1 ANGULAR SORTING (w/o angles)

1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

1 ANGULAR SORTING (w/o angles)

1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

```
genericSort(L, comparator)
```

1 ANGULAR SORTING (w/o angles)

1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

```
genericSort(L, comparator)
```

list of points
$[(x_1, y_1), (x_2\ y_2) \cdots$

how do you determine whether one point is "bigger"?
(now to compare 2 points?)

1 ANGULAR SORTING (w/o angles)

1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

`genericSort(L, comparator)`

list of points
$[(x_1, y_1), (x_2, y_2) \cdots$

currently: here is our comparator

COMPARE (A, B):
1. calculate $\angle AQ = \alpha$
2. calculate $\angle BQ = \beta$
3. return $\alpha \geq \beta$

1 ANGULAR SORTING (w/o angles)

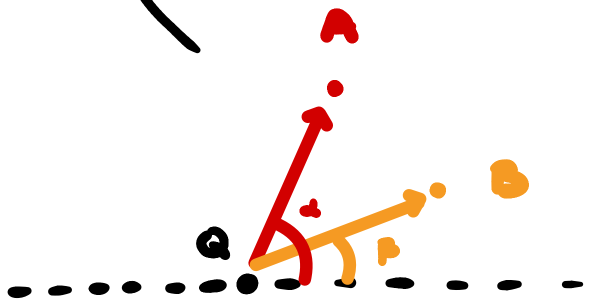
1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

```
genericSort(L, comparator)
```

list of points

$[(x_1, y_1), (x_2, y_2) \cdots$



1 ANGULAR SORTING (w/o angles)

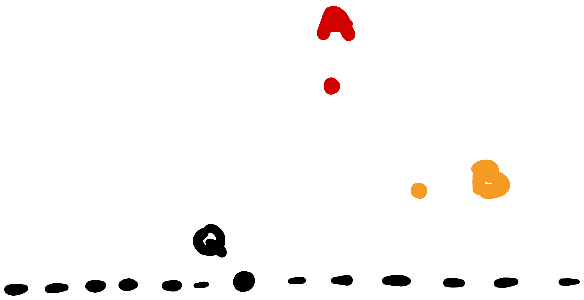
1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

```
genericSort(L, comparator)
```

list of points
$[(x_1, y_1), (x_2, y_2) \dots$

what is another
comparator that
uses line-side test?



1 ANGULAR SORTING (w/o angles)

1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

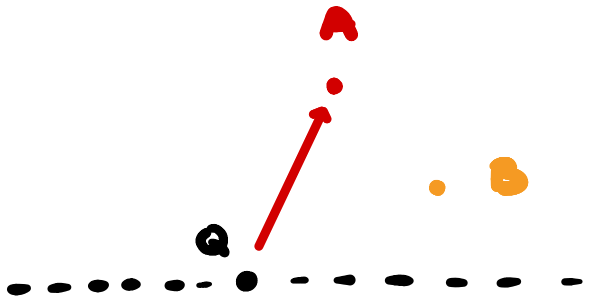
   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

```
genericSort(L, comparator)
```

list of points
$[(x_1, y_1), (x_2 y_2) \cdots$

what is another comparator that uses line-side test?

is B left or right of $\overrightarrow{QA}$



1 ANGULAR SORTING (w/o angles)

1. **(Angular sorting without angles)** Recall that the first step of the *Graham scan* algorithm for convex hull that we learned in lectures is to sort the points with respect to their angle from the bottom-most point. The most straightforward way to do this is of course to simply compute the angles and then sort the points. This has some drawbacks, such as having to perform floating-point computations that are susceptible to rounding errors.

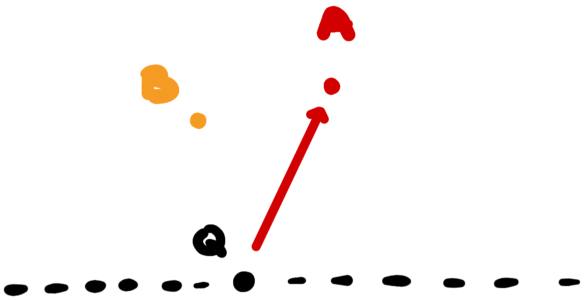
   Given a set of n points with integer-valued coordinates, describe how to sort them with respect to their angle to the bottom-most point without using any floating-point computations. (Hint: use the line-side test primitive).

```
genericSort(L, comparator)
```

list of points

$[(x_1, y_1), (x_2, y_2) \cdots$

what is another comparator that uses line-side test?

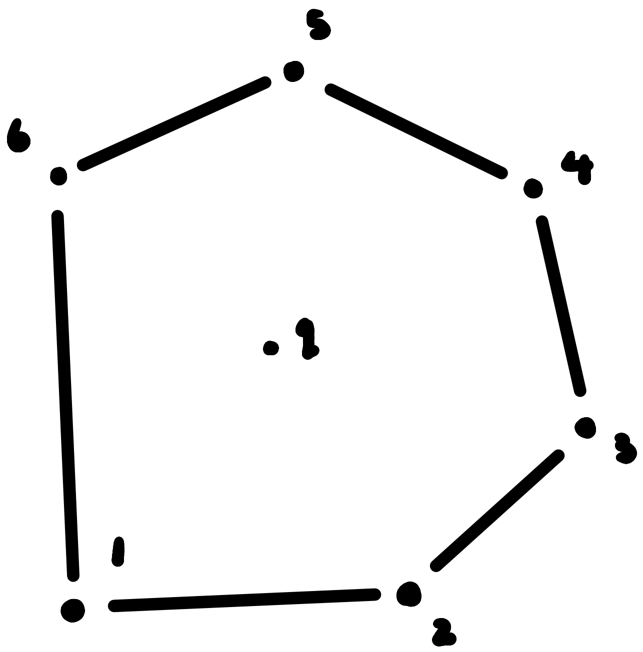


is B left or right of $\overrightarrow{QA}$

2 POINT IN POLYGON

2. **(Point-in-convex-polygon)** Given a convex polygon P represented by points $P[1], P[2], \ldots, P[n]$ in counter-clockwise order, and a point q

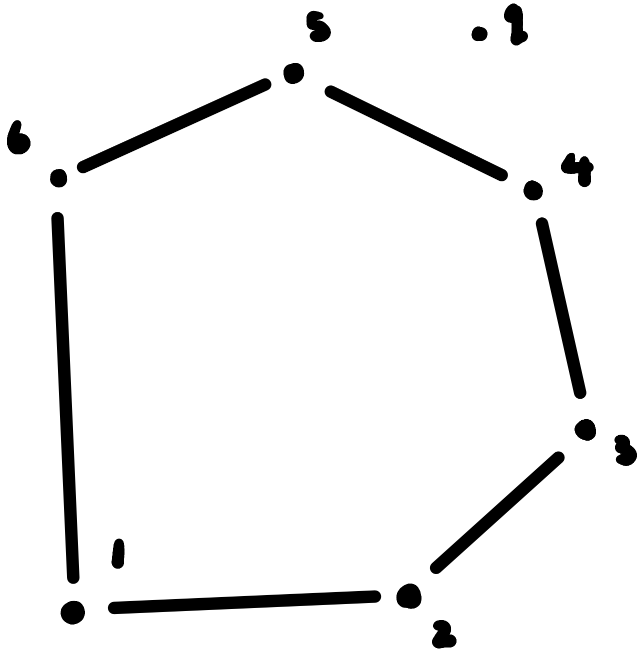
   (a) Determine whether q is in the polygon in time $O(n)$.



2 POINT IN POLYGON

2. **(Point-in-convex-polygon)** Given a convex polygon P represented by points $P[1], P[2], \ldots, P[n]$ in counter-clockwise order, and a point q

   (a) Determine whether q is in the polygon in time $O(n)$.



KEY STRATEGY:

use line side tests as your primative operation

CLAIM

a directed segment btw a pair of points (p_i, p_j) is on the convex hull

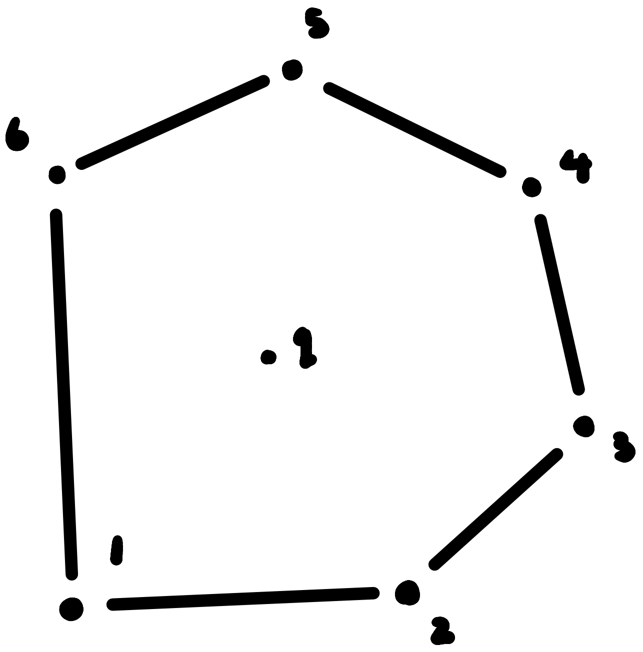
$\Leftrightarrow$

all other points p_k are to the left of the ray: $\overrightarrow{p_i p_j}$

2 POINT IN POLYGON

2. **(Point-in-convex-polygon)** Given a convex polygon P represented by points $P[1], P[2], \dots, P[n]$ in counter-clockwise order, and a point q

   (a) Determine whether q is in the polygon in time $O(n)$.



*we can iterate thru $\overrightarrow{P[i]\,P[i+1]}$ and ensure q is always left

KEY STRATEGY:

use line side tests as your primative operation

CLAIM

a directed segment btw a pair of points (p_i, p_j) is on the convex hull

<=>

all other points p_k are to the left of the ray: $\overrightarrow{p_i p_j}$

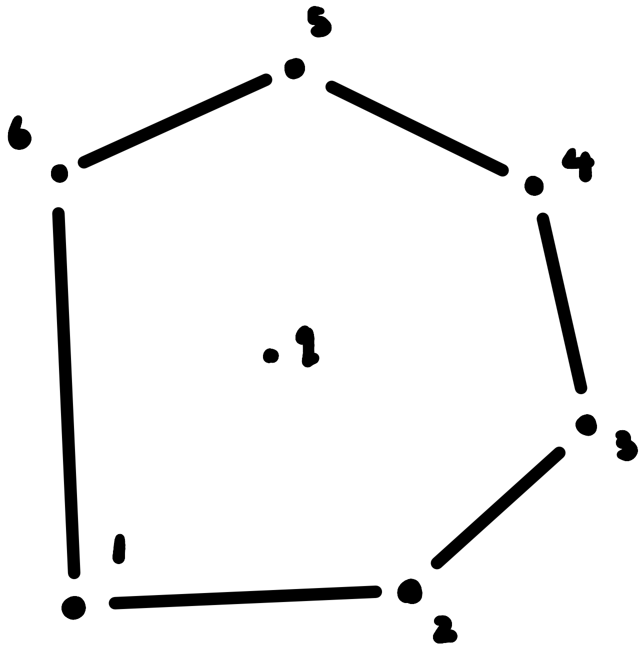
(b) Speed up your algorithm to $O(\log n)$

(b) Speed up your algorithm to $\underline{O(\log n)}$

- binary search will be involved somewhere

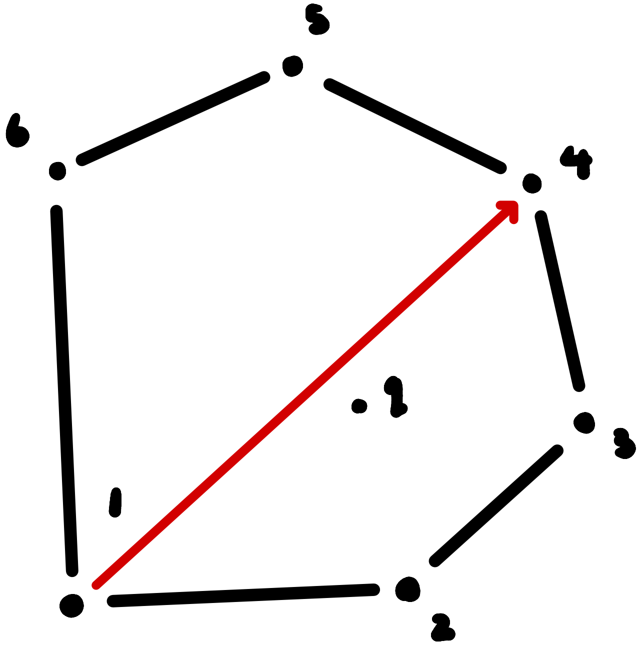
- line-side tests

- let's try to combine these
  to start...



(b) Speed up your algorithm to $\underline{O(\log n)}$

* search for closest/smallest region q is in

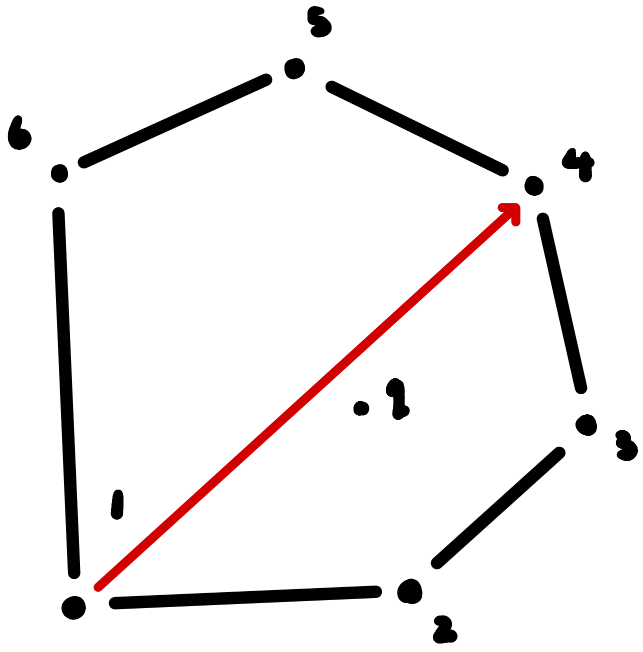


(b) Speed up your algorithm to $\underline{O(\log n)}$

*we don't have to check....

- $\overrightarrow{P_6 P_1}$
- $\overrightarrow{P_5 P_6}$
- $\overrightarrow{P_4 P_5}$

... knowing the result of this line-side test



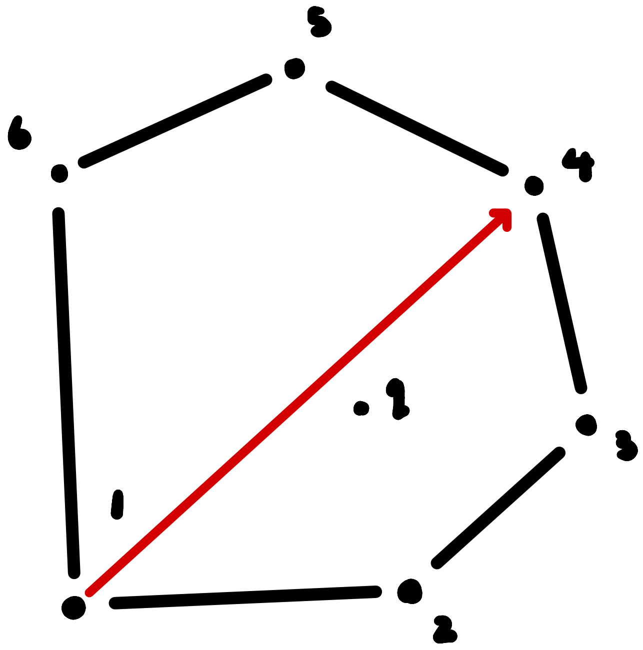
(b) Speed up your algorithm to $\underline{O(\log n)}$

*we don't have to check....

- $\overrightarrow{P_6 P_1}$
- $\overrightarrow{P_5 P_6}$
- $\overrightarrow{P_4 P_5}$

... knowing the result of this line-side test

this is because any segment to the right will have the power to tell us the answer...



(b) Speed up your algorithm to $\underline{O(\log n)}$

*we don't have to check....

- $\overrightarrow{P_6 P_1}$
- $\overrightarrow{P_5 P_6}$
- $\overrightarrow{P_4 P_5}$

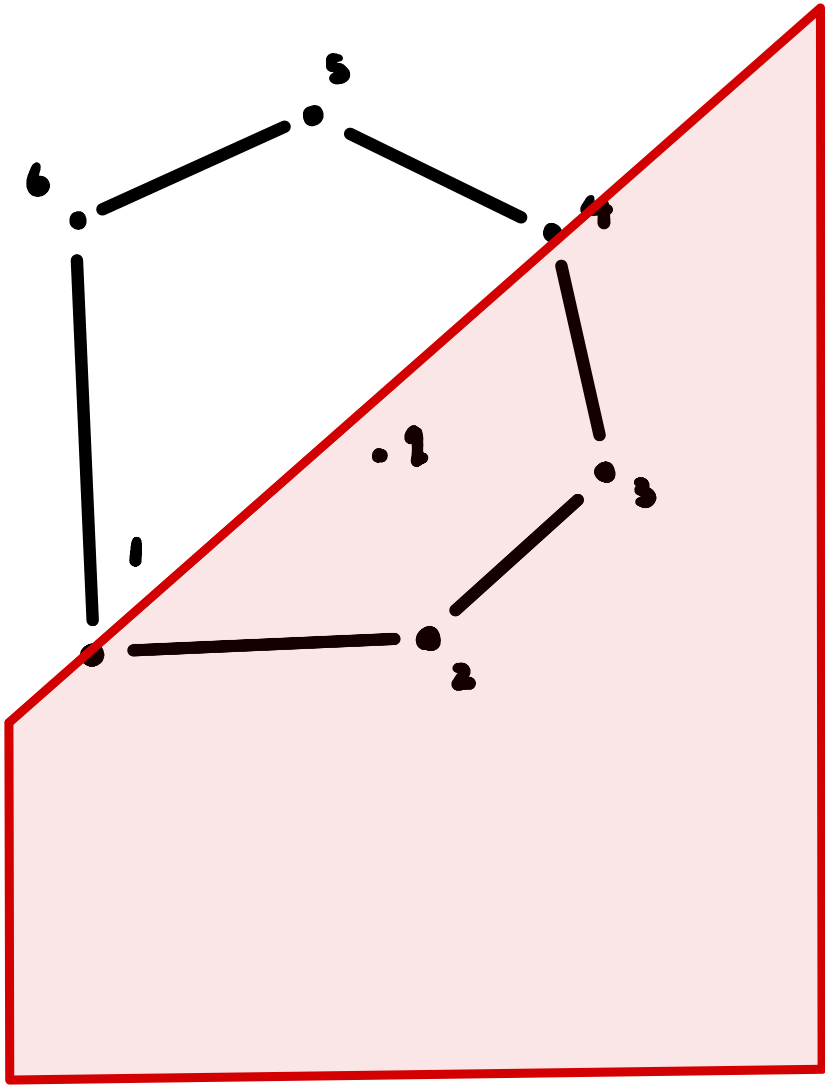
... knowing the result of this line-side test

this is because any segment to the right will have the power to tell us the answer...

KEY IDEA:

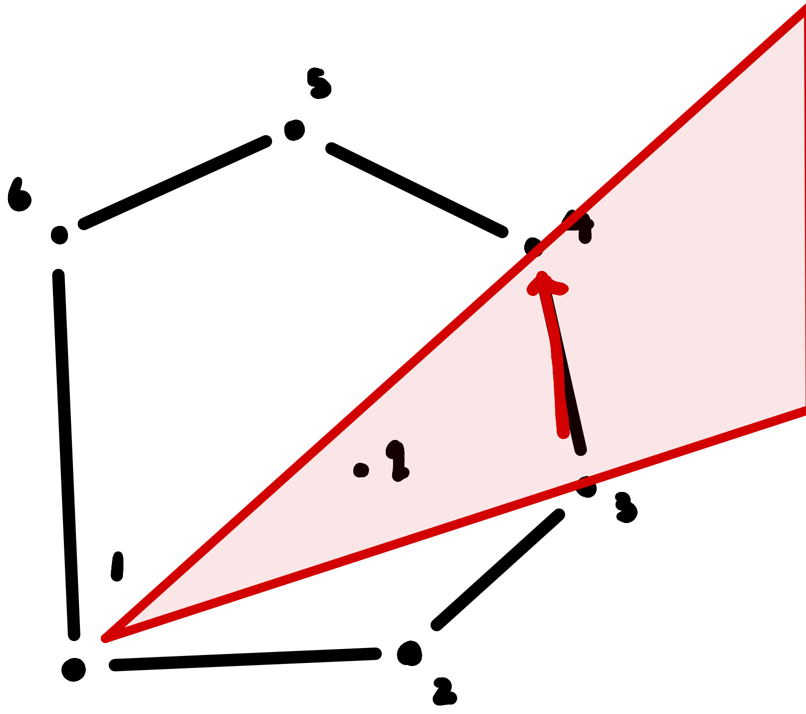
use binary search to narrow down the polygon segments we need to check...

aka narrow the region that q is in.



(b) Speed up your algorithm to $\underline{O(\log n)}$

* we have the triangle where q must live thru binary search....
(narrowed down polygon edges to just 1) :)

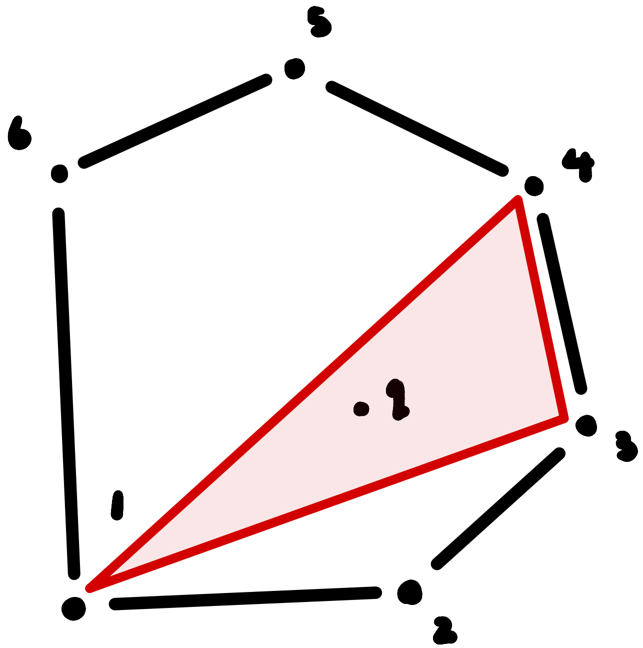


(b) Speed up your algorithm to $\underline{O(\log n)}$

* we have the triangle where q must live thru binary search....
(narrowed down polygon edges to just 1) ☺

is it in the polygon?

yep... we just check $\overrightarrow{P_3 P_4}$



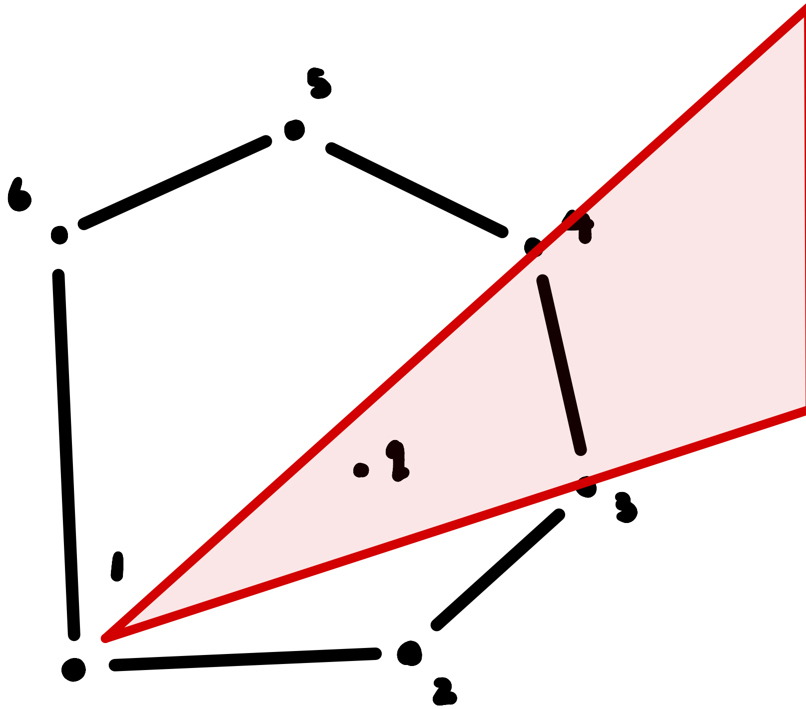
(b) Speed up your algorithm to $\underline{O(\log n)}$

* we have the triangle where q must live thru binary search....
(narrowed down polygon edges to just 1) ☺

is it in the polygon?

yep... we just check $\overrightarrow{P_3 P_4}$

* so what's the final algorithm?



(b) Speed up your algorithm to $\underline{O(\log n)}$

* we have the triangle where q must live thru binary search....
(narrowed down polygon edges to just 1) ☺

is it in the polygon?

yep... we just check $\overrightarrow{P_3P_4}$

* so what's the final algorithm?

0. the formal alg does $\overrightarrow{P_1P_2}$, $\overrightarrow{P_6P_1}$ LST to avoid edge cases
1. binary search for the triangle slice that q is in
2. check whether inside or out on that slice

<u>NOTE</u>: start point doesn't matter