectangles must intersect in both x - and y -coordinate, which means that they intersect in 2D. Therefore, these 4 conditions are sufficient. Hence, the algorithm `boundingBox` is the following.

```

1 bool boundingBox(LineSegment l1, LineSegment l2) {
2     Rectangle r1 = new Rectangle(l1);
3     Rectangle r2 = new Rectangle(l2);
4
5     return r2.bottomLeft.x <= r1.topRight.x &&
6           r1.bottomLeft.x <= r2.topRight.x &&
7           r2.bottomLeft.y <= r1.topRight.y &&
8           r1.bottomLeft.y <= r2.topRight.y;
9 }

```

Putting this all together, we find that for line segments $p - q$ and $r - s$ intersect if and only if

$$\text{onOppositeSides}(p, q, r-s) \wedge \text{onOppositeSides}(r, s, p-q) \\ \wedge \text{boundingBox}(p-q, r-s).$$

If two line segments intersect, then we know that for each of the line segment, the two points lie on opposite sides of the other one, and that their bounding boxes must intersect. So, this is a necessary condition for the line segments to intersect. Moreover, if two line segments do not intersect and their bounding

boxes intersect, then one of them must be below the other, meaning that it is not on opposite sides- these conditions are sufficient as well.

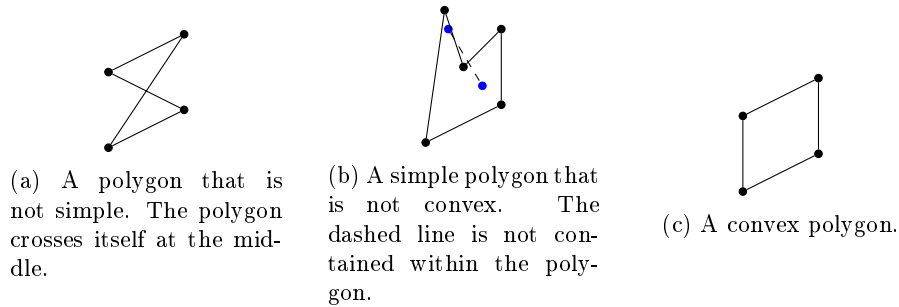
Both the methods `onOppositeSides` and `boundingBox` are $O(1)$, so the intersection algorithm is also $O(1)$. Moreover, the algorithm does not involve division.

1.2 Simple Polygons and Convex Hulls

In this section, we will look at constructing different polygons given a set of points. These polygons we construct will have various applications that we will consider later. First, we consider some further geometric objects:

- a *path* is a sequence of distinct points p_1 to p_n .
- a *polygon* is a path where $p_1 = p_n$ (i.e. p_1 to p_{n-1} are distinct, but $p_1 = p_n$);
- a *simple polygon* is a polygon which does not intersect itself- it is characterised by partitioning the space into its interior and exterior;
- a *convex polygon* is a simple polygon where for any two points p and q inside the polygon, the line segment $p - q$ is also inside it.

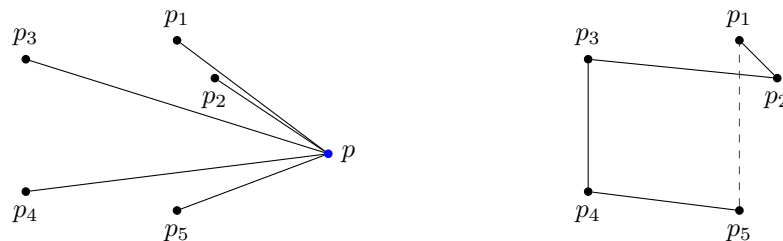
Below are some polygons that illustrate simple and convex polygons.



Simple Polygon

We will now consider how to construct a simple polygon given a set of points. In the example above, we saw that it is not possible to construct a simple polygon by arbitrarily joining the set of points. So, we will need to reorder the points in the set so that the resulting set of points can be joined together pairwise to get a simple polygon.

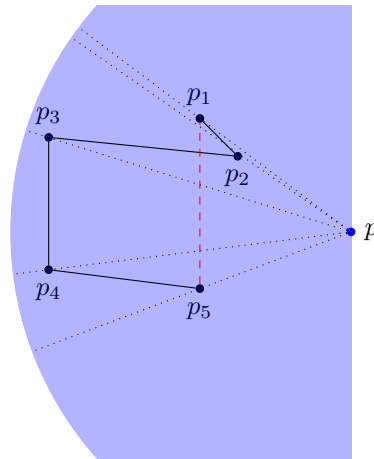
To do so, we can choose a point on the plane as a pivot. Then, we scan the set of points in an anti-clockwise manner from the pivot to find the order in which we should construct the simple polygon. We illustrate this with an example.



On the left, we use the pivot p to scan and label the points going in an anti-clockwise fashion. That way, when we join the points $p_1 - p_2 - \dots - p_5$, we do

not have any self-intersection. However, when we join p_5 and p_1 , we do get a self-intersection.

We can understand why there is a self-intersection using the figure below.



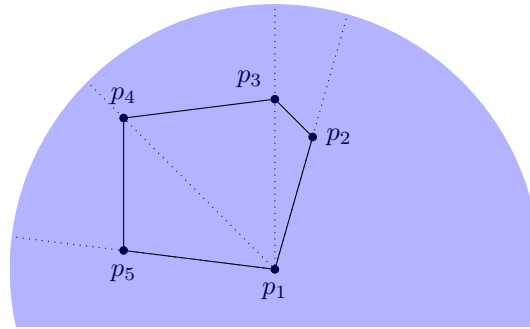
The image represents a semi-circle centered at the pivot p . When we label each point, we partition the semi-circle into different sectors. And, when we connect two points, we go from one sector to another. If we exclude the last connection, there is only one line from one sector to another.

To avoid the intersection, we should choose the pivot as one of the points that we were given. The pivot is an extreme point- one way to choose the point would be to take the one with the highest x -coordinate (and lowest y -coordinate, if necessary). We first illustrate this with an example.



Here, the pivot was p_1 . Using this pivot, we labelled the other points depending on when we encountered it with respect to the pivot, turning anti-clockwise. As we can see, this does not lead to a self-intersection at the end.

We will now use a similar diagram to above to show that this method always works.



Like before, we do not expect there to be an intersect going from a point to a non-final point. Moreover, at the end, we are traveling along the radius- this ensures that we do not move to another sector and get a self-intersection.

Next, we consider the algorithm. To label the points, we sort them with respect to the angle it makes with the pivot- the smaller the angle, the earlier it is placed. The following is the algorithm:

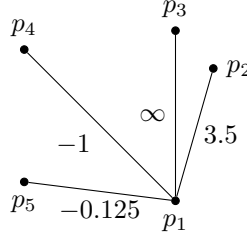
```

1 void simplePolygon(List<Point> points) {
2     // find the pivot
3     int i = 0;
4     Point pivot = points[0];
5     for (int j = 1; j < points.length; j++) {
6         // Change the pivot if this point has x-coordinate bigger
7         // or if this point has the same x-coordinate, and the
8         // y-coordinate is smaller.
9         if (pivot.x < points[j].x ||
10            (pivot.x == points[j].x && pivot.y < points[j].y)) {
11             pivot = points[j];
12             i = j;
13         }
14     }
15
16     // move the pivot to the start
17     points[i] = points[0];
18     points[0] = pivot;
19
20     Map<Point, double> angles = {};
21     Map<Point, double> distances = {};
22
23     // sort the points wrt angle it makes with the pivot
24     points.sort((p1, p2) {
25         double angle1 = angles[p1] ??
26             atan2(pivot.x-p1.x, pivot.y-p1.y);
27         angles[p1] = angle1;
28
29         double angle2 = angles[p2] ??
30             atan2(pivot.x - p2.x, pivot.y - p2.y);
31         angles[p2] = angle2;
32
33         return angle1.compareTo(angle2);
34     });
35 }

```

The function `atan2` takes in the numerator and the denominator of the `arctan` function as two different arguments and gives the right `arctan` value, e.g. if the denominator is 0 or if the angle is greater than π .

We now consider the complexity of the algorithm- if there are n points, then the loop to compute angles is $O(n)$, and the sorting is $O(n \log n)$. Therefore, the total complexity is $O(n \log n)$. So, this is a very efficient algorithm. It does use the `arctan` function, which is considered expensive. Moreover, there is no reason to explicitly compute the angles. We will now illustrate how to label the points without explicitly compute the arctangent value. So, consider the previous pivot labelling:



The labelled numbers are the gradients of the line going between p_1 and p_i for $2 \leq i \leq 5$. So, given two gradients m_1 and m_2 , we compare them as follows:

- if m_1 and m_2 have different signs, then the positive one comes first;
- if m_1 and m_2 have the same sign, then the smaller one comes first.

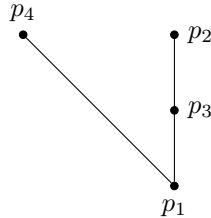
Using this pattern, we can sort the values by just comparing the gradients. This can be done without any division too, i.e. if we have

$$m_1 = \frac{dy_1}{dx_1}, \quad m_2 = \frac{dy_2}{dx_2},$$

then we can cross multiply to find out whether m_1 should come before m_2 - we need to pay attention to the sign of the values being multiplied to return the correct result.

Although the algorithm works in most cases, it does not consider the following exceptional cases:

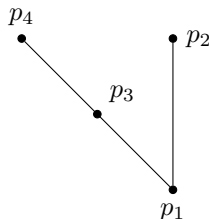
- If all the points are collinear with respect to the pivot, then there is no enclosing polygon.
- If points are collinear with respect to the pivot, they can be sorted in any order. However, we would require them to be sorted in the ‘right’ order. For instance, consider the following example:



Here, the pivot is p_1 . Since p_2 and p_3 make the same angle with the pivot, we would need to choose the right order- the current ordering does

not give us a simple polygon since p_3 should be the top right point. We mitigate this by further sorting the points by the distance it makes with the pivot.

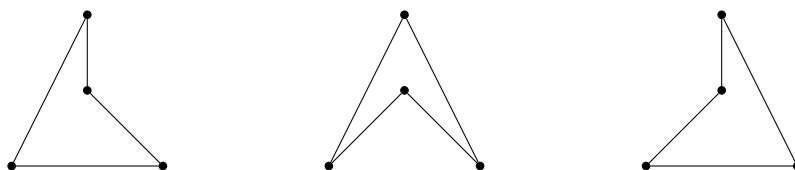
- If the final points are collinear to the pivot, then sorting them by increasing distance from the pivot does not work. For instance, consider the following example:



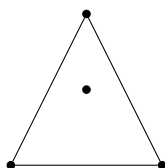
Here, the pivot is p_1 . Since p_3 and p_4 make the same angle with the pivot, we would sort p_3 and p_4 in the order given above by the observation we made in the previous point. However, this does not work when the point is getting connected to the pivot. In that case, we would need to reverse these points, i.e. sort them with respect to the largest distance with the pivot, not the smallest.

Convex Hull

Now, we will consider the convex hull problem. For a set of points on the plane, the convex hull is the smallest convex polygon that contains all the points. We measure the size of the polygon by its perimeter. So, we expect some of the points to lie on the boundary of this polygon. It is possible that not all of them lie on the boundary, as shown in the figure below.



In the figure above, we have 3 simple polygons from the same set of points. These are the only simple polygons containing all the points. Clearly, none of these are convex polygons. In fact, the convex hull of these 4 points is the following:

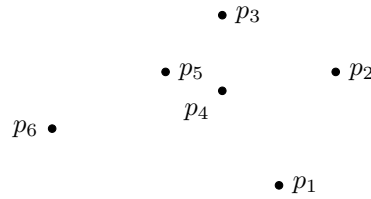


In a given set of points, the point that has the smallest or the largest x/y -coordinate will always lie on the convex hull. This implies that the choice of pivot from the simple polygon algorithm above will still work in this case.

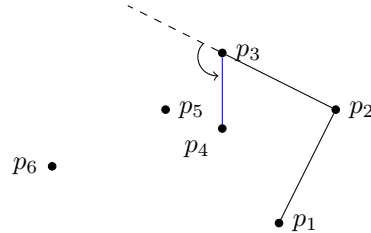
The algorithm for constructing the convex hull is the Graham Scan algorithm. It starts by running the simple polygon algorithm to sort the set of points with respect to the angle they make with the pivot. We then add these points in order to the convex hull, and remove any points as we realise that shouldn't lie on it. One way of finding out whether a point should be included is to:

- choose the pivot to be the point with the largest x -coordinate (and smallest y -coordinate); and
- choose to keep a point p_i if, when adding the next point p_{i+1} , the triplet (p_{i+1}, p_i, p_{i-1}) , the anti-clockwise angle between the two lines is less than π .

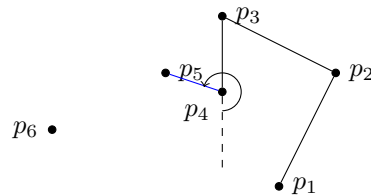
We illustrate the procedure with an example. Consider the following set of points.



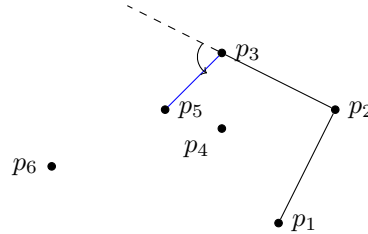
We have already applied the simple polygon algorithm to label the points with respect to the pivot p_1 . We first connect the first three points- it is guaranteed that the angle between $-p_1 - p_2-$ and $-p_2 - p_3$ is less than π , since the angles have been sorted in increasing order with respect to p_1 .



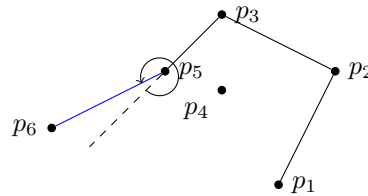
The next point to add is p_4 . The angle between the two lines $-p_2 - p_3-$ and $-p_3 - p_4-$ is less than π , so we do not remove p_3 .



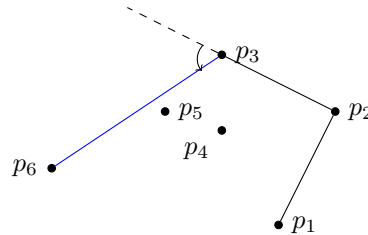
The next point to add is p_5 . However, the corresponding angle is clearly bigger than π , so we will first need to discard p_4 .



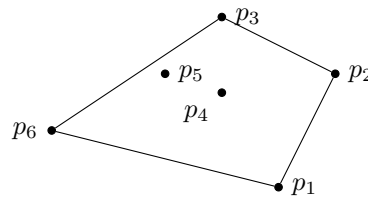
Now, the angle is less than π . So, we add p_5 to the convex hull.



This angle is also more than π , so we have to discard p_5 too.



The angle the lines $-p_2 - p_3 -$ and $-p_3 - p_6 -$ make is less than π , so p_6 gets added to the convex hull. Since it is the last point, we have constructed the convex hull:



We are guaranteed to make the right angle when we connect the last point with the pivot, as long as the pivot belongs to the convex hull.

The algorithm for the Graham Scan is given below.

```

1 List<Point> grahamScan(List<Point> points) {
2     simplePolygon(points);
3     List<Point> cHull = [points[0], points[1], points[2]];
4 }

```

```

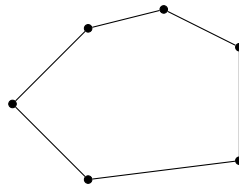
5 // add all the other points to the list => remove
6 // if the last 3 form a left angle
7 for (var i = 3; i < points.length; i++) {
8     while (angle(cHull[-2], cHull[-1], points[i]) > pi) {
9         cHull.pop();
10    }
11    cHull.add(points[i]);
12 }
13
14 return cHull;
15 }

```

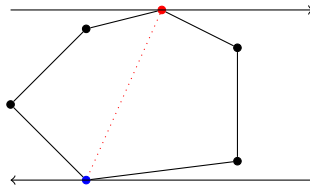
Now, we consider the complexity of the algorithm. The algorithm uses simple polygon construction, which has $O(n \log n)$ complexity. We can check whether the turn is anti-clockwise in $O(1)$ time. Now, the while loop within the for loop could imply that the complexity is $O(n^2)$. However, since a point is eliminated at most once from the list, the complexity of the entire loop is $O(n)$. Hence, the algorithm has complexity $O(n \log n)$.

Next, we will consider some applications of the convex hull algorithm. A basic usage might be for a robot to avoid some obstacles, while taking the shortest path. That is, the robot is at position A , wants to get to position B , and there are some obstacles in the middle. Then, we consider the set of points containing the obstacles and the positions A and B . We construct a convex hull for this set of points. The shortest path avoiding the obstacles will then be either the clockwise or the anti-clockwise path from A to B .

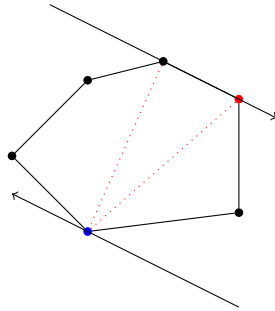
Another application is to compute the furthest pair of points in a set of points. A naive method would check each pair of points- that is $O(n^2)$. However, we can use the convex hull and compute the furthest pair of points in $O(n \log n)$ time- this is because the convex hull contains the pair of points that are furthest from each other.



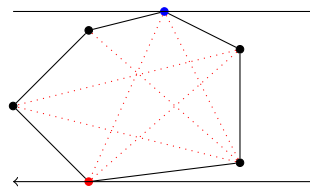
We first select 2 points- one that has a maximum x -coordinate, and a minimum y -coordinate.



We keep track of the distances between any two points we encounter. We rotate in clockwise direction, until we hit another point.



We continue this until the two lines have flipped.



Then, we choose the pair such that the distance is the longest- this is the longest distance of the entire set of points, not just the convex hull. This algorithm is called the rolling calipers.