Computational Geometry
[csci 3250]

Laura Toma

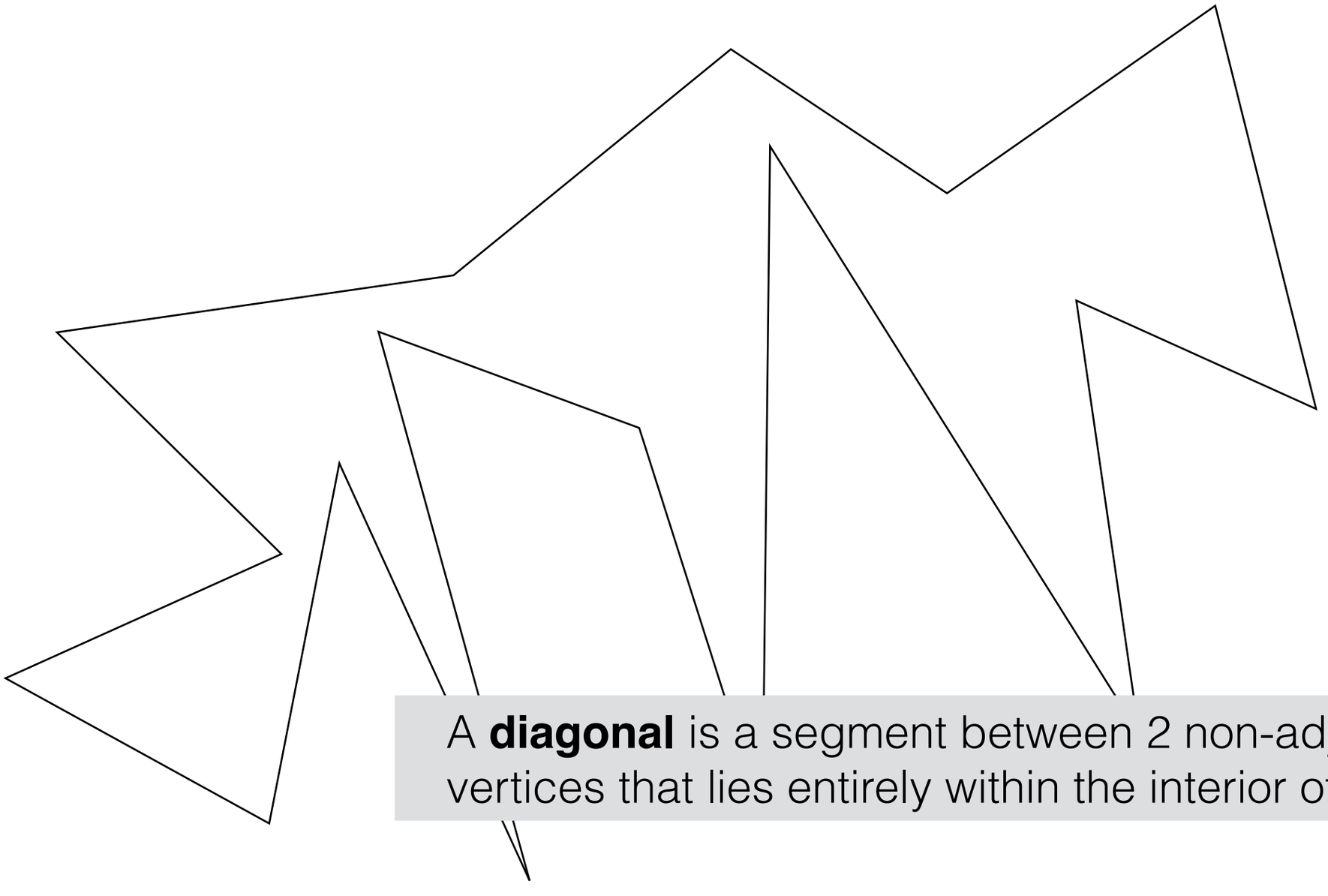
Bowdoin College

Polygon Triangulation

Polygon Triangulation: Definition

Polygon P

Triangulation of P: a partition of P into triangles using a set of diagonals.

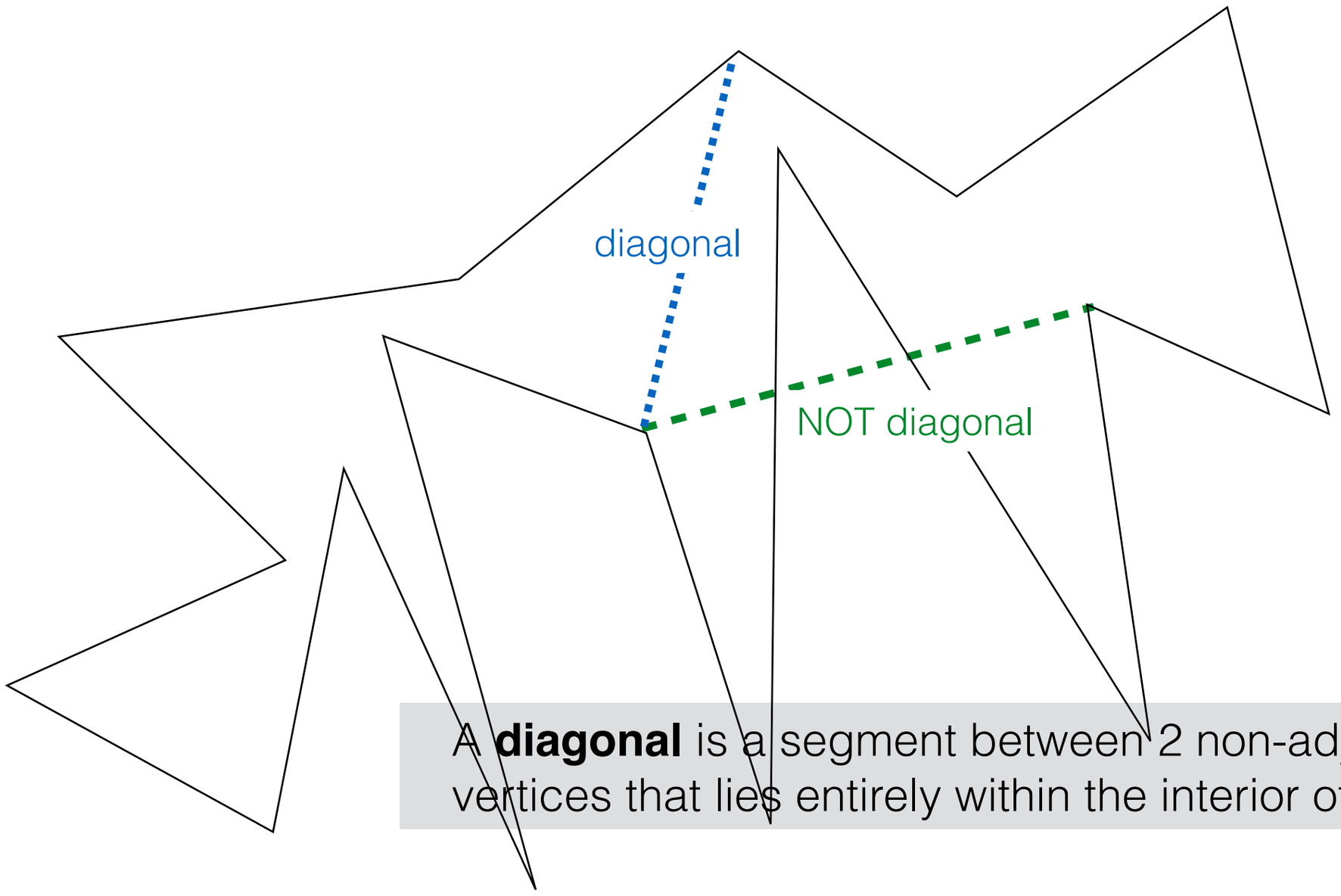


A **diagonal** is a segment between 2 non-adjacent vertices that lies entirely within the interior of the polygon.

Polygon Triangulation: Definition

Polygon P

Triangulation of P: a partition of P into triangles using a set of diagonals.

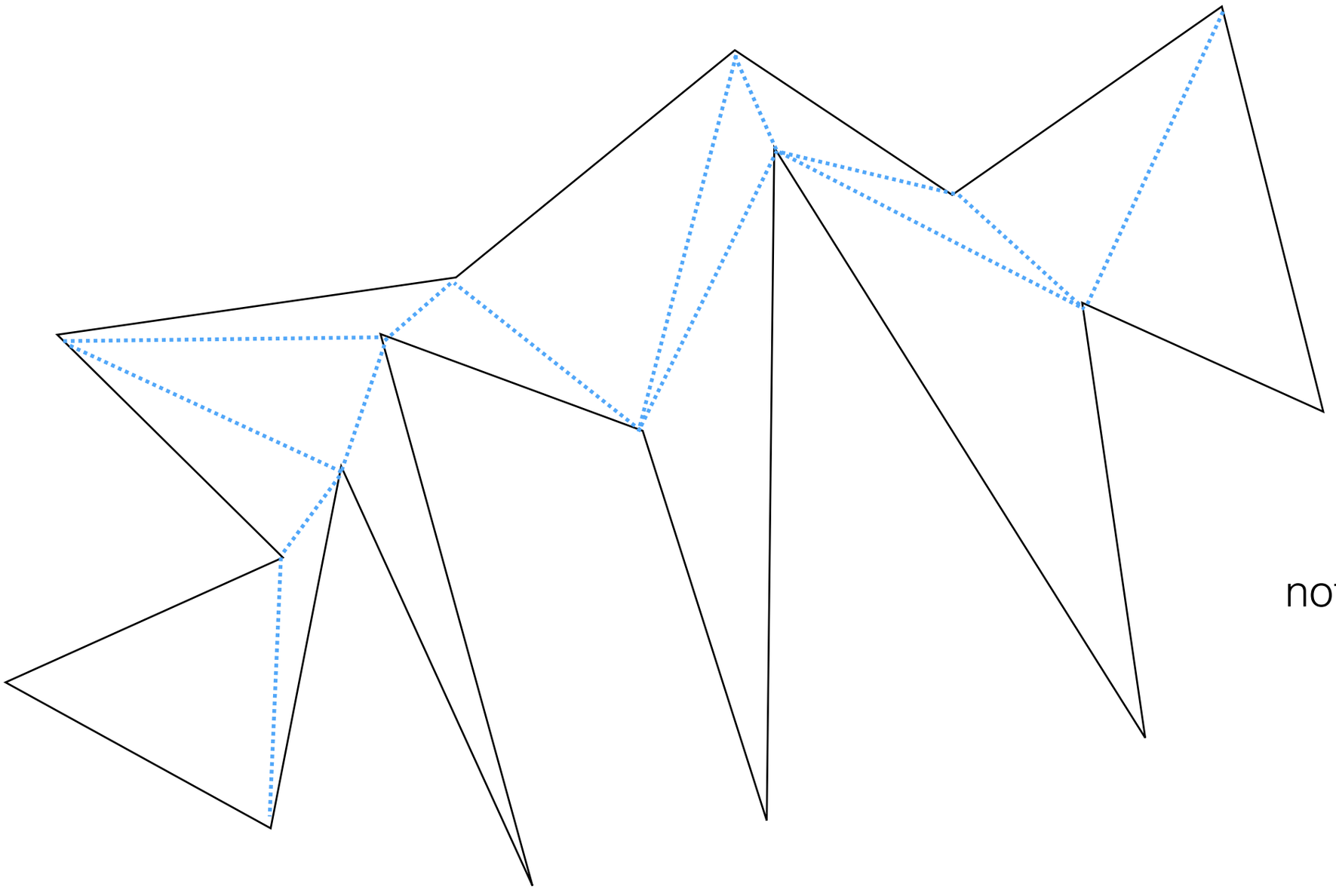


A **diagonal** is a segment between 2 non-adjacent vertices that lies entirely within the interior of the polygon.

Polygon Triangulation: Definition

Polygon P

Triangulation of P: a partition of P into triangles using a set of diagonals.

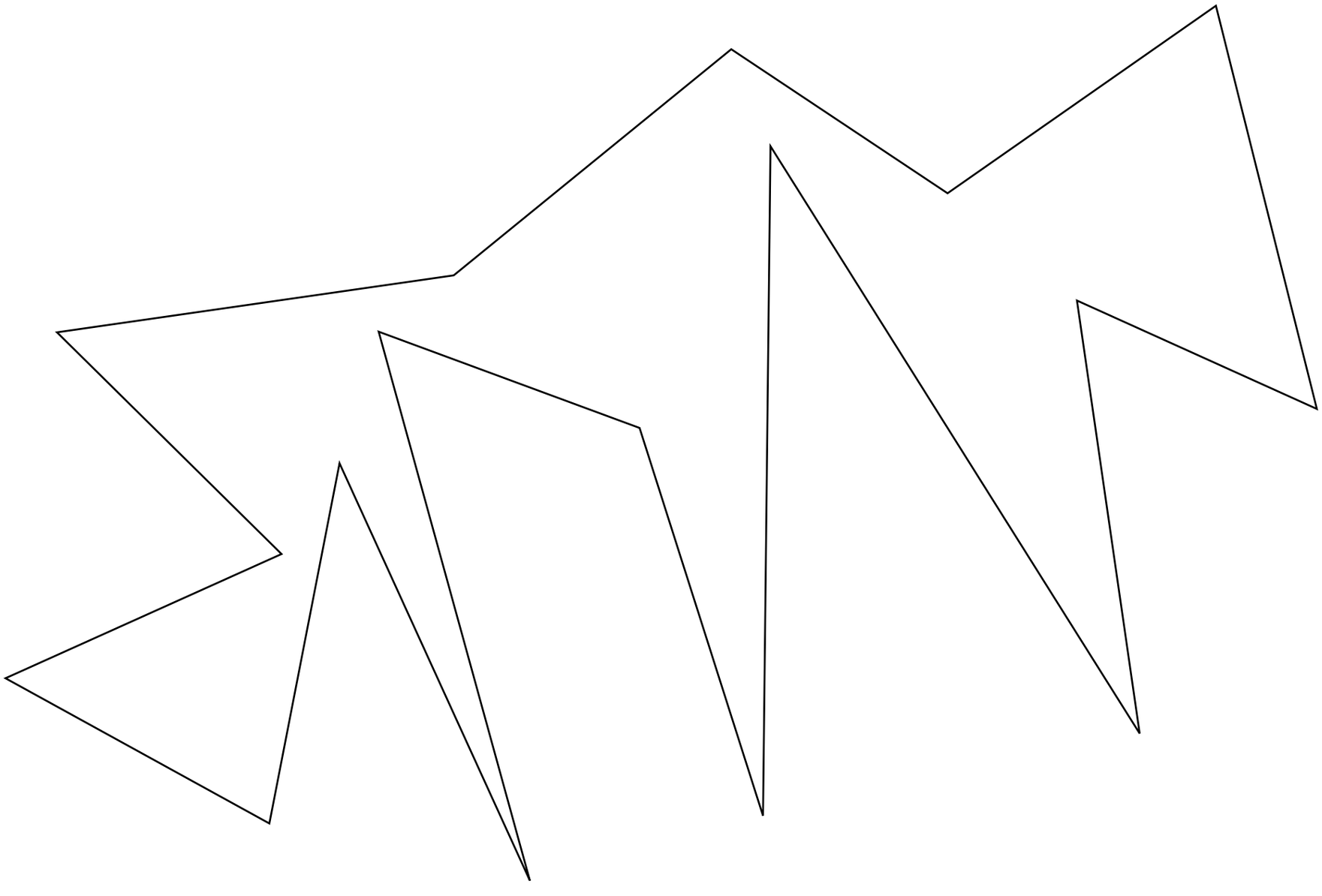


not unique

Polygon Triangulation: The problem

Given a polygon P, triangulate it.

(output a set of diagonals that partition the polygon into triangles).

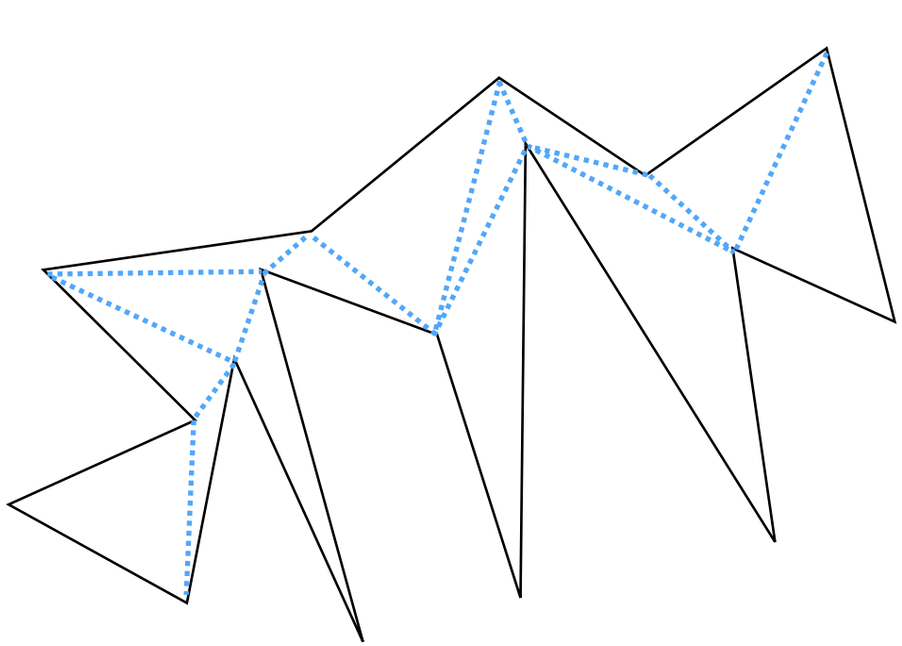


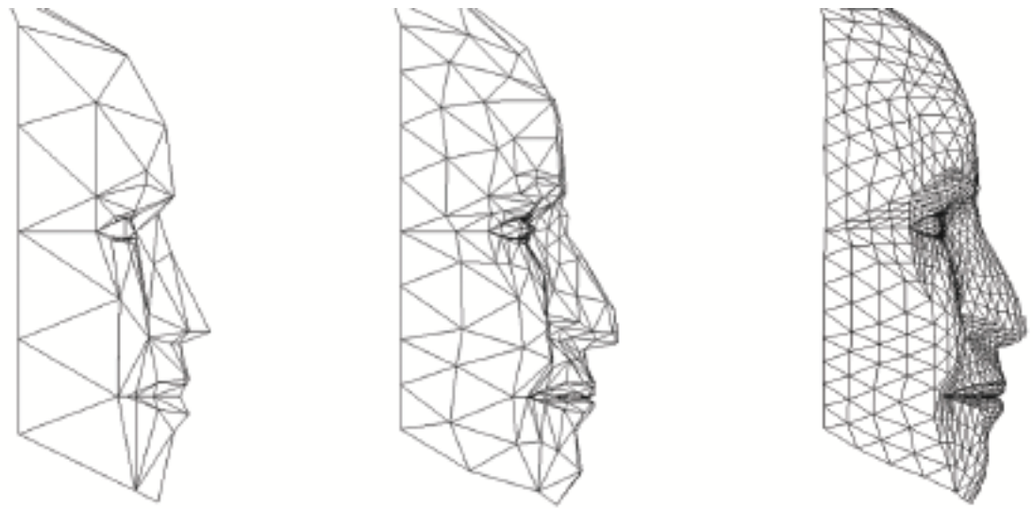
Motivation

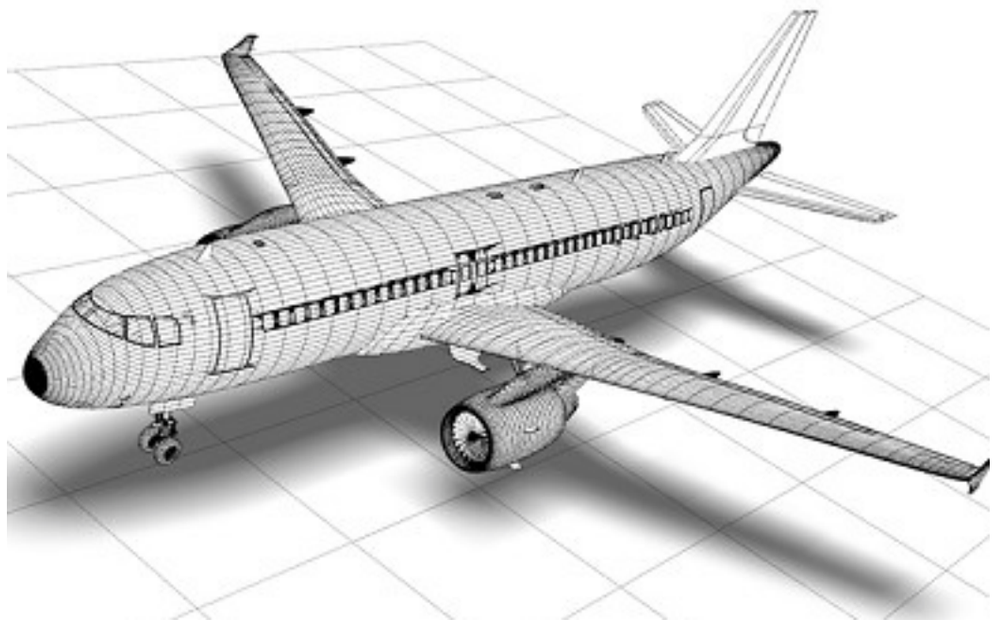
Partitioning into simpler shapes: technique for dealing with complexity

2D: polygon triangulation

3D: meshing



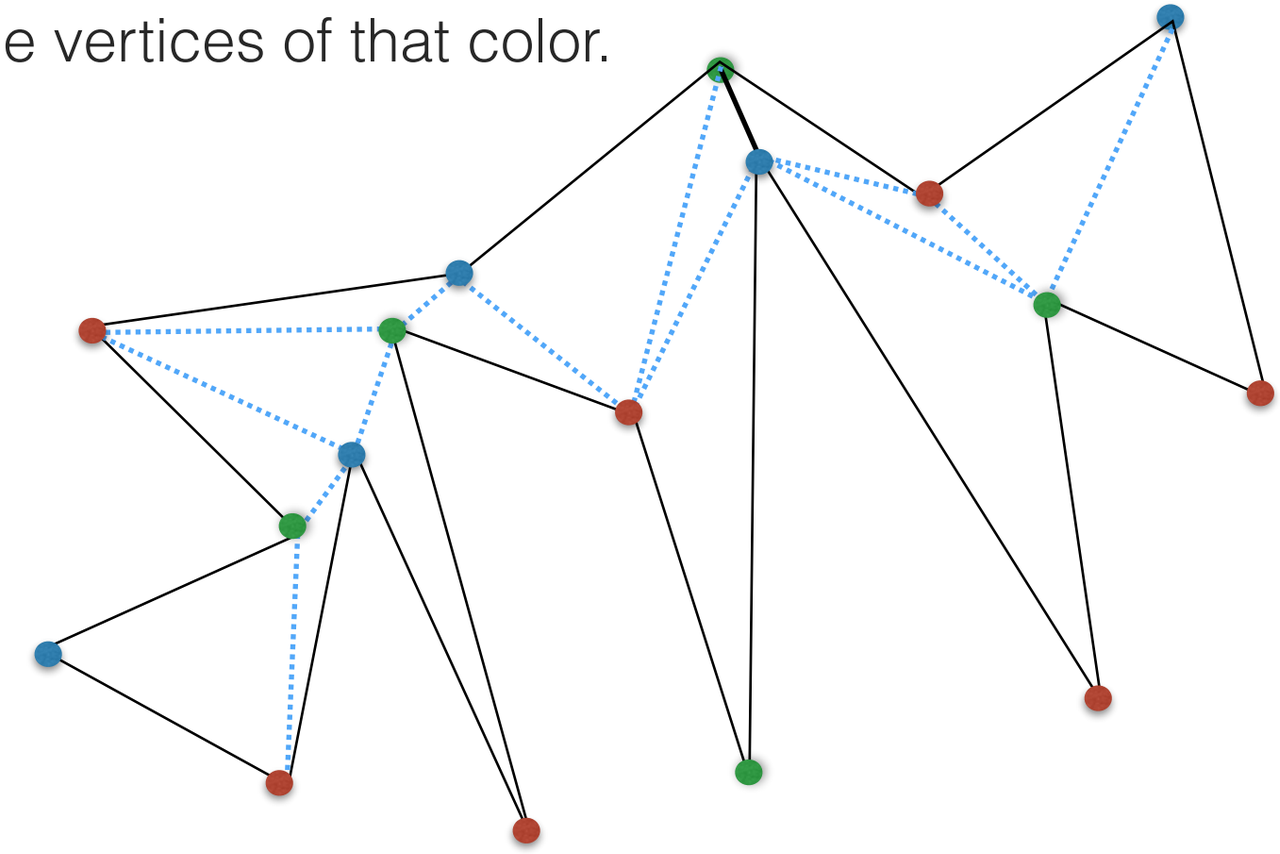




Motivation

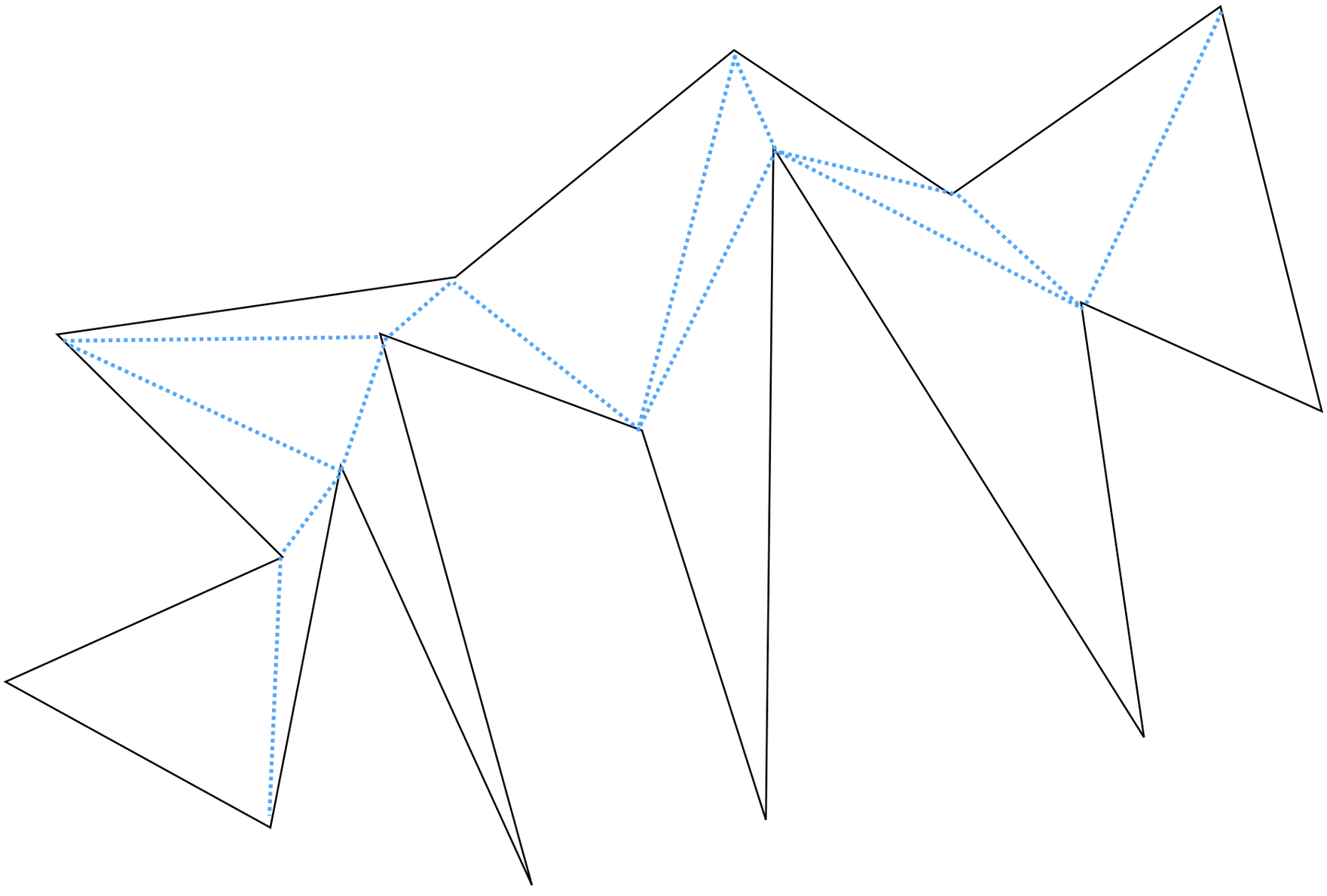
Art gallery , Fisk's proof

1. Any simple polygon can be triangulated.
2. Any triangulated simple polygon can be 3-colored.
3. Placing the guards at all the vertices assigned to one color guarantees the polygon is covered.
4. There must exist a color that's used at most n/3 times. Pick that color and place guards at the vertices of that color.



Does a triangulation always exist?

YES. The key to proving this is that any polygon (n>3) has a diagonal.

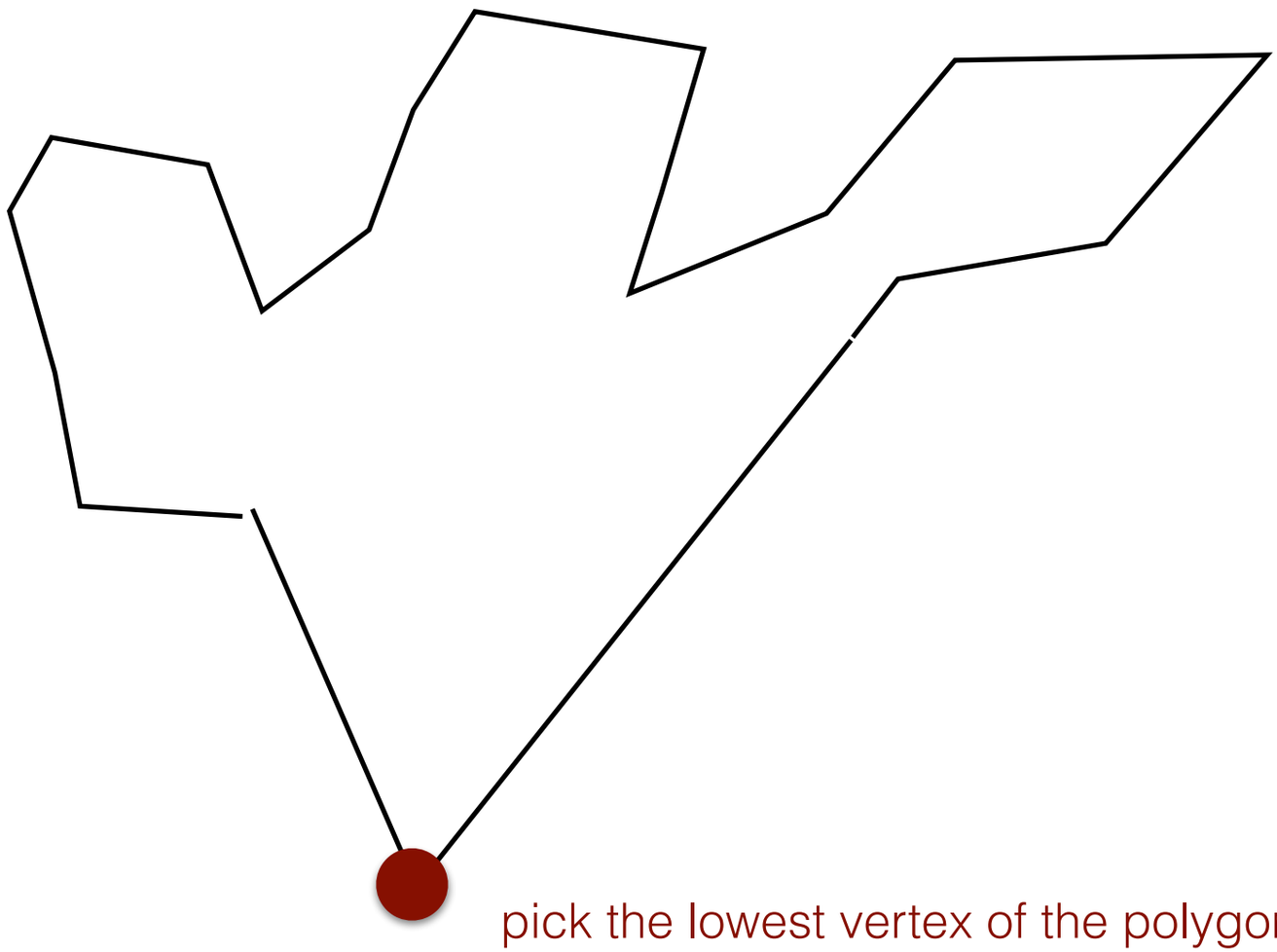


Known Results

- Theorem 1: Any simple polygon must have a convex vertex (angle <180).
- Theorem 2: Any simple polygon with n>3 vertices contains (at least) a diagonal.
- Theorem 3: Any polygon can be triangulated by adding diagonals.
- Theorem 4: Any triangulation of a polygon of n vertices has n-2 triangles and n-3 diagonals.
- Theorem 5: Any simple polygon has at least two ears.

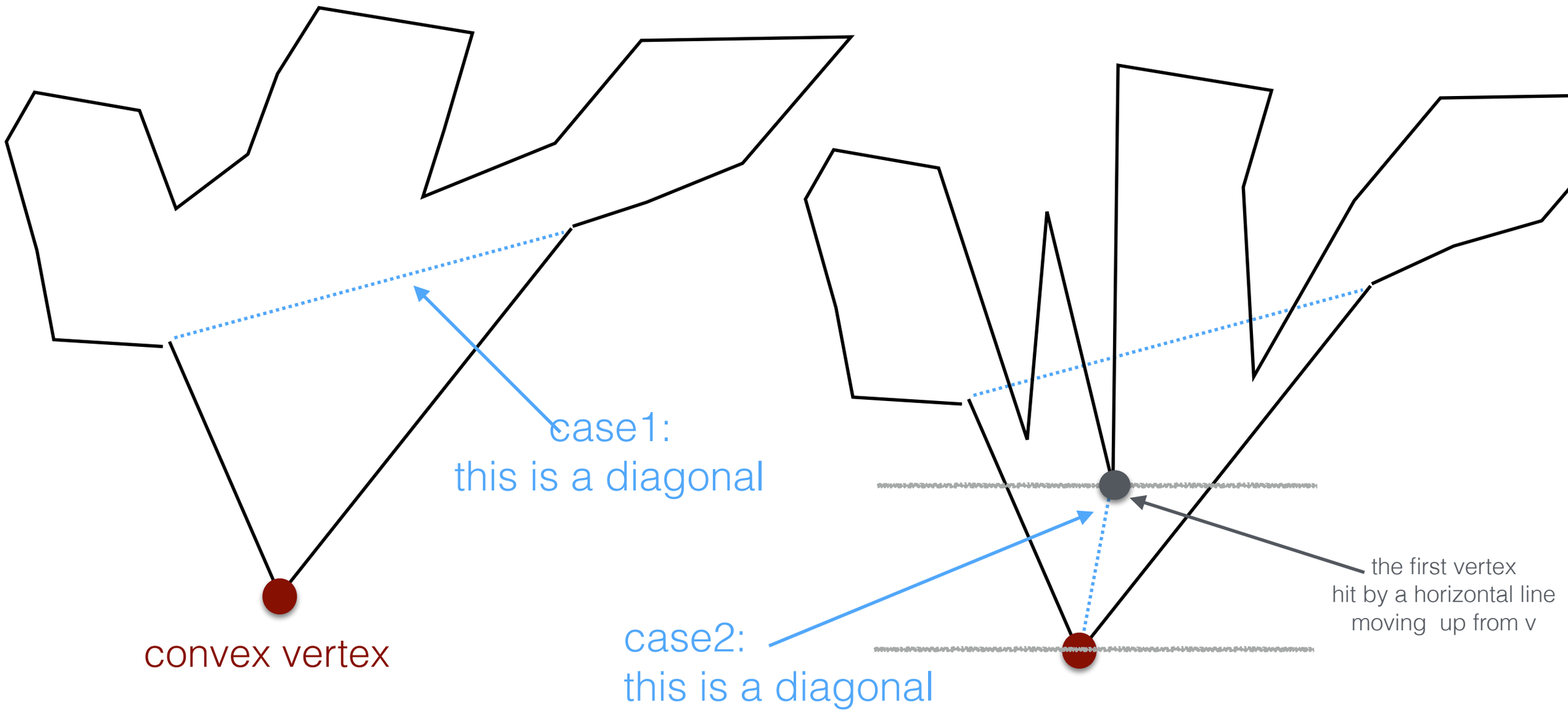
Theorem 1: Any simple polygon contains at least one convex vertex

the angle is <180



pick the lowest vertex of the polygon

Theorem 2: Any simple polygon contains at least one diagonal.

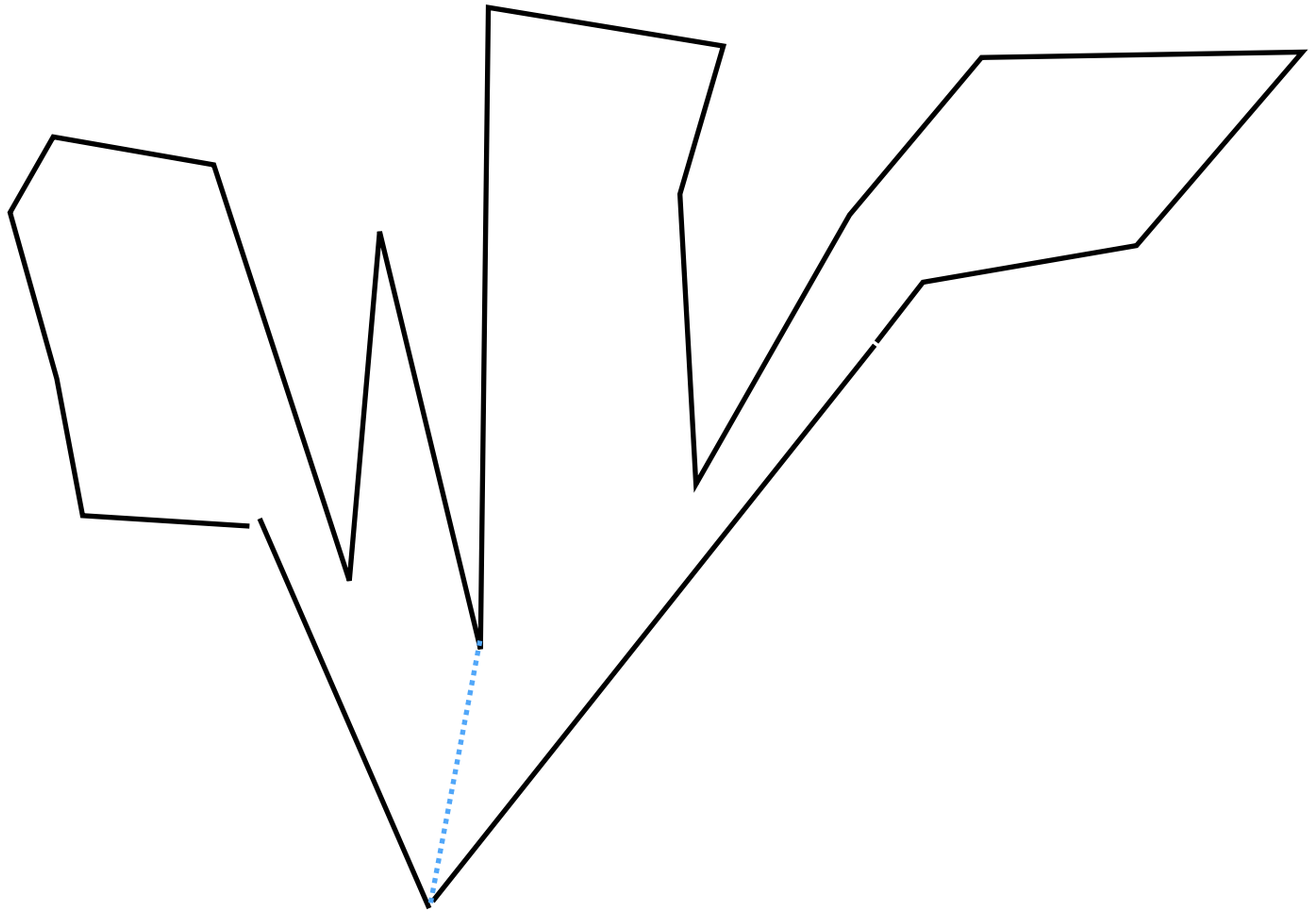


Theorem 3: Any polygon can be triangulated by adding diagonals

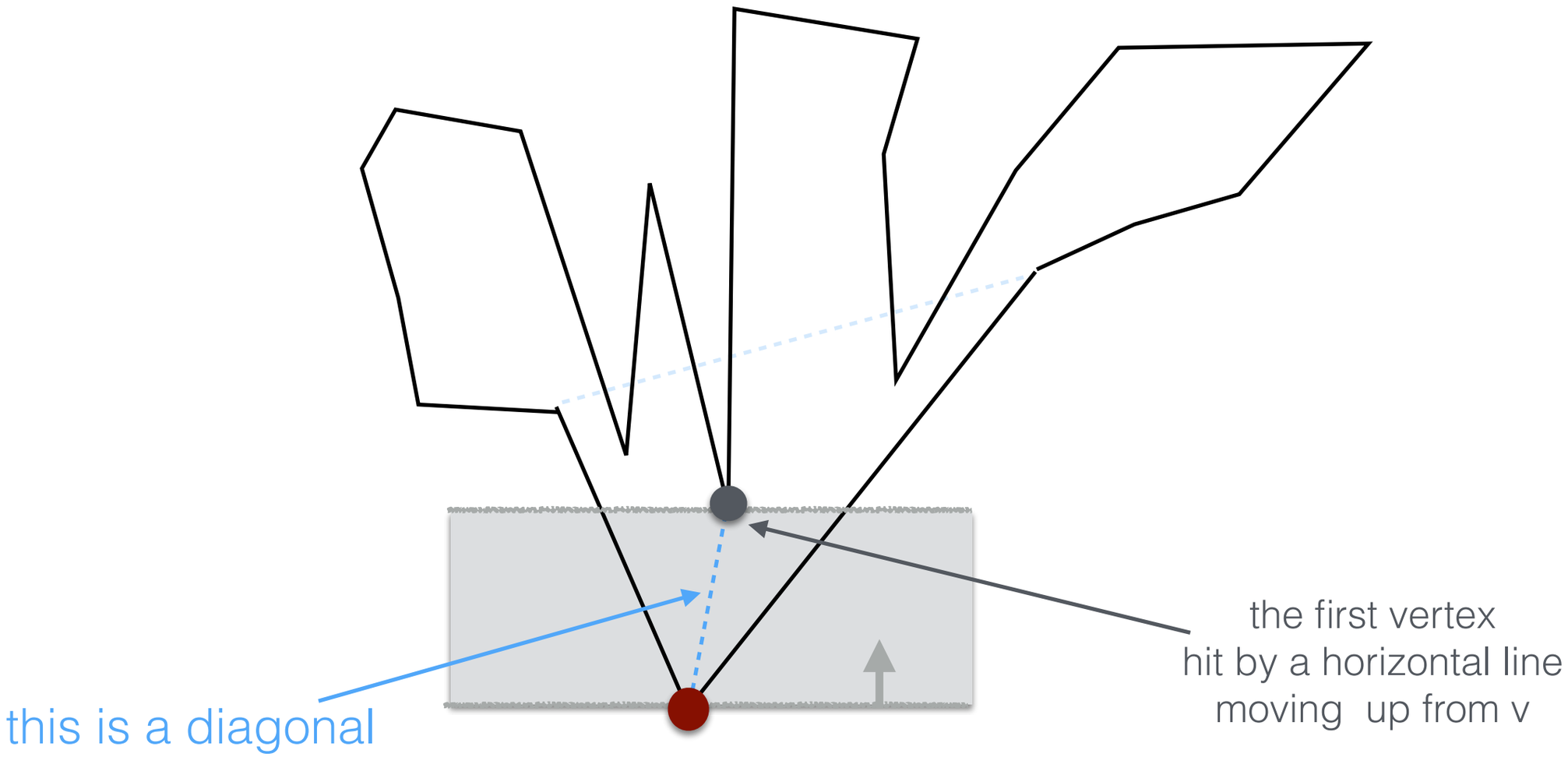
Proof: By induction on the size of the polygon

if n=3, holds trivially

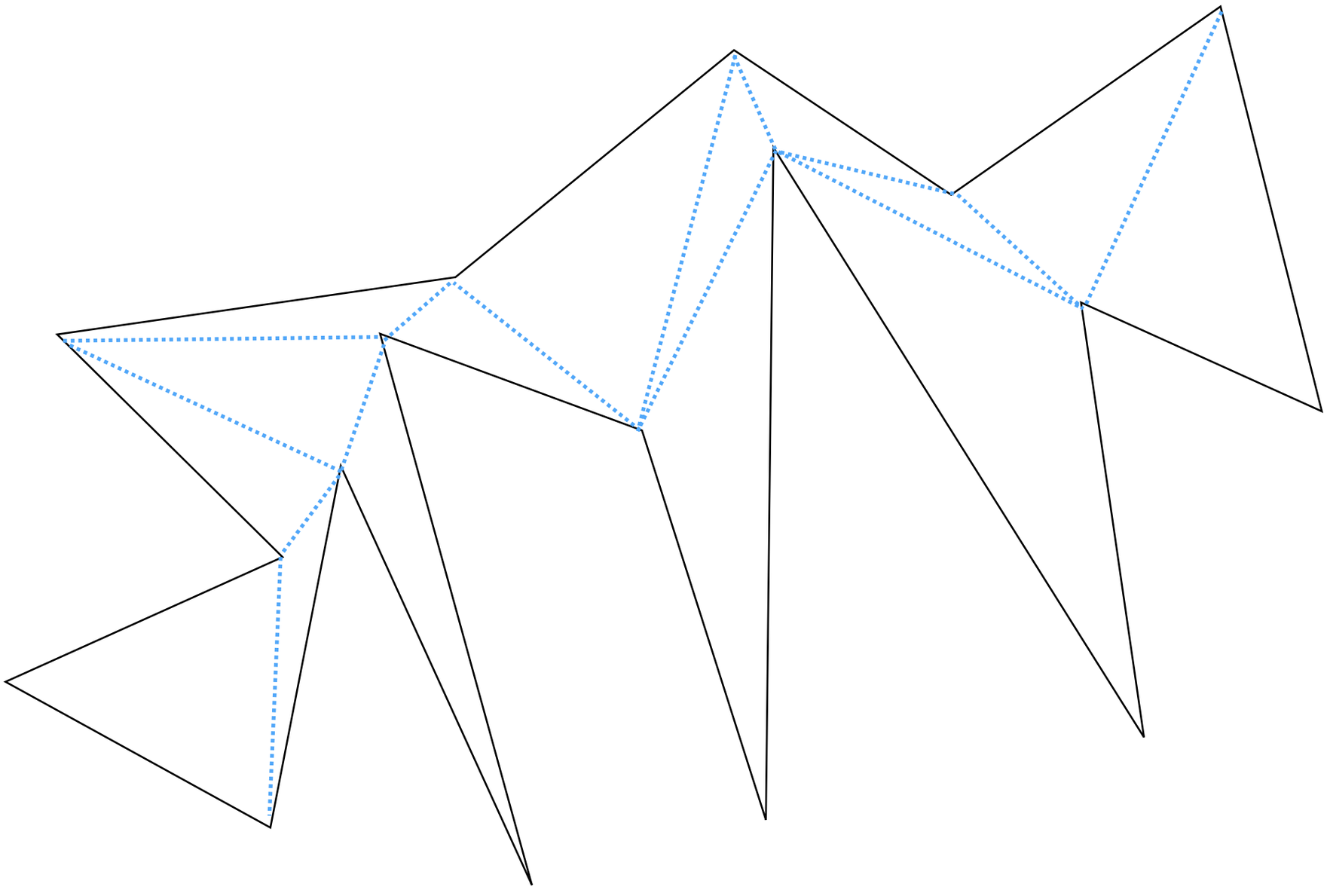
Assume it holds for any k < n. Partition P into two polygons with a diagonal. Each one has < n vertices, and can be triangulated by ind. hyp.



Theorem 2: Any simple polygon contains at least one diagonal.



We want to triangulate a polygon



Naive triangulation by recursively finding diagonals

- Assume P is a polygon given as a vector of points (in ccw order along boundary)
- Idea:
    - Find a diagonal
    - Recurse
- Let's come up with an algorithm to determine if two vertices of P form a diagonal: isDiagonal(p_i, p_j)

- Let P be a polygon given as a vector of points

- ab is diagonal if it does not intersect the edges of P and is interior to P

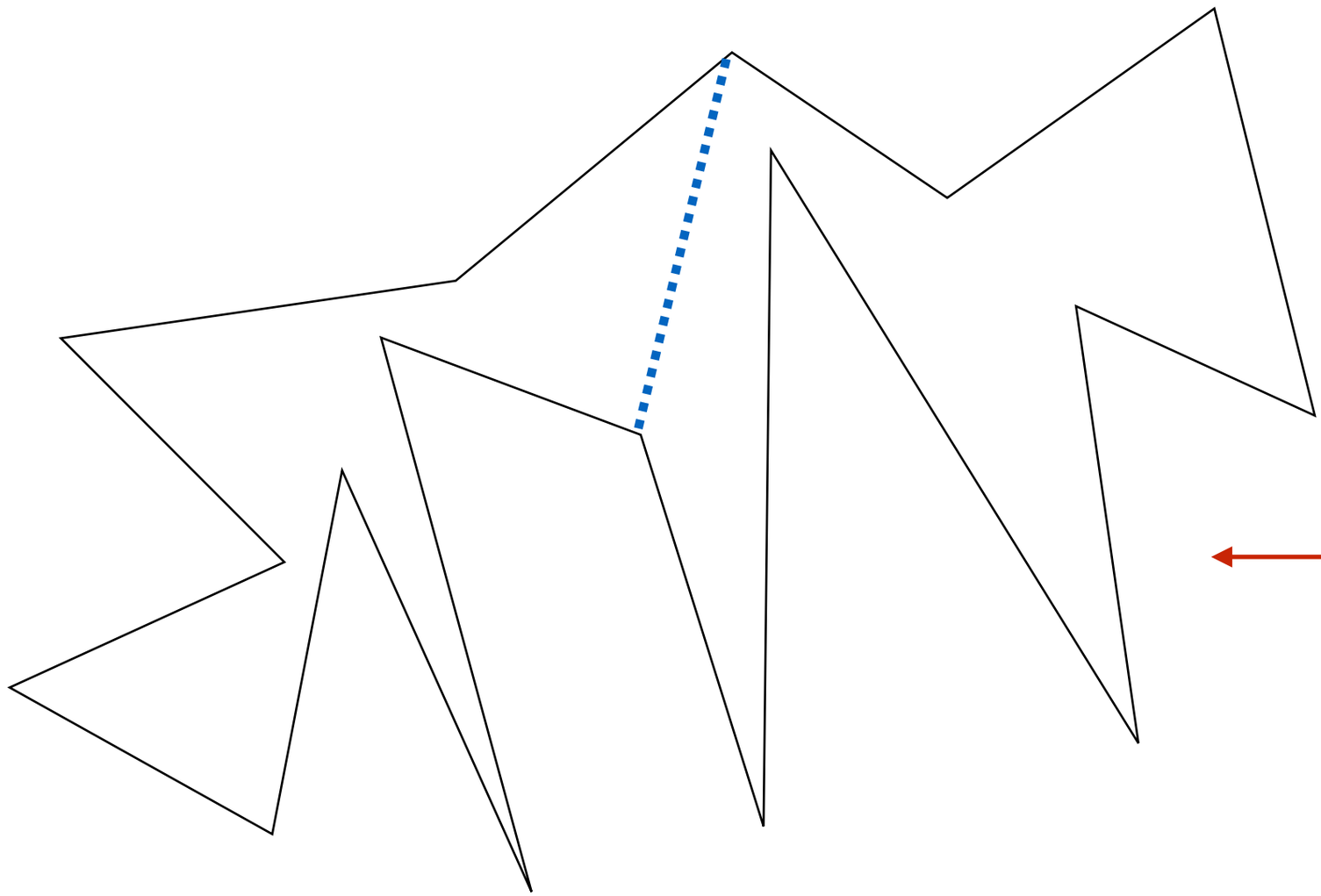
isDiagonal(a,b):

//does any of the edges intersect ab?

- for i=0, i<n, i++:
  - if $p_i p_{(i+1) mod\ n}$ intersects ab: return False ← actually: intersection at vertices is ok for the edges adjacent to a and b

//if we got here, we know that ab intersects no edge. The only thing left to check is whether it's inside or outside P

- return True if inside, False if outside



actually: intersection at vertices is ok for the edges adjacent to a and b

- Let P be a polygon given by a vector of vertices P

isDiagonal(a,b):

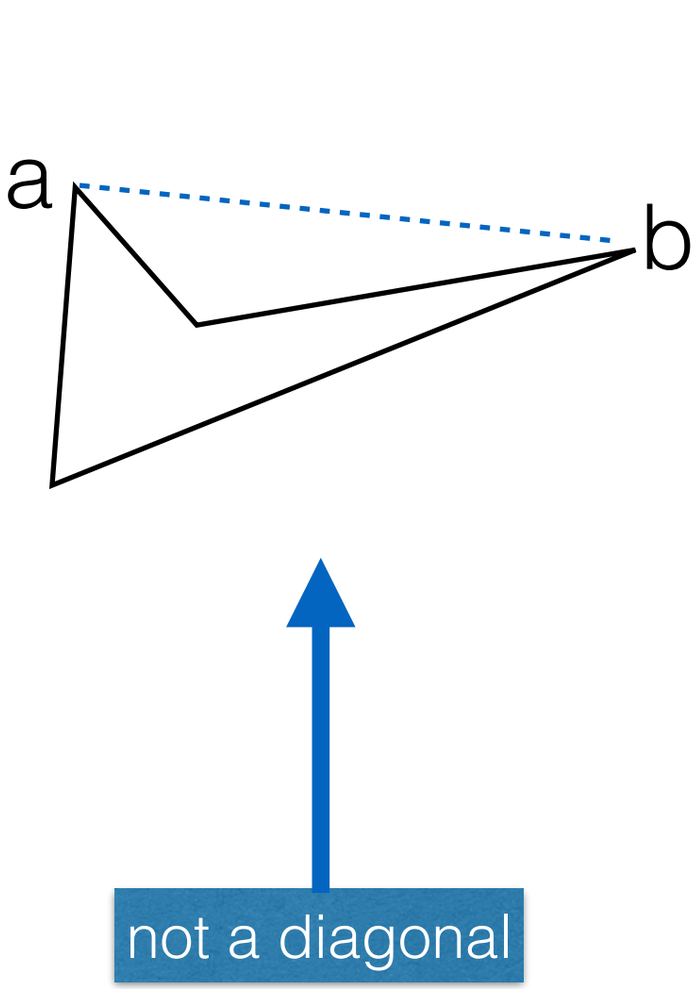
//does any of the edges intersect ab?

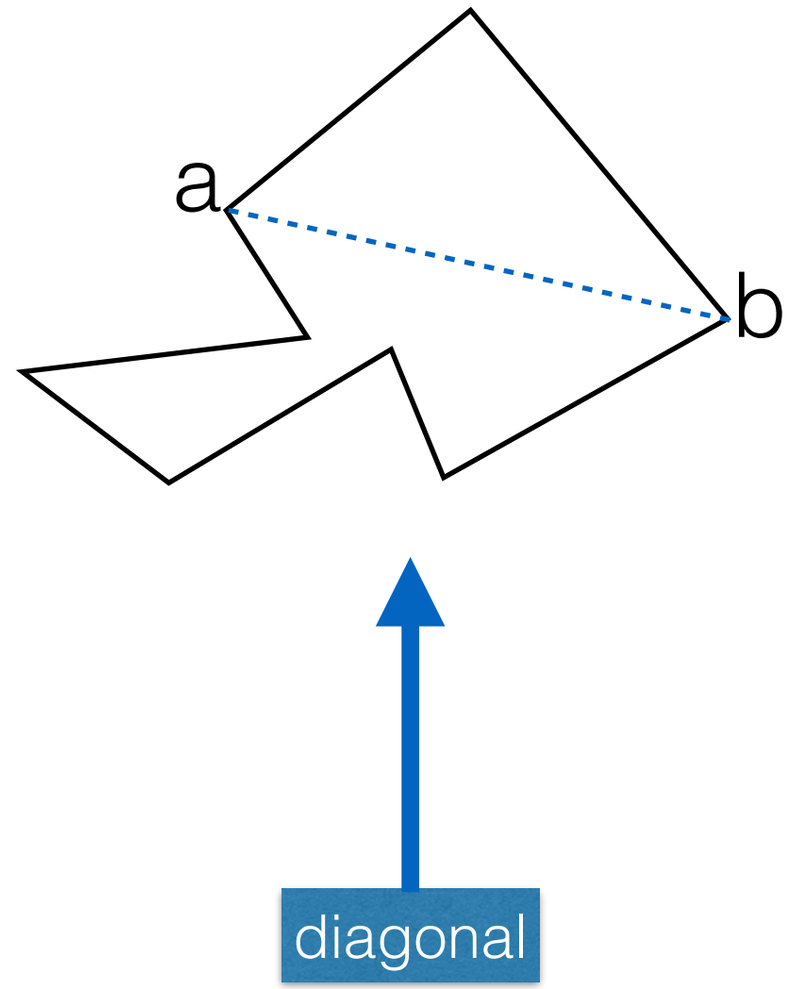
- for i=0; i< n; i++
  - if $a != p_i$ and $a != p_{(i+1) mod\ n}$ and $b != p_i$ and $b != p_{(i+1) mod\ n}$ and intersect$(a, b, p_i, p_{(i+1) mod\ n})$:
    return False

//if we got here, we know that ab intersects no edge. The only thing left to check is whether it's inside or outside P

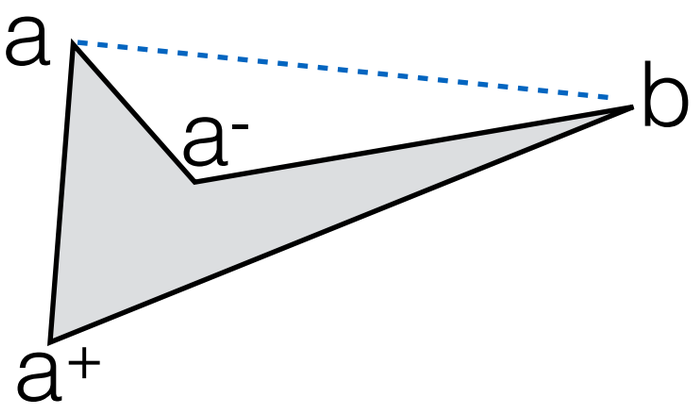
- return True if inside, False if outside

- So ab does not intersect any edges. Is ab interior or exterior?



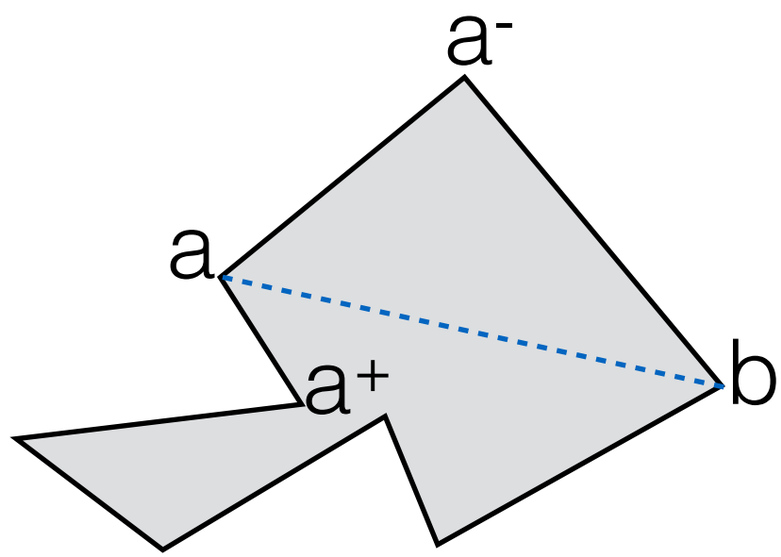


- So ab does not intersect any edges. Is ab interior or exterior?



not a diagonal

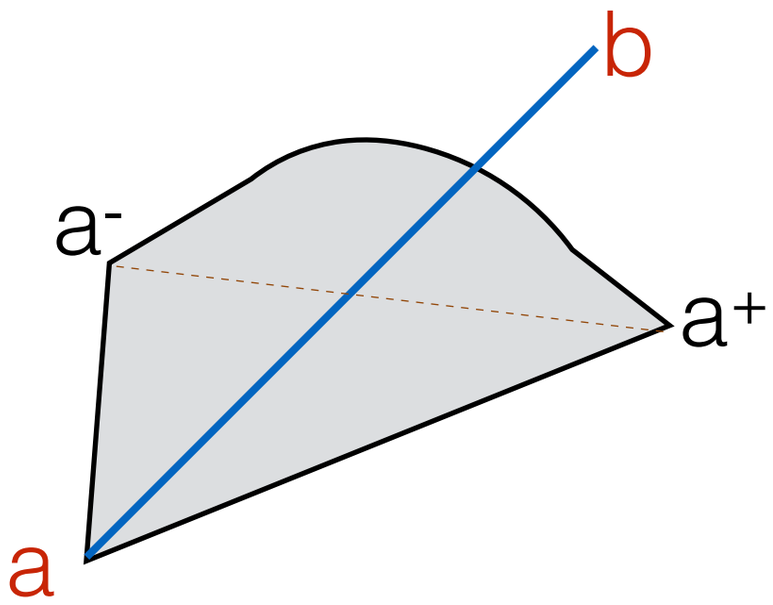
ab outside a^-, a, a^+

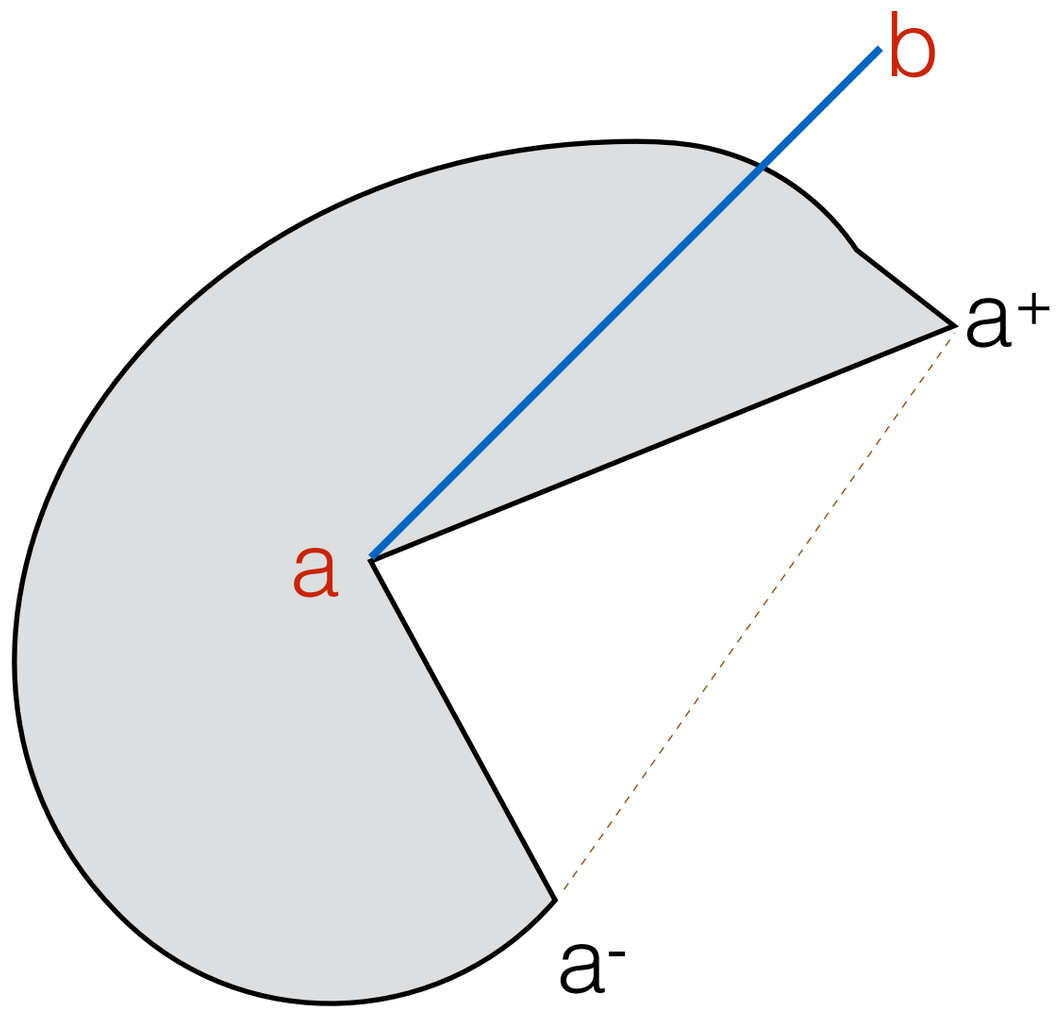


diagonal

ab is inside the cone formed by a^-, a, a^+

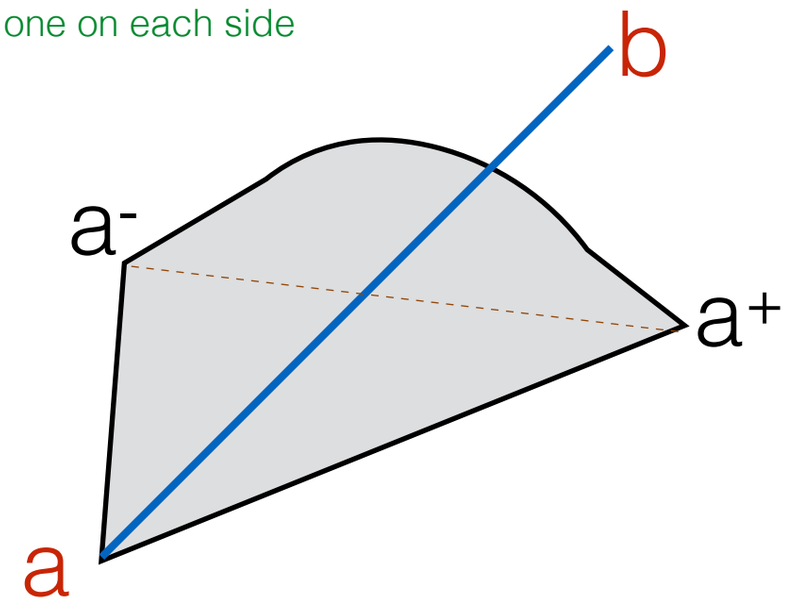
- InCone(a, b): return True if ab is in the cone determined by a^-, a, a^+

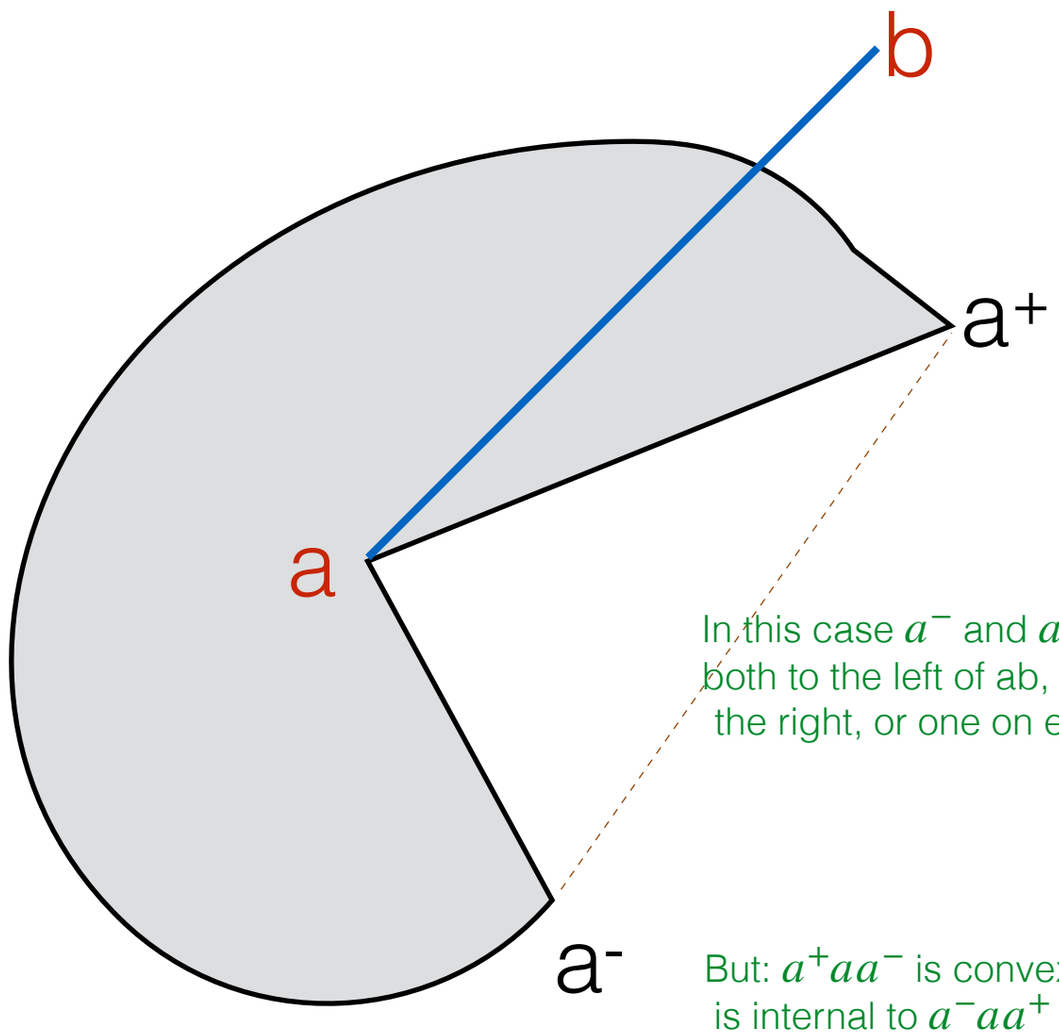




- InCone(a, b): return True if ab is in the cone determined by a^-, a, a^+

In this case a^- and a^+ must be one on each side

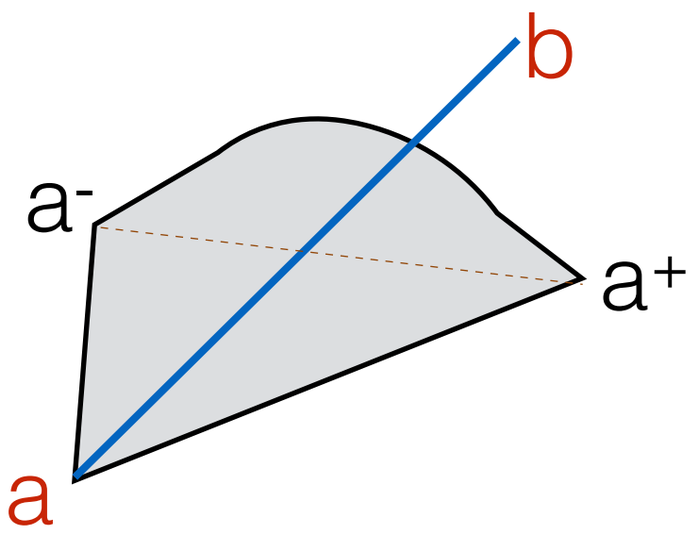


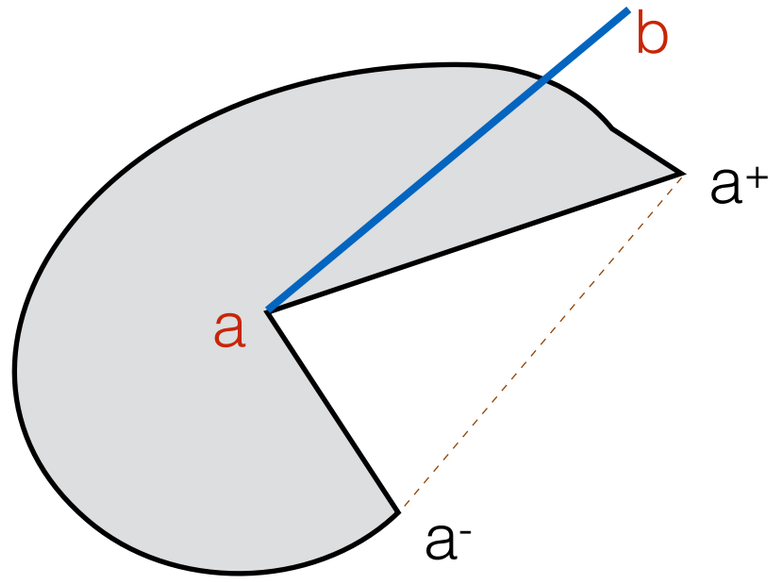


In this case a^- and a^+ may be both to the left of ab, or both to the right, or one on each side

But: a^+aa^- is convex, and ab is internal to a^-aa^+ if it is not internal to the convex a^+aa^-

- InCone(a, b): return True if ab is in the cone determined by $a\_, a, a_+$





//a, b must be points in P

- InCone(point2d a, b)
  - a^- = point before a
  - a^+ = point after a

Note: strict Left() to exclude ab collinear overlap with the cone

//if a is a convex vertex

- if LeftOn(a^-, a, a^+): return Left(a, b, a^-) && Left(b, a, a^+)

//else its a reflex vertex

- return !(LeftOn(a, b, a^+) && LeftOn(b, a, a^-))

Putting it all together: Is ab a diagonal?

//a, b must be points in P

isDiagonal(a,b):

//does any of the edges intersect ab?

- for i=0; i< n; i++

O(n) - if $a != p_i$ and $a != p_{(i+1) mod\ n}$ and $b != p_i$ and $b != p_{(i+1) mod\ n}$ and intersect$(a, b, p_i, p_{(i+1) mod\ n})$:

return False

//if we got here, we know that ab intersects no edge properly. The only thing left to check is whether it's inside or outside P

O(1) - return inCone(a, b) && InCone(b, a)

Overall: O(n) time

In conclusion,

we can test if ab is a diagonal in O(n) time

Straightforward way to find a diagonal:

- for i=0, i<n, i++
  - for j=i+1, j<n, j++
    - check if $p_i p_j$ is diagonal

Naive triangulation by recursively finding diagonals

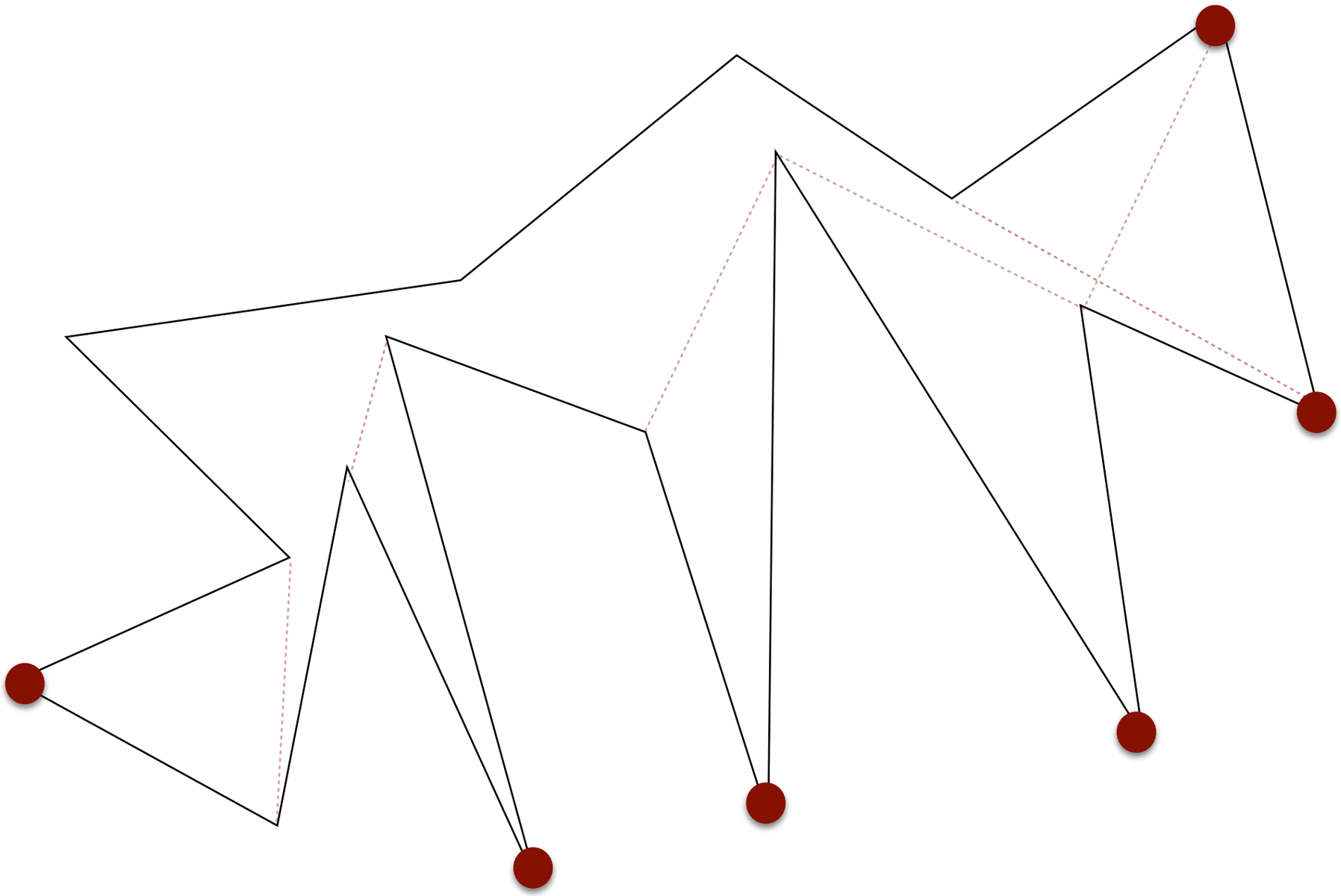
- **Algorithm 1:** Triangulation by finding diagonals
  - Idea: Check all pairs of vertices to find one which is a diagonal; recurse.
  - Analysis:
    - checking all vertices: $O(n^2)$ candidates for diagonals, checking each takes O(n), overall $O(n^3)$
    - recurse, worst case on a problem of size n-1
      - overall $O(n^4)$

- **Algorithm 2:** Triangulation by *smartly* finding diagonals
  - A diagonal can be found in O(n) time (using the proof that a diagonal exists)
  - Idea: Find a diagonal, output it, recurse.
  - $O(n^2)$

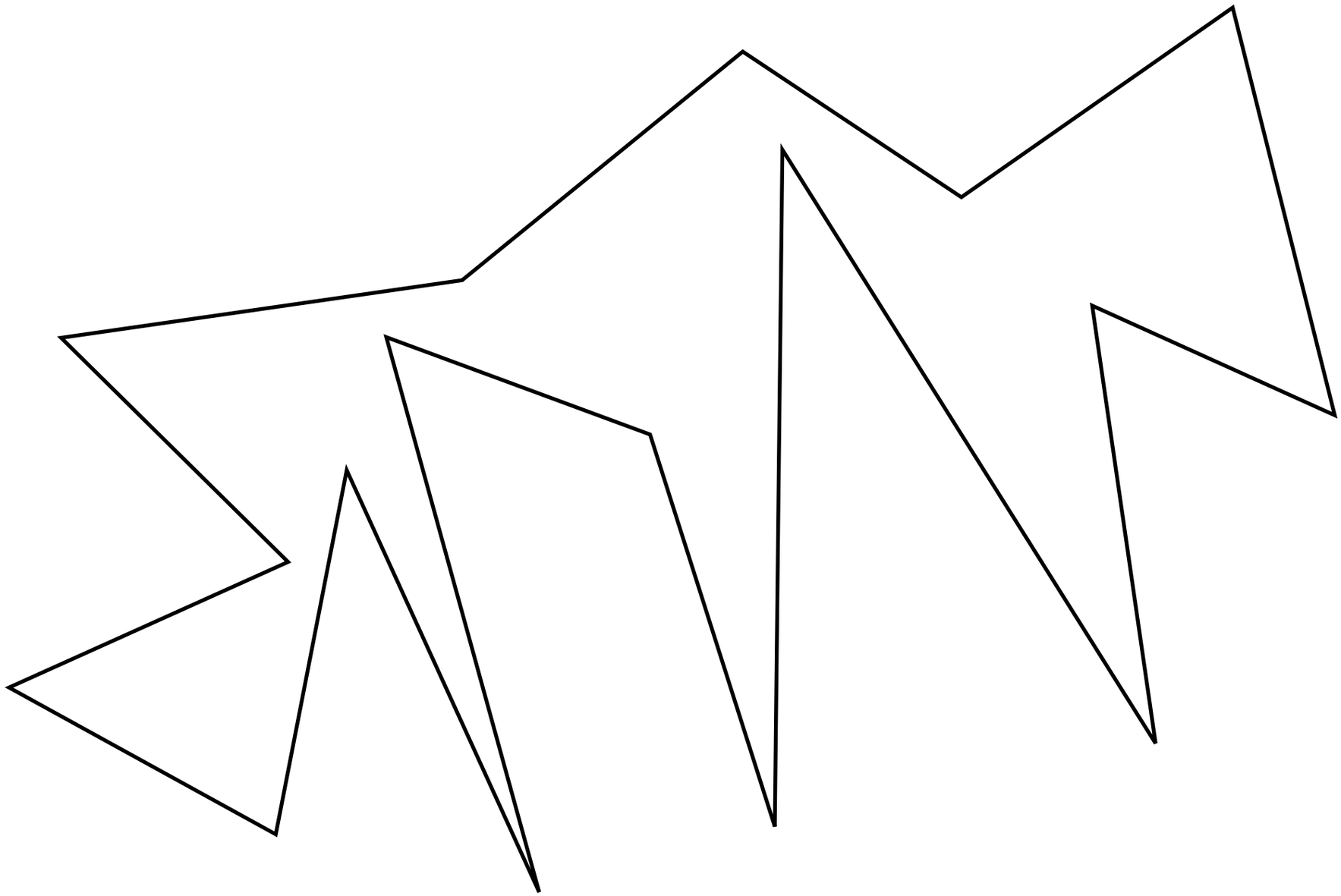
Algorithm 3: Triangulation by finding ears

A bat-eared fox peeks from the grass in Hwange National Park, Zimbabwe.
PHOTOGRAPH BY ROY TOFT, NAT GEO IMAGE COLLECTION

A vertex p of P is called **ear** if $p^- p^+$ is a diagonal

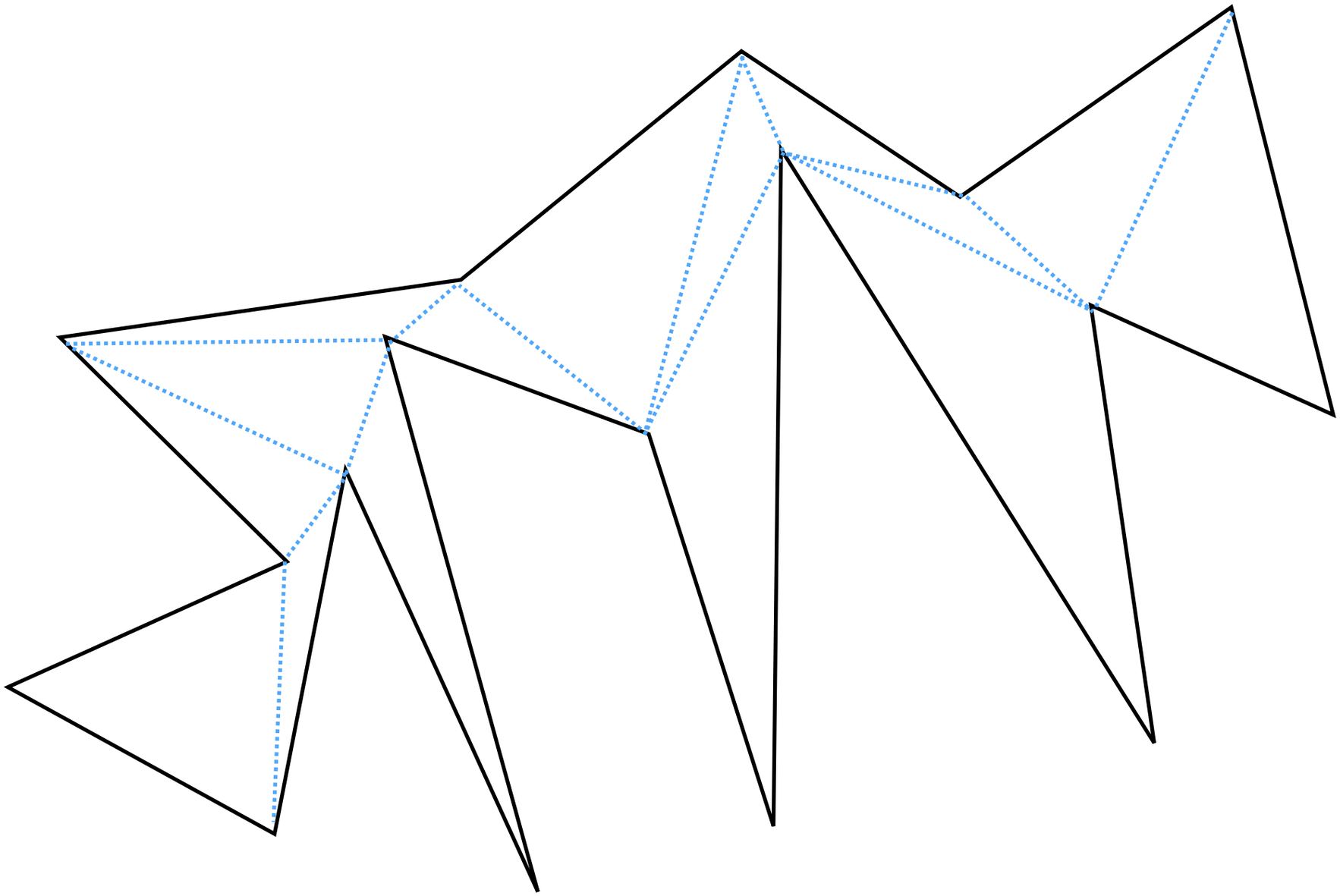


Theorem: Any simple polygon has at least two ears.



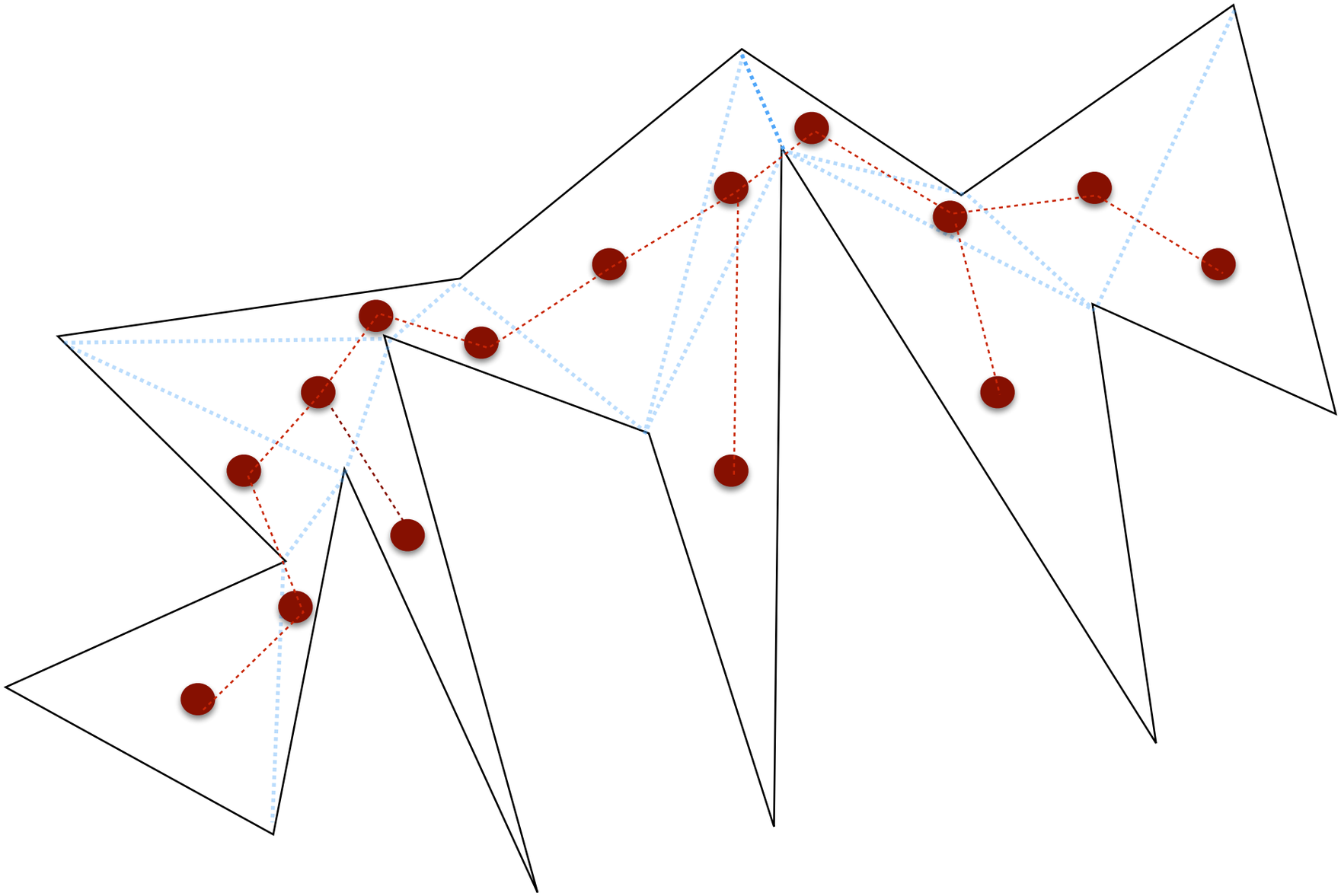
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P.



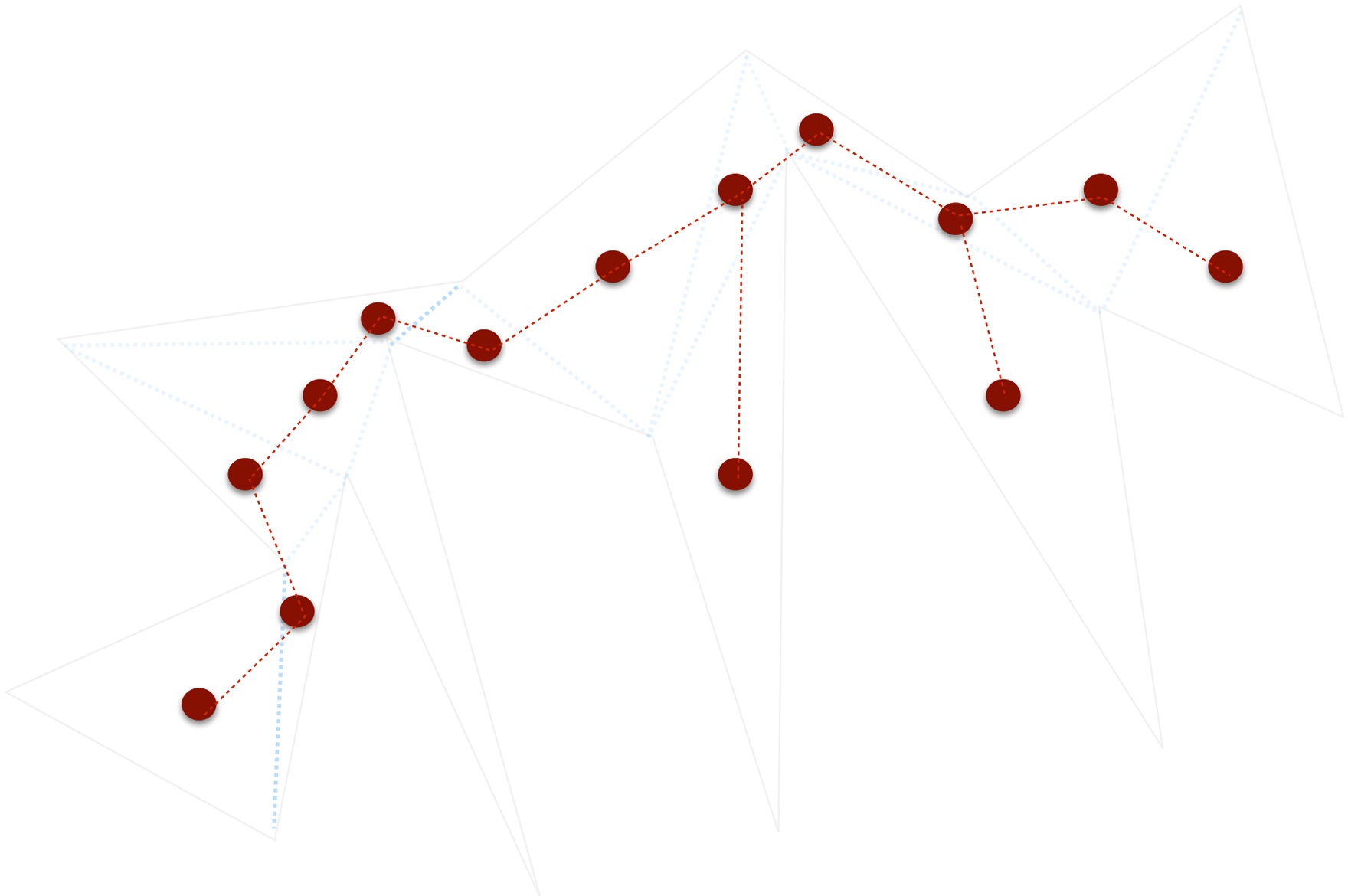
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P. Consider the dual graph.



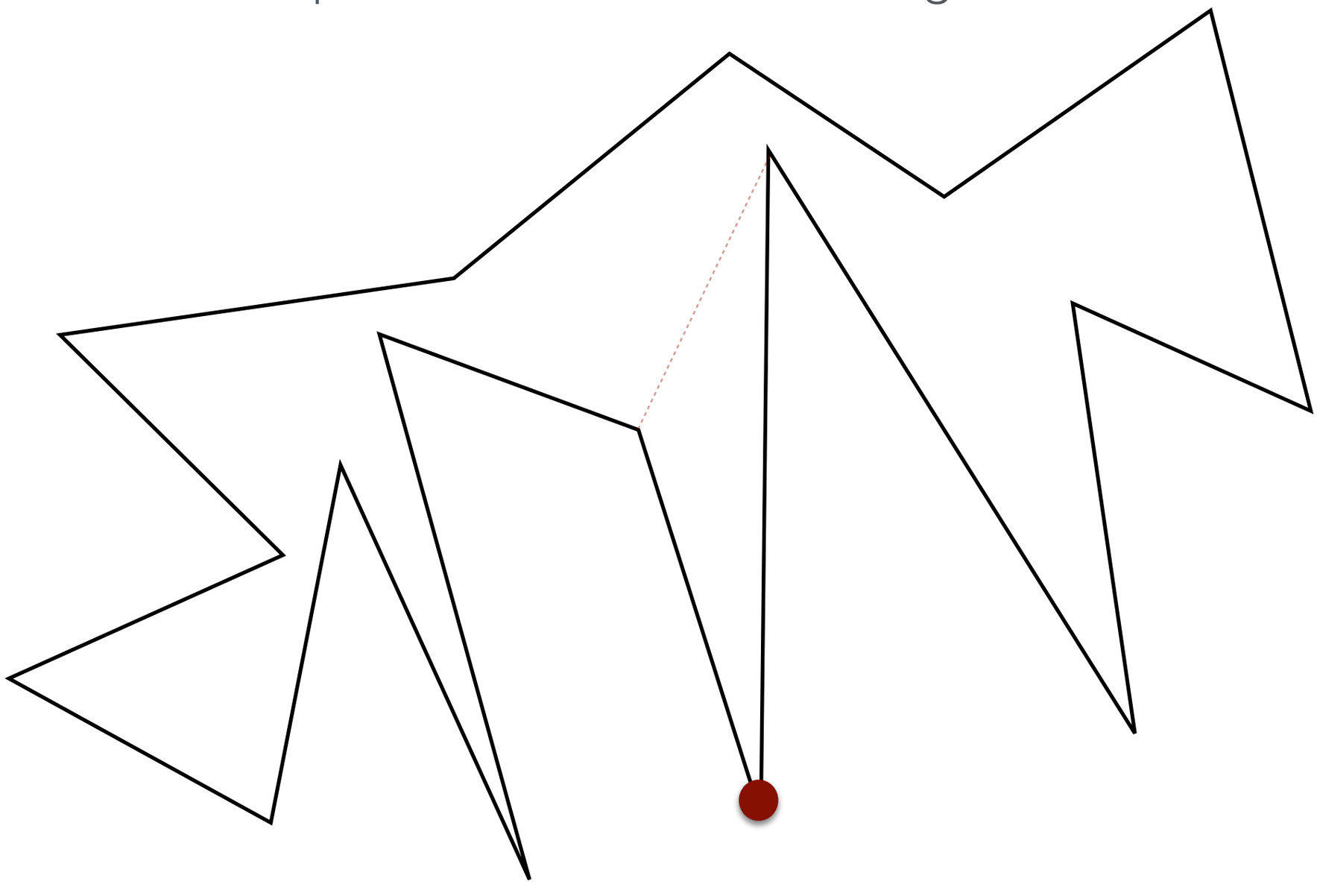
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P. Consider the dual graph. The dual graph is a tree. Any tree has at least two leaves. A leaf => ear



Algorithm 3: Triangulation by finding ears

- Traverse P and for each point p, determine if it's an ear
- When find a ear p: recurse on the remaining P



Algorithm 3: Triangulation by finding ears

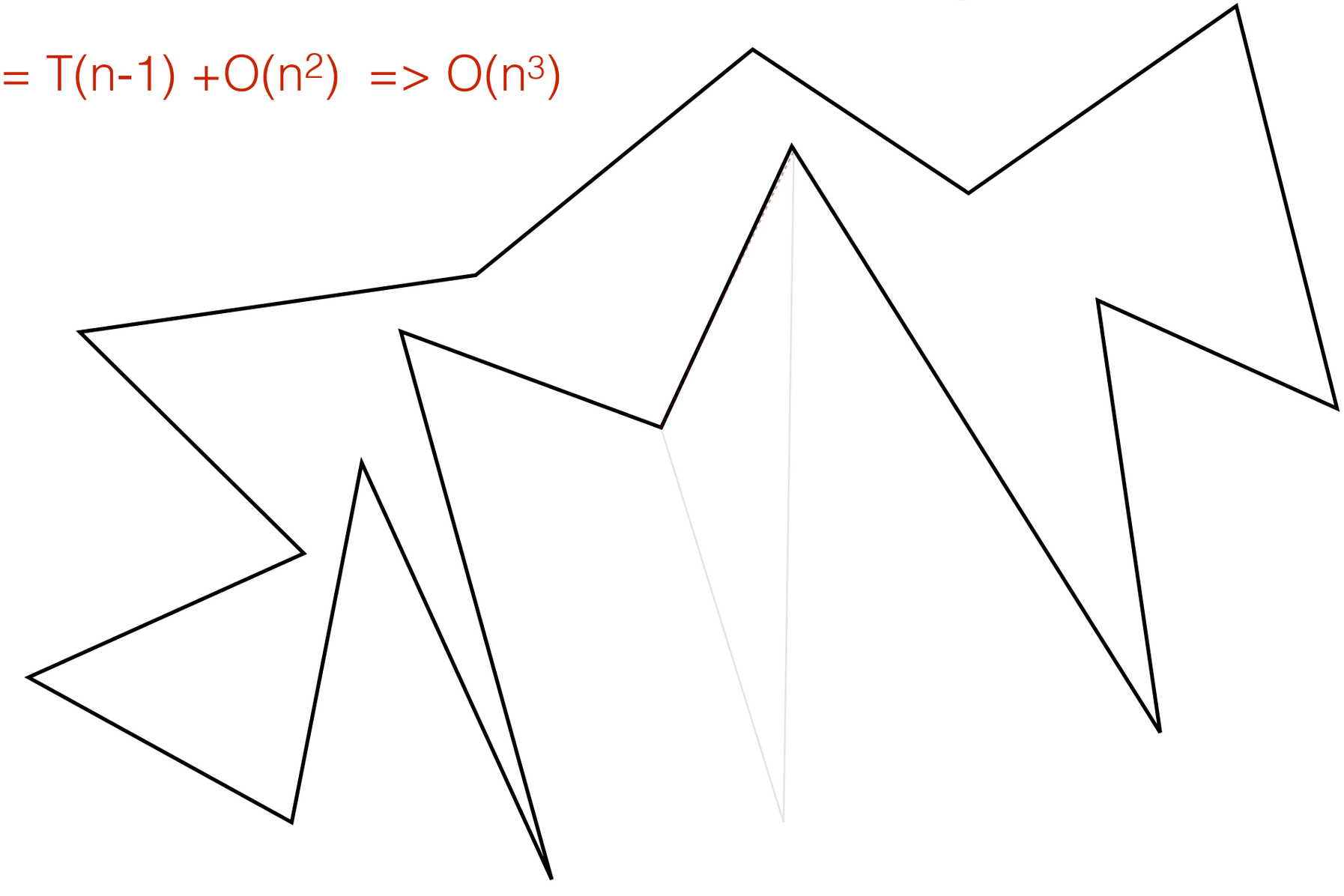
O(n)

- Traverse P and for each point p, determine if it's an ear

O(n)

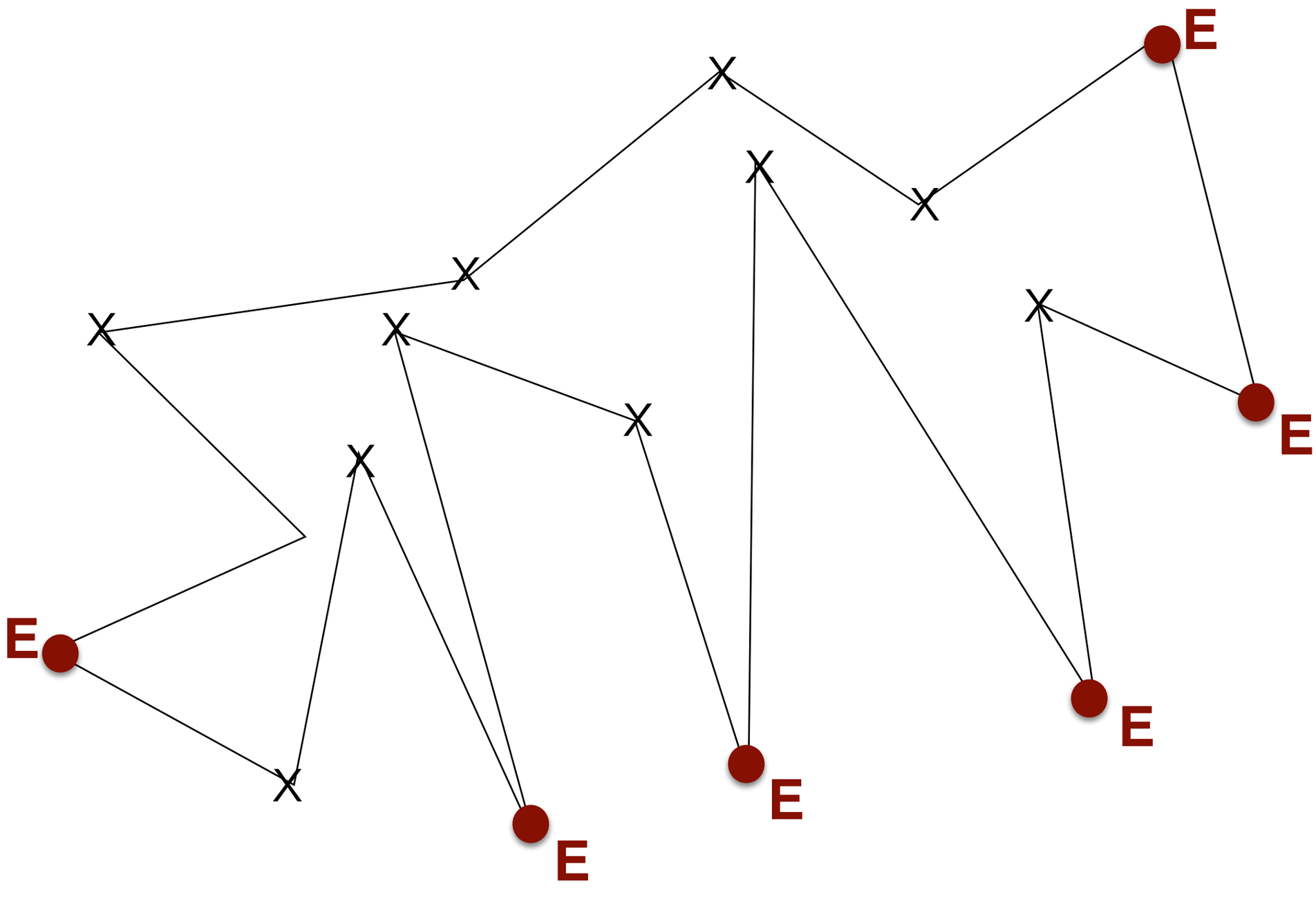
- When find a ear p: recurse on the remaining P

$T(n) = T(n-1) + O(n^2) \Rightarrow O(n^3)$



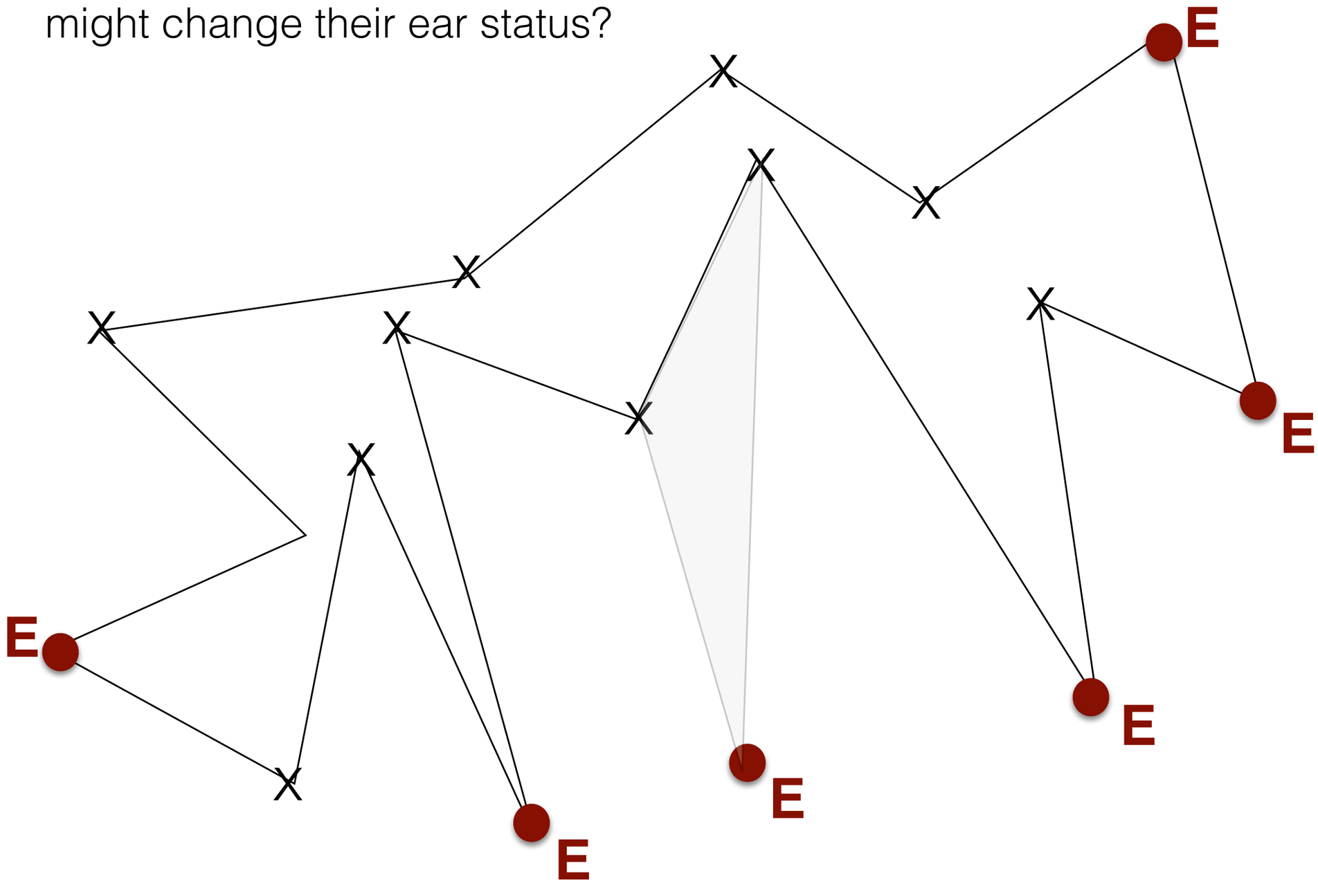
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time



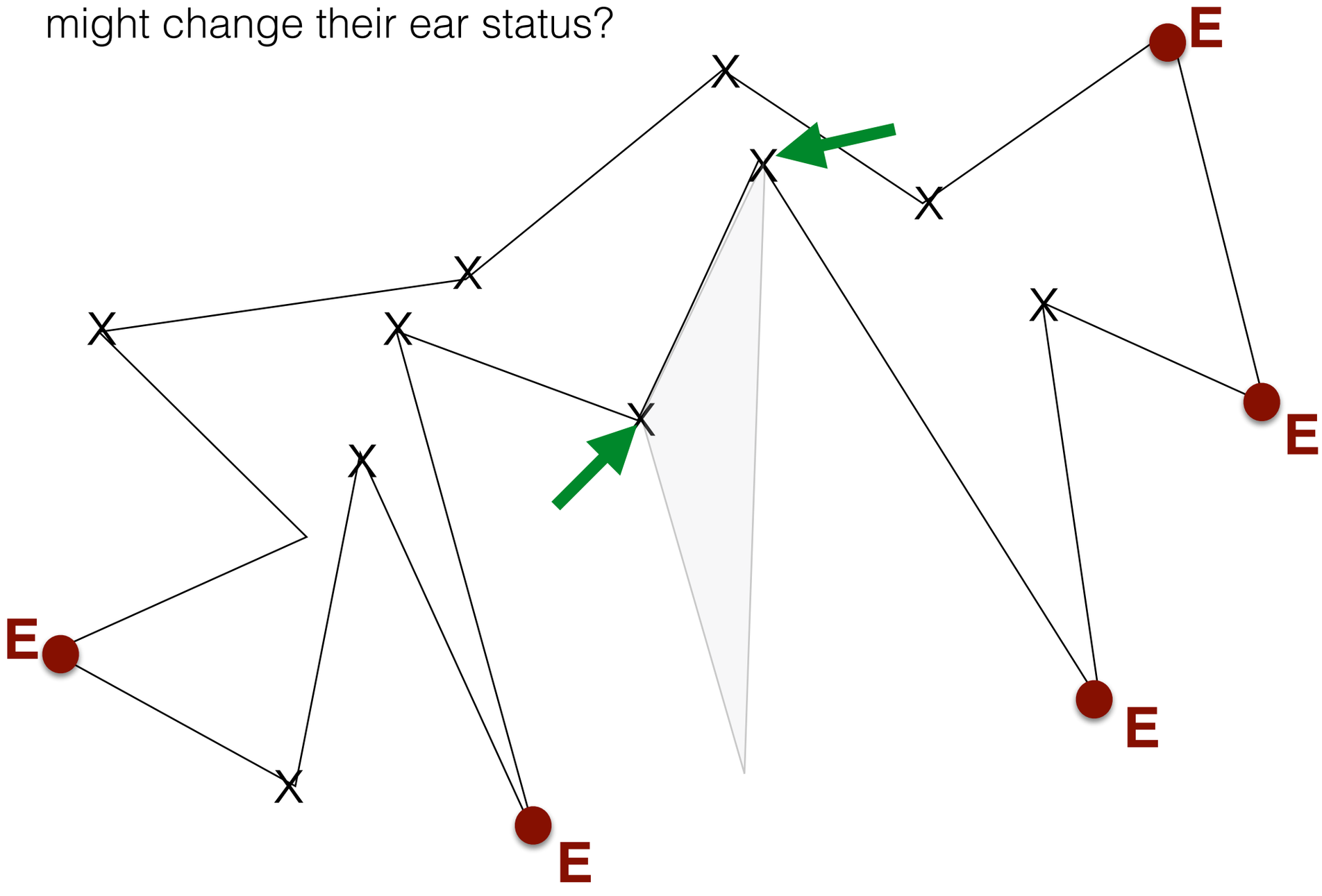
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



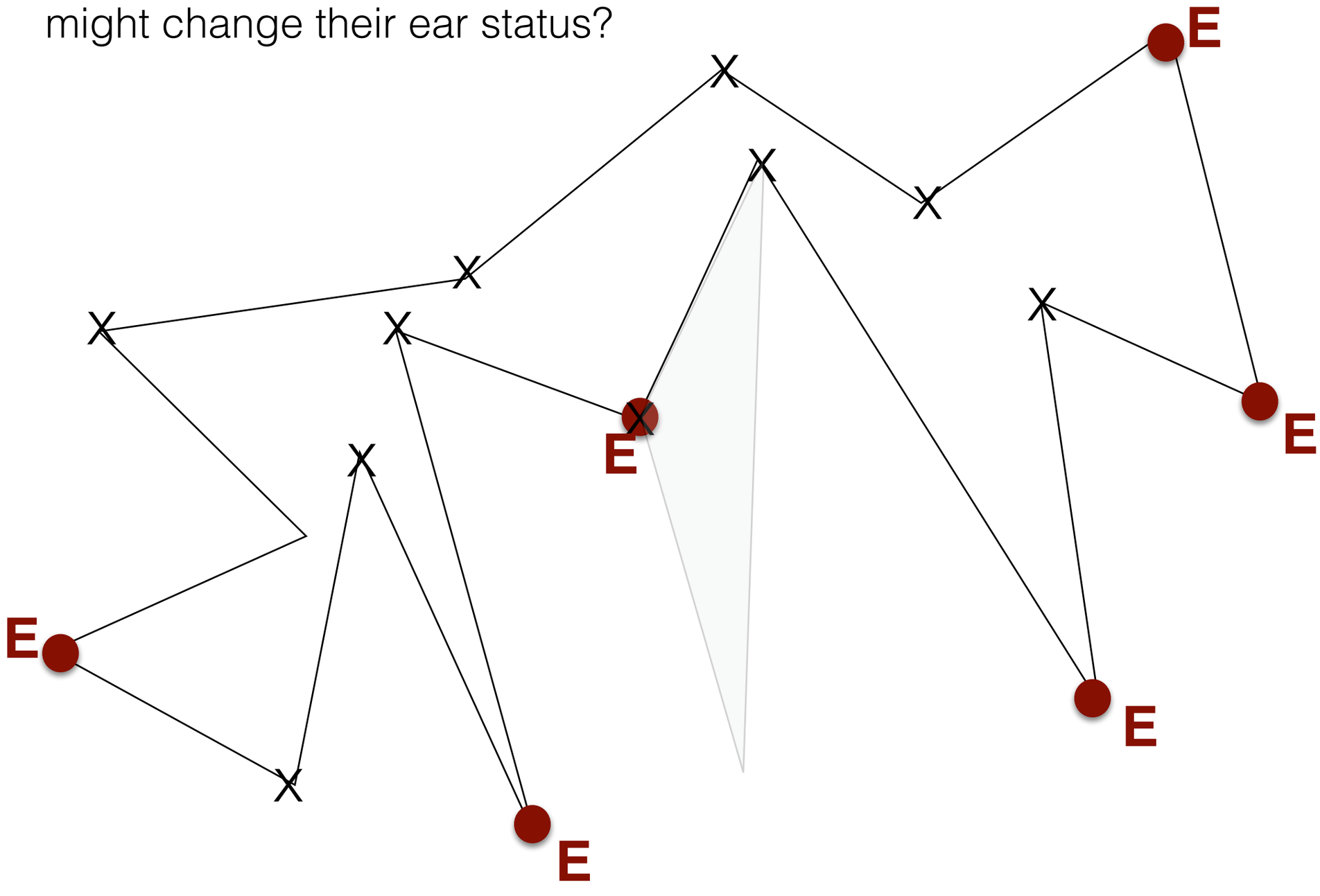
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



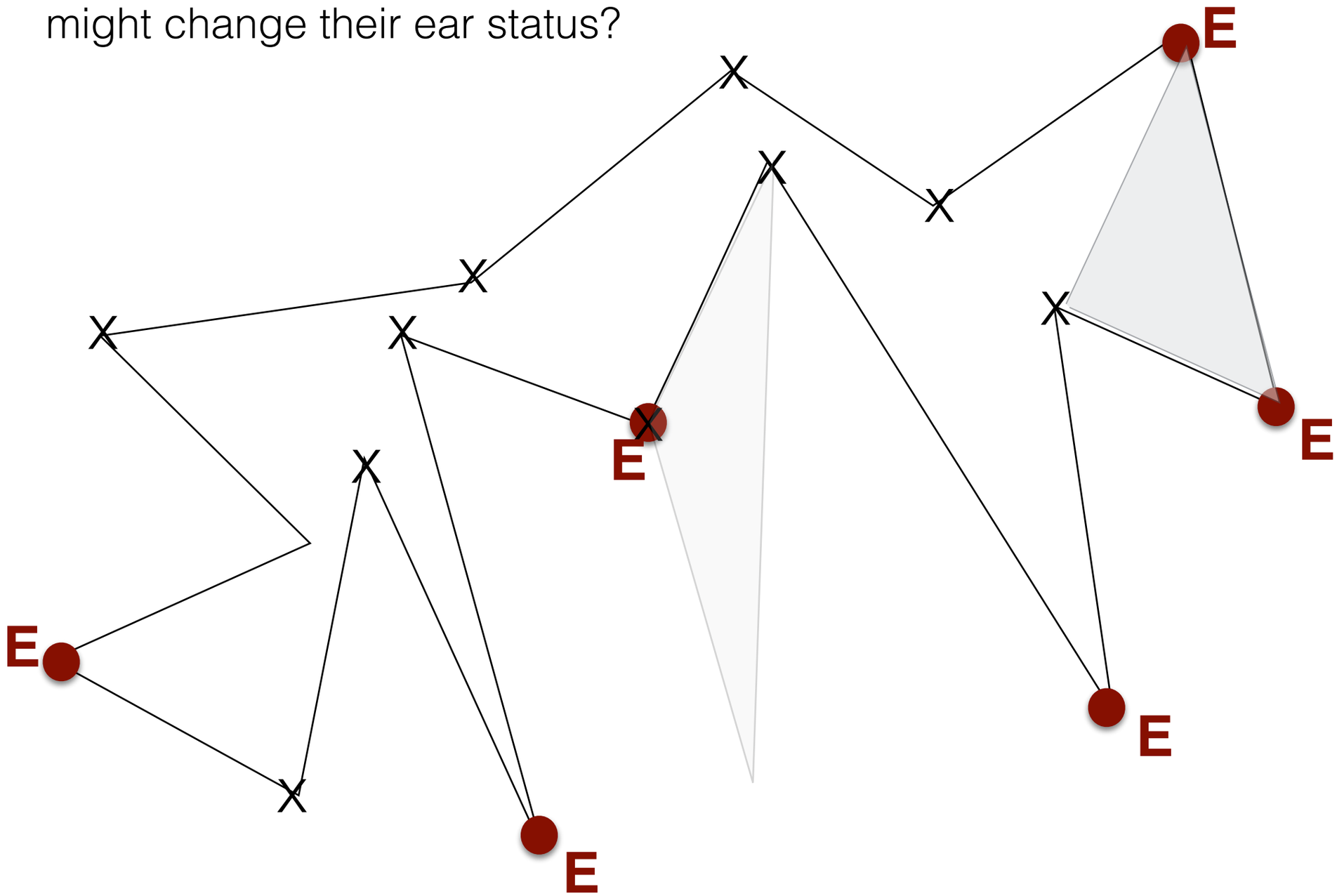
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



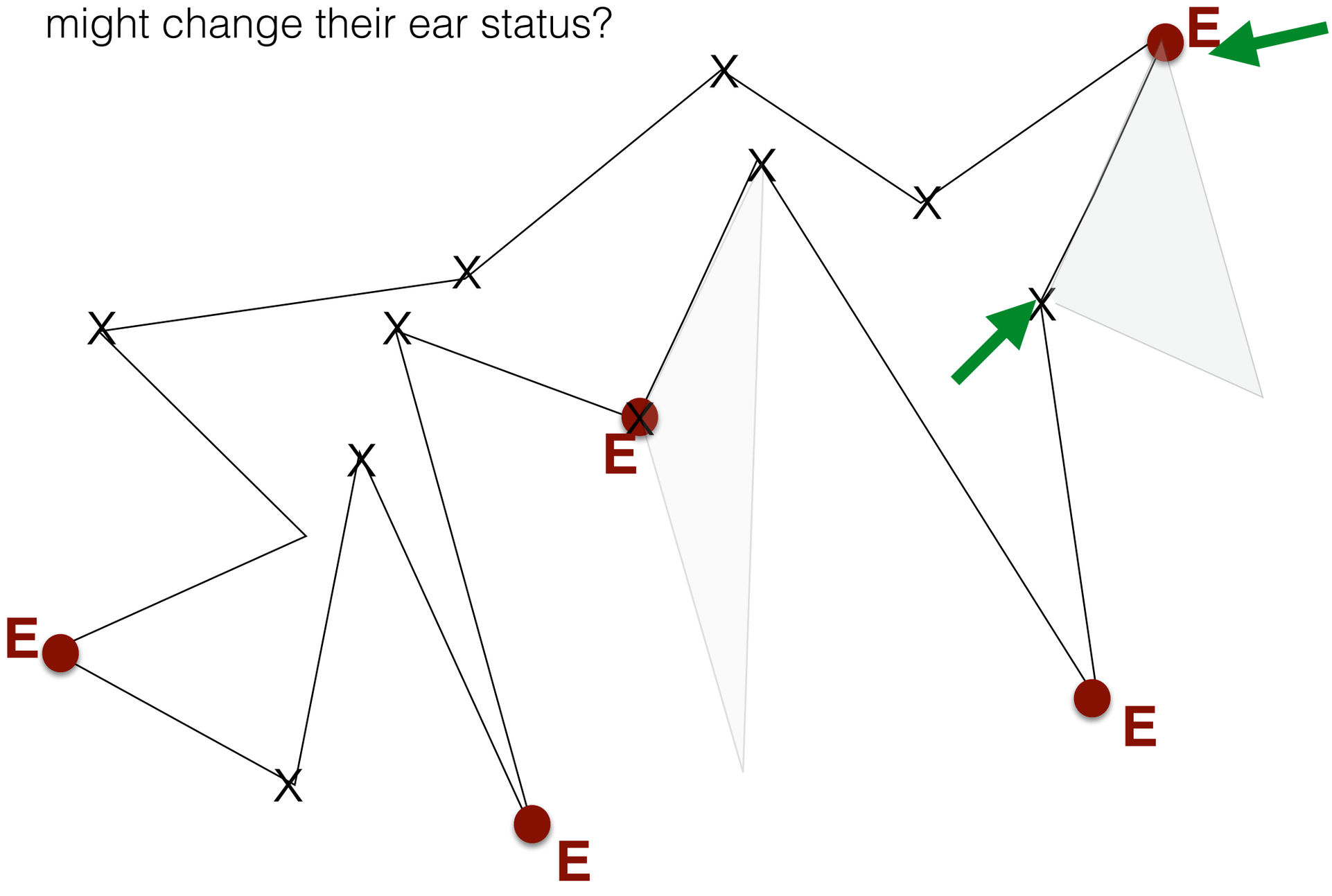
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



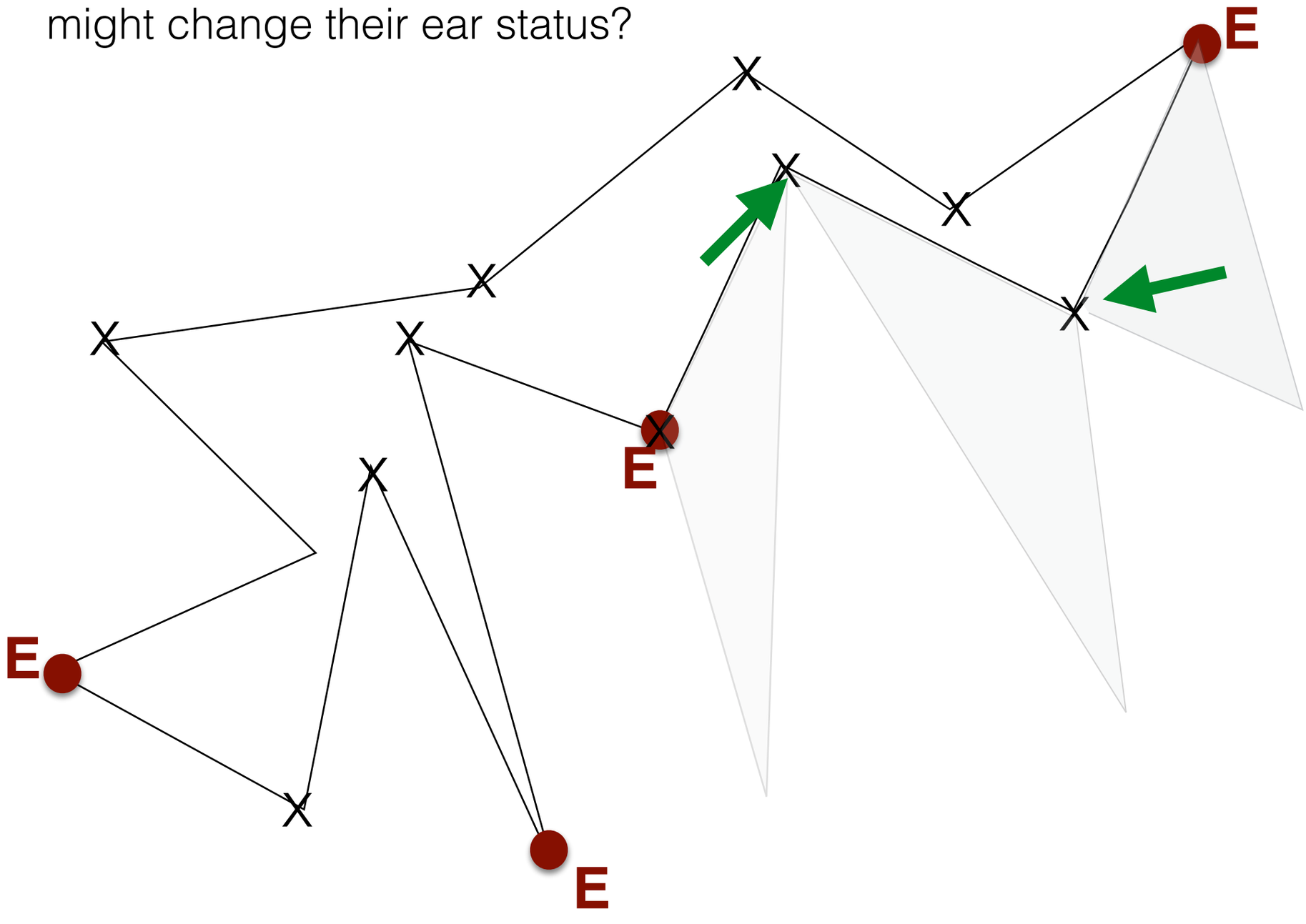
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



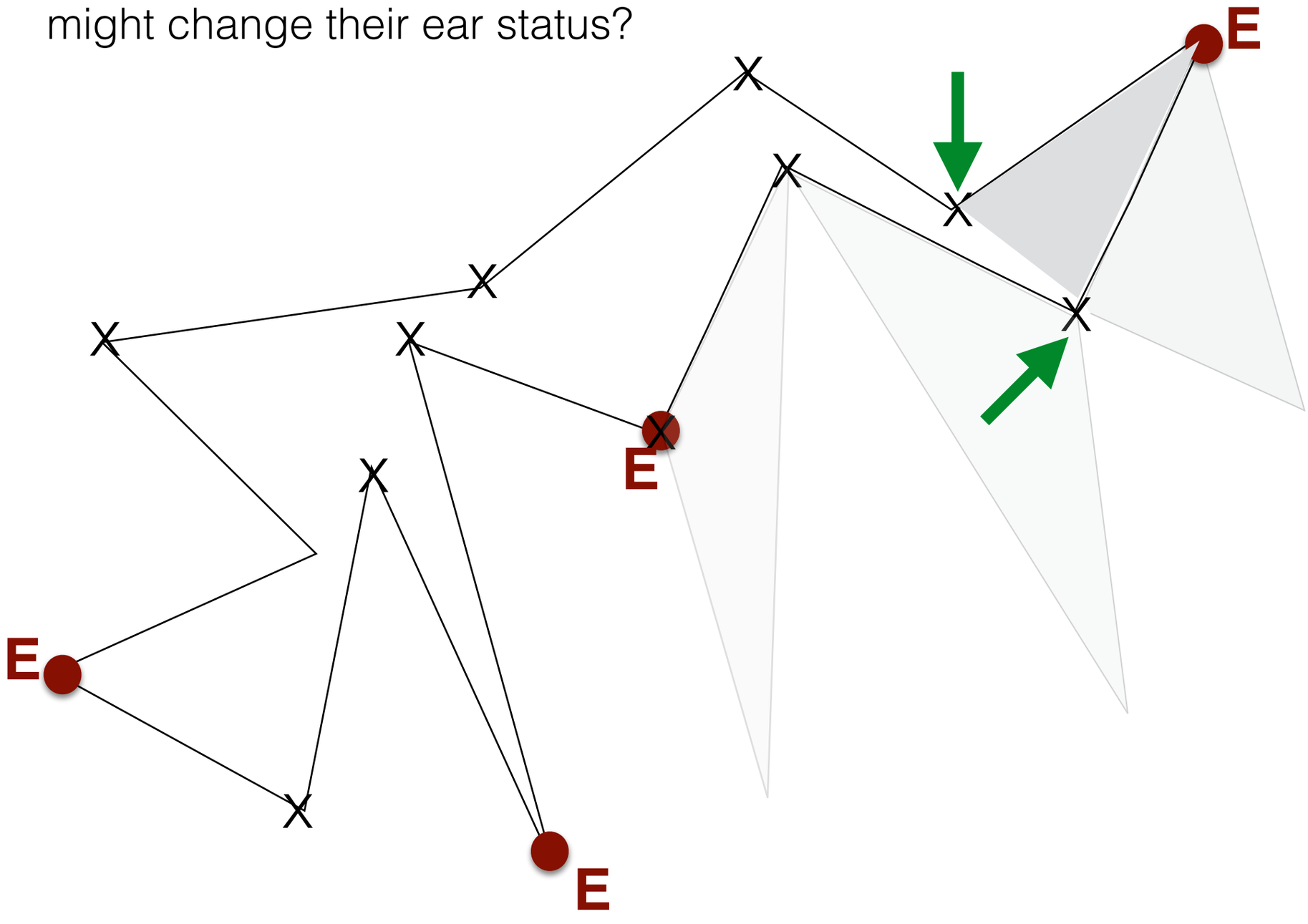
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



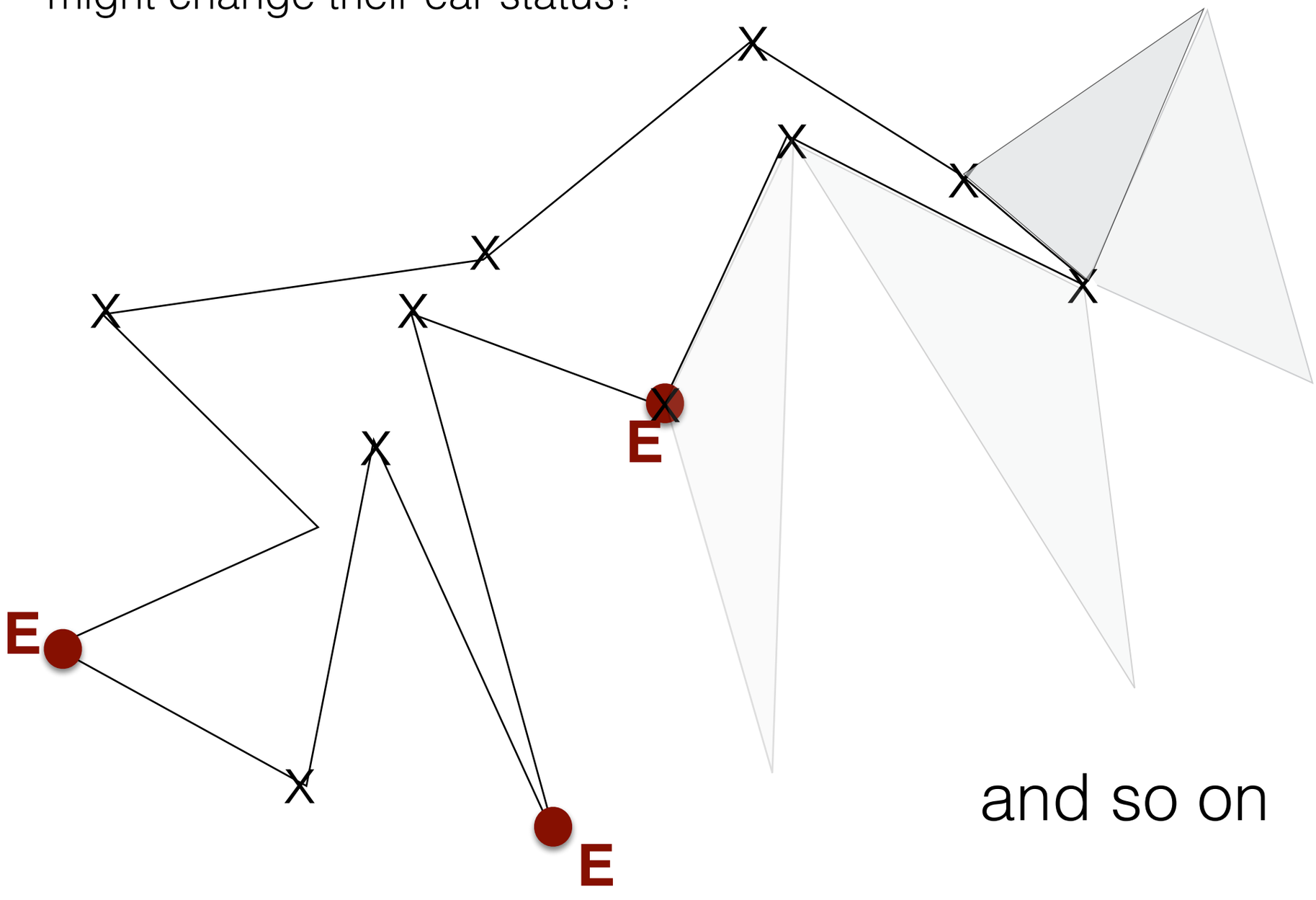
Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



Algorithm 4: Improved ear removal

- **Idea:** Avoid recomputing ear status for all vertices every time
  - When you remove an ear tip from the polygon, which vertices might change their ear status?



and so on

Algorithm 4: Improved ear removal

- Initialize the ear tip status of each vertex of P
- while n>3 do:
    - locate an ear tip p
    - output diagonal p^-p^+
    - delete p
    - update ear tip status of p^- and p^+

Or, with a bit more detail,

Algorithm 4: Improved ear removal

//Initialize the ear tip status of each vertex of P

- for i=0, i<n, i++
    - $p\_$ is ear if isDiagonal($p^- p^+$)

- while n>3 do:
    - i=0
    - while i < P.size():
        - if p[i] is ear:
            - output diagonal $p[i-1]p[i+1]$
            - update ear status for $p[i-1]$ and $p[i+1]$
            - delete p[i] from P and set n = n-1
        - else: i++

Algorithm 4: Improved ear removal

//Initialize the ear tip status of each vertex of P

- for i=0, i<n, i++
    - $p\_$ is ear if isDiagonal($p^- p^+$)

O(n²)

- while n>3 do:
    - i=0
    - while i < P.size():
        - if p[i] is ear:
            - output diagonal $p[i-1]p[i+1]$
            - update ear status for $p[i-1]$ and $p[i+1]$ ← this takes O(n)
            - delete p[i] from P and set n = n-1
        - else: i++

a vertex causes ear status updates for 2 other vertices

O(n) ear status updates

Overall: O(n²) time

History of Polygon Triangulation

- Early algorithms: $O(n^4)$, $O(n^3)$, $O(n^2)$
- Several O(n lg n) algorithms known ← practical
- …
- Many papers with improved bounds ← not practical
- …
- 1991: Bernard Chazelle (Princeton) gave an O(n) algorithm
  - https://www.cs.princeton.edu/~chazelle/pubs/polygon-triang.pdf
  - Ridiculously complicated, not practical
  - O(1) people actually understand it (seriously) (and I'm not one of them)

- No algorithm is known that is practical enough to run faster than the O(n lg n) algorithms
- OPEN problem
  - A practical algorithm that's theoretically better than O(n lg n).

An O(n lg n) Polygon Triangulation Algorithm

- Ingredients
  - Triangulate **monotone/unimonotone** polygons
  - Convert an arbitrary polygon into monotone/unimonotone polygons

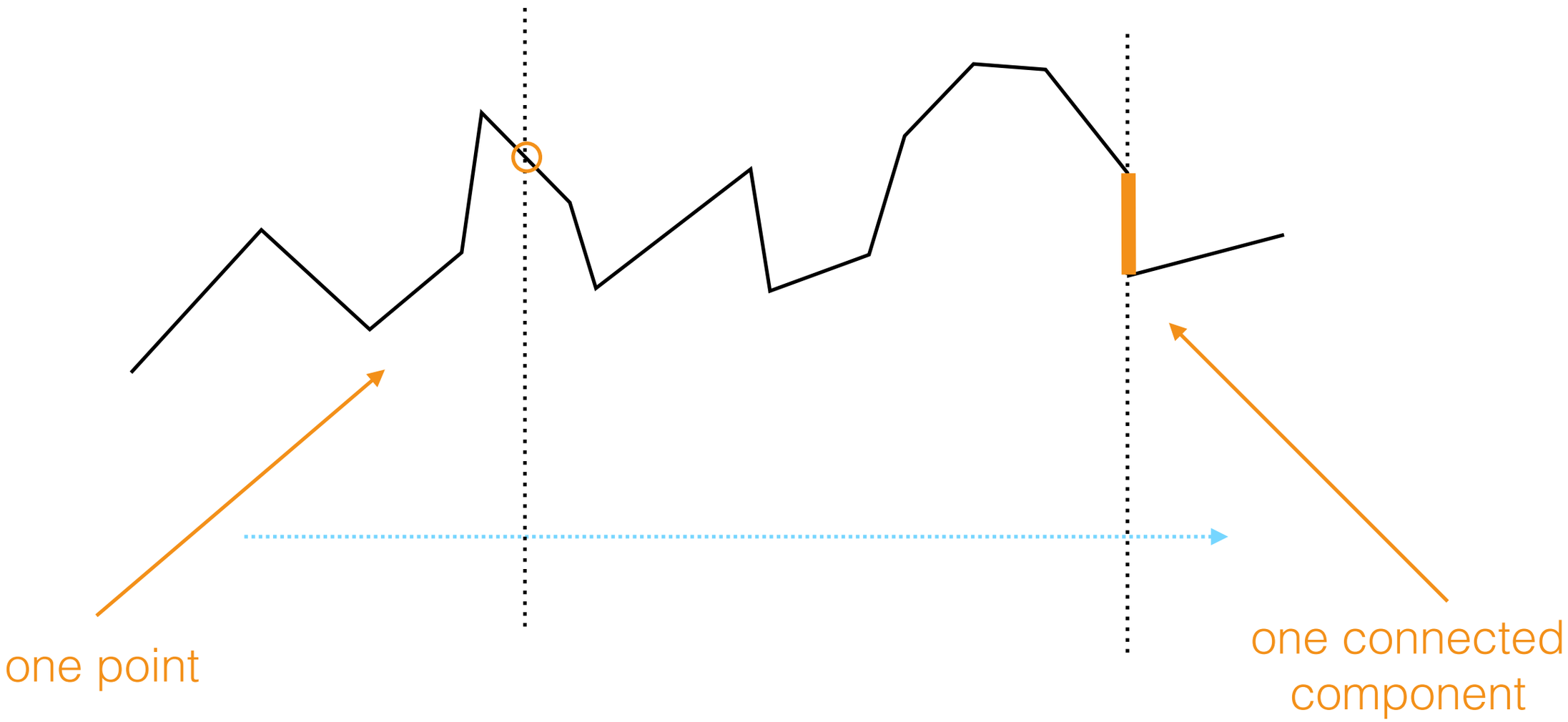
Monotone chains

- Let P be a vector of points, not necessary a polygon (polygonal chain)

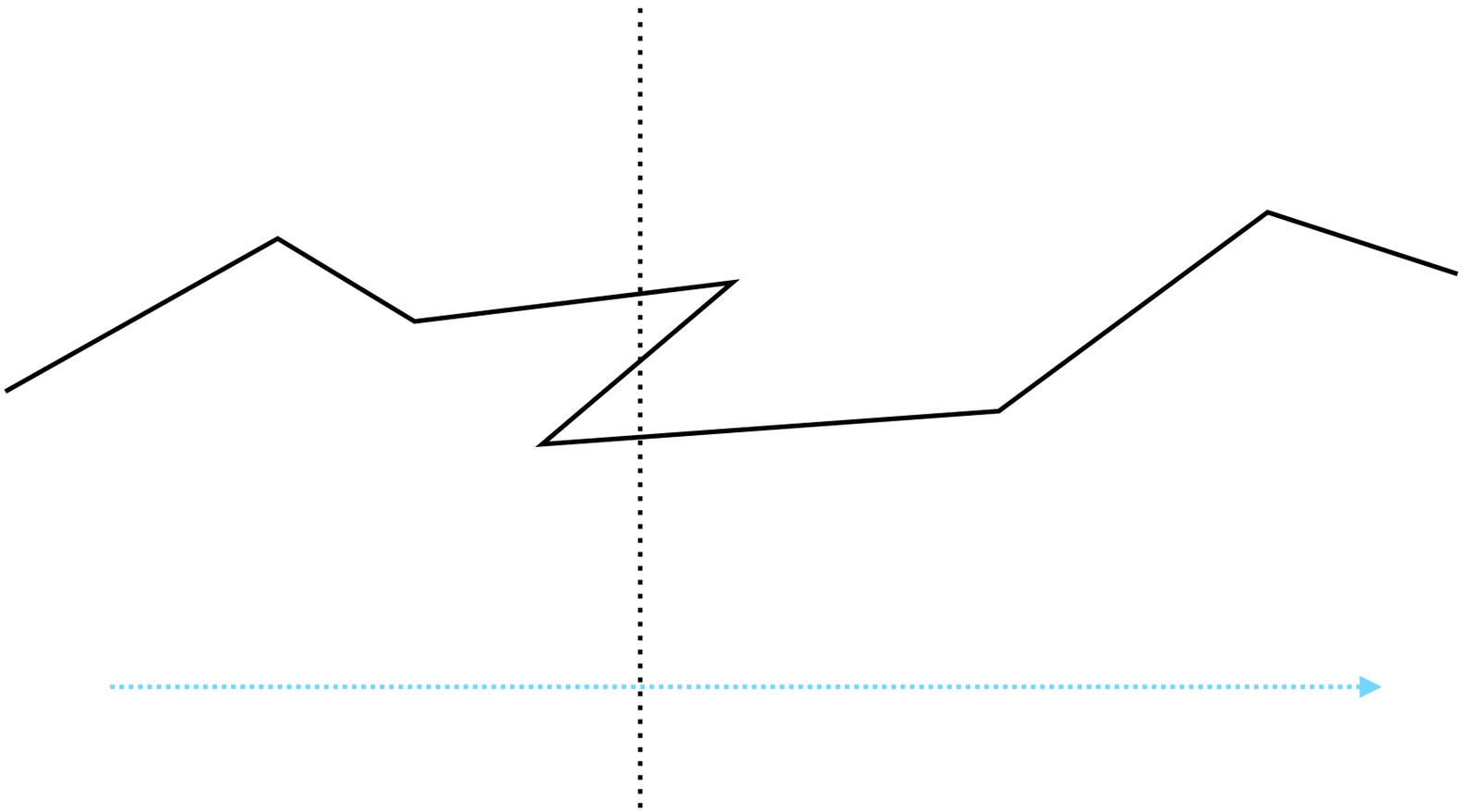
A polygonal chain is **x-monotone** if any line perpendicular to x-axis intersects it in one point (one connected component).

Monotone chains

A polygonal chain is **x-monotone** if any line perpendicular to **x-axis** intersects it in one point (one connected component).



Monotone chains

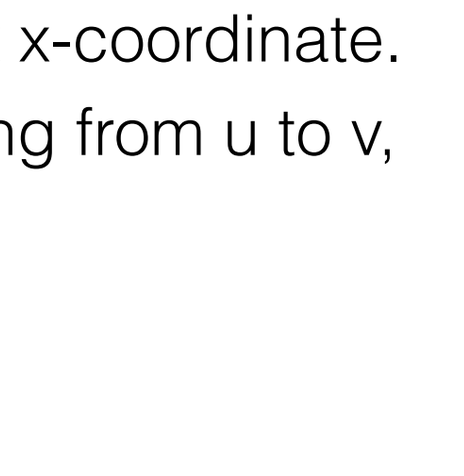


Not x-monotone

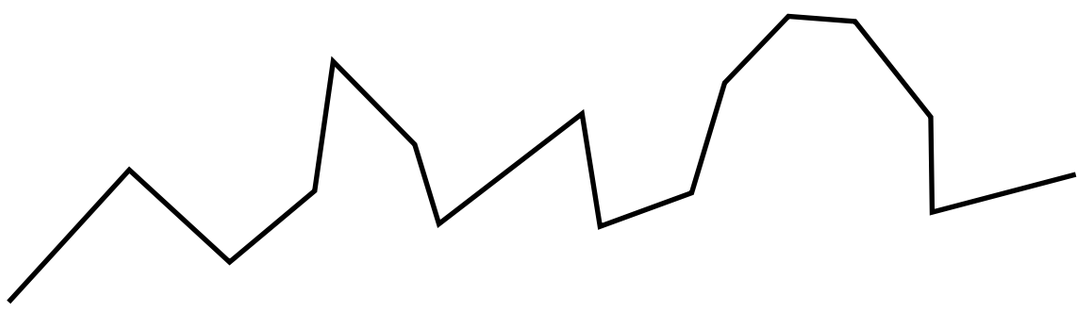
Monotone chains

- Claim: Let u and v be the points on the chain with min/max x-coordinate. The vertices on the boundary of an x-monotone chain, going from u to v, are in x-order.

x-monotone



Monotone chains



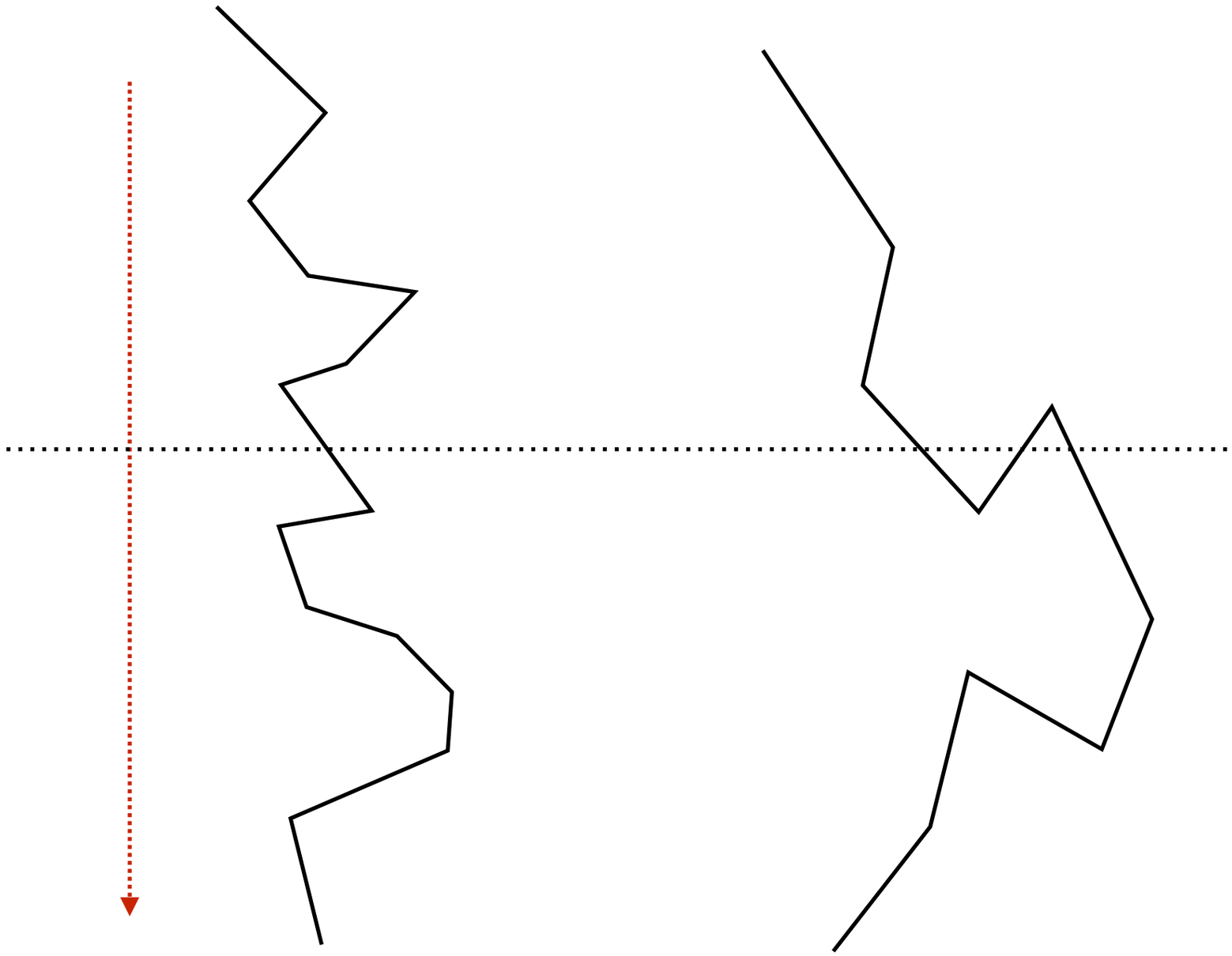
x-monotone

not x-monotone

As you travel along this chain, your x-coordinate is staying the same or increasing

In general..

A polygonal chain is **y-monotone** if any line perpendicular to **y-axis** intersects it in one point (one connected component).

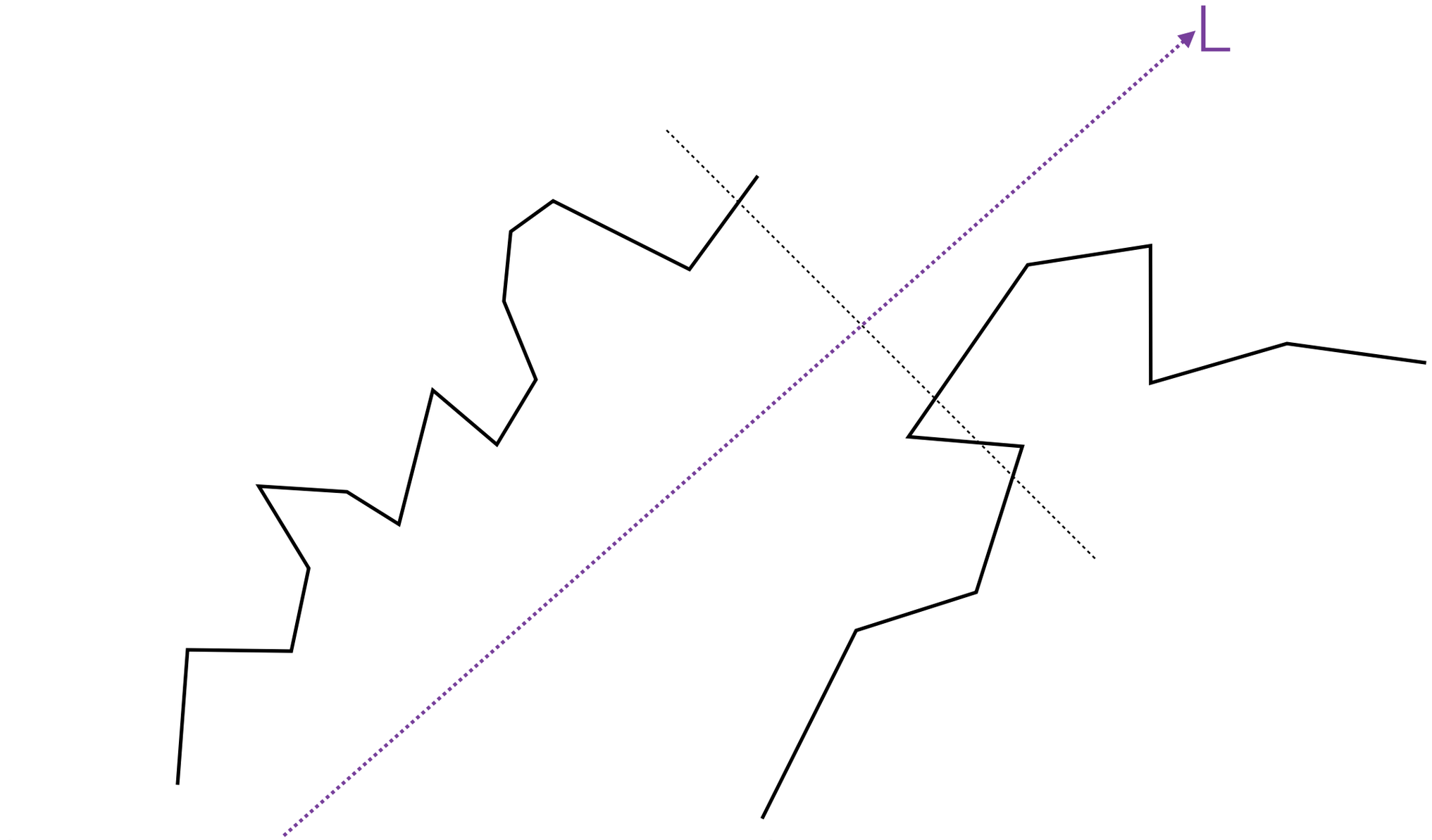


y-monotone

not y-monotone

In general..

A polygonal chain is **L-monotone** if any line perpendicular to **line L** intersects it in one point (one connected component).

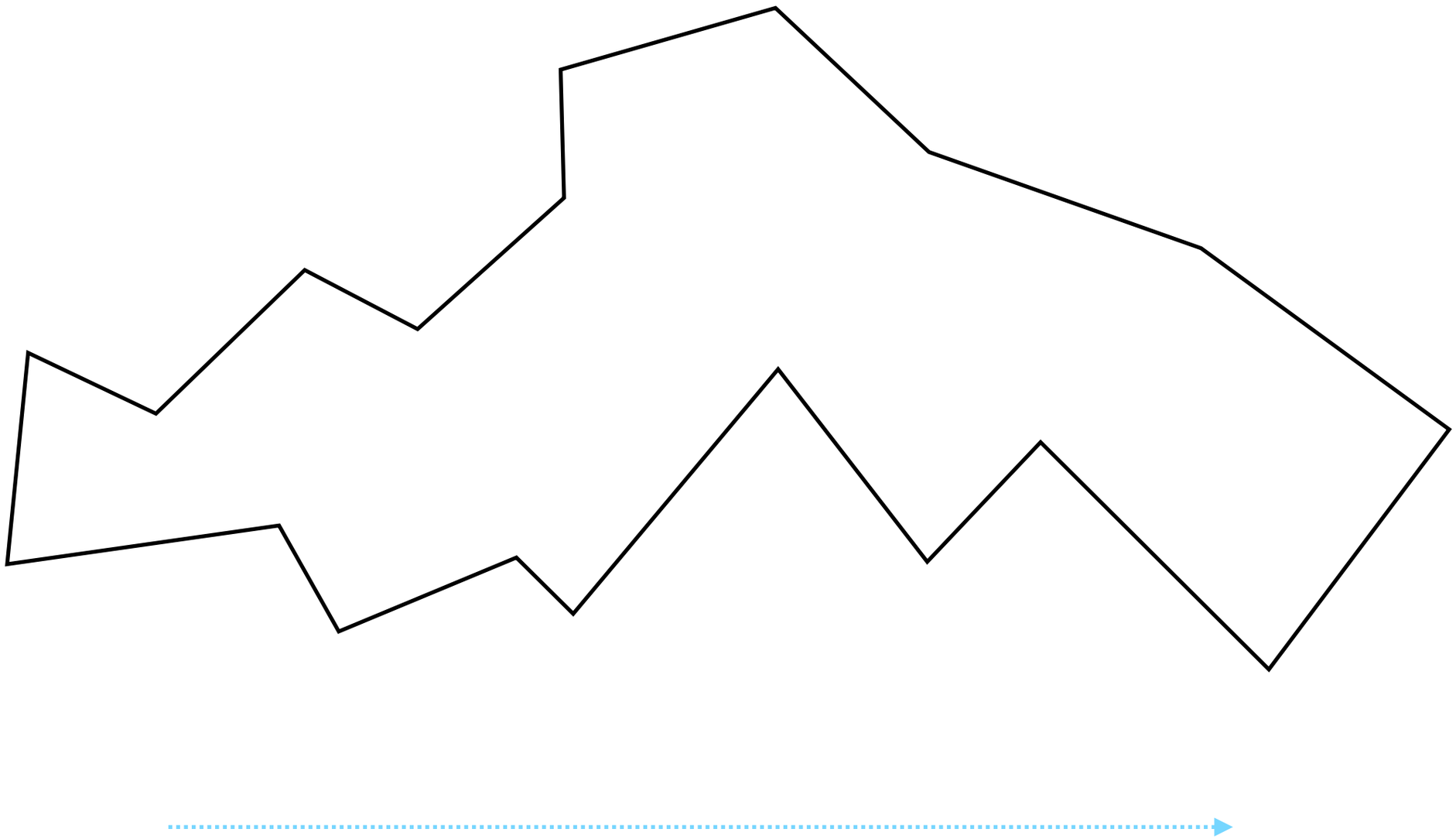


L-monotone

not L-monotone

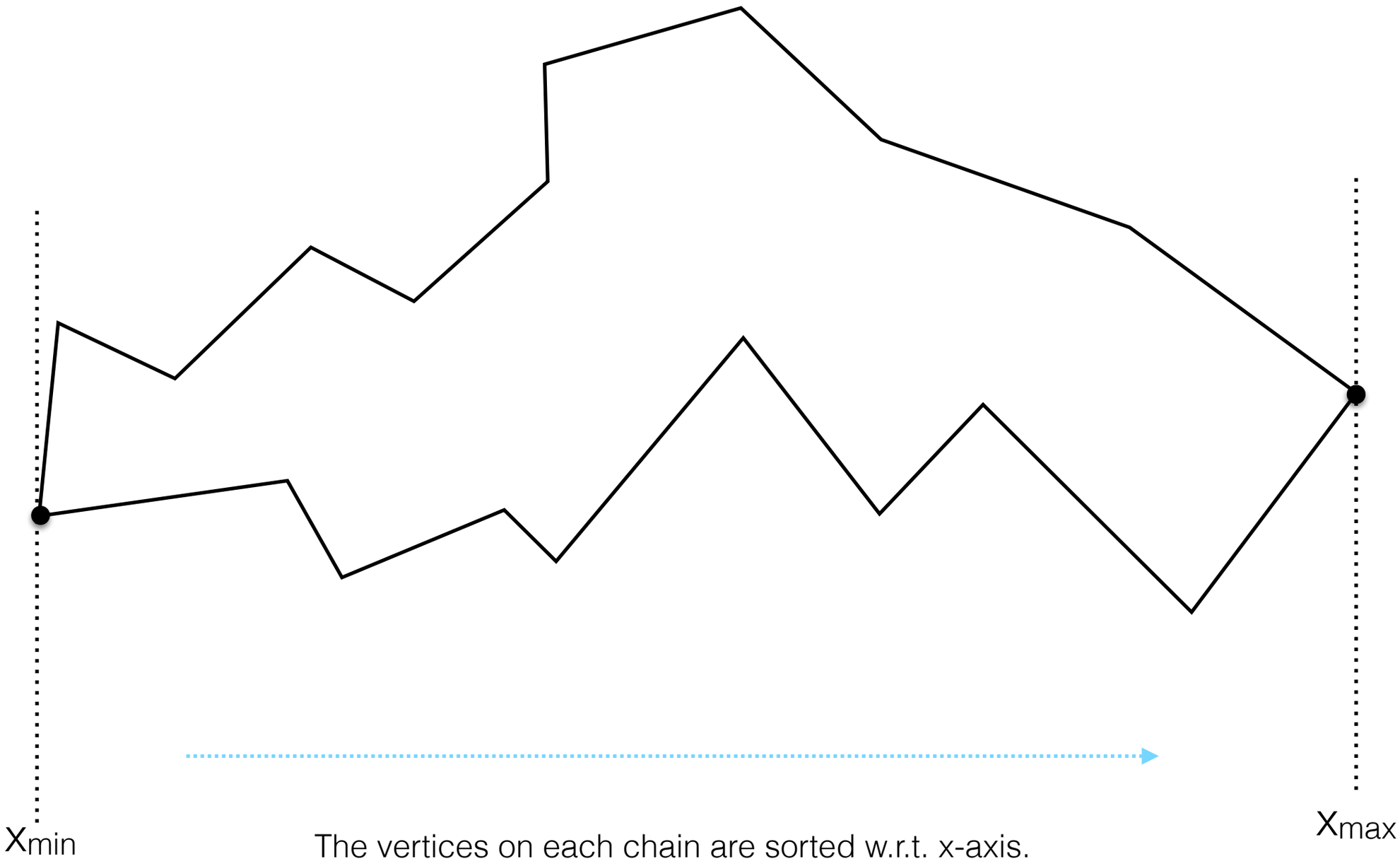
Monotone polygons

A polygon is x-monotone if its boundary can be split into two x-monotone chains.

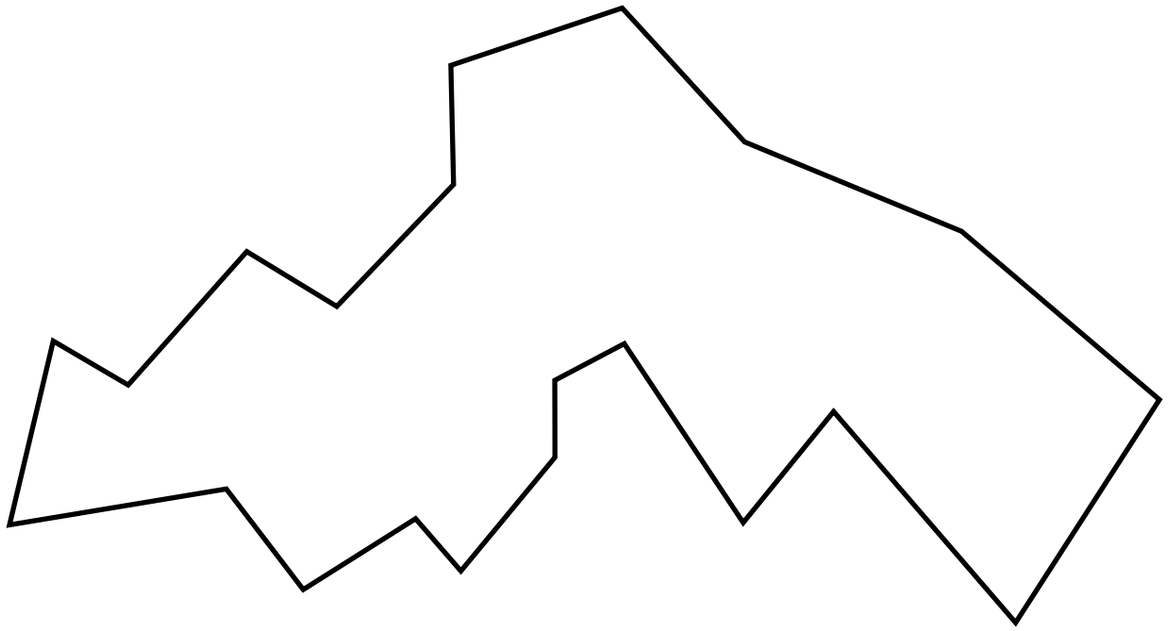


Monotone polygons

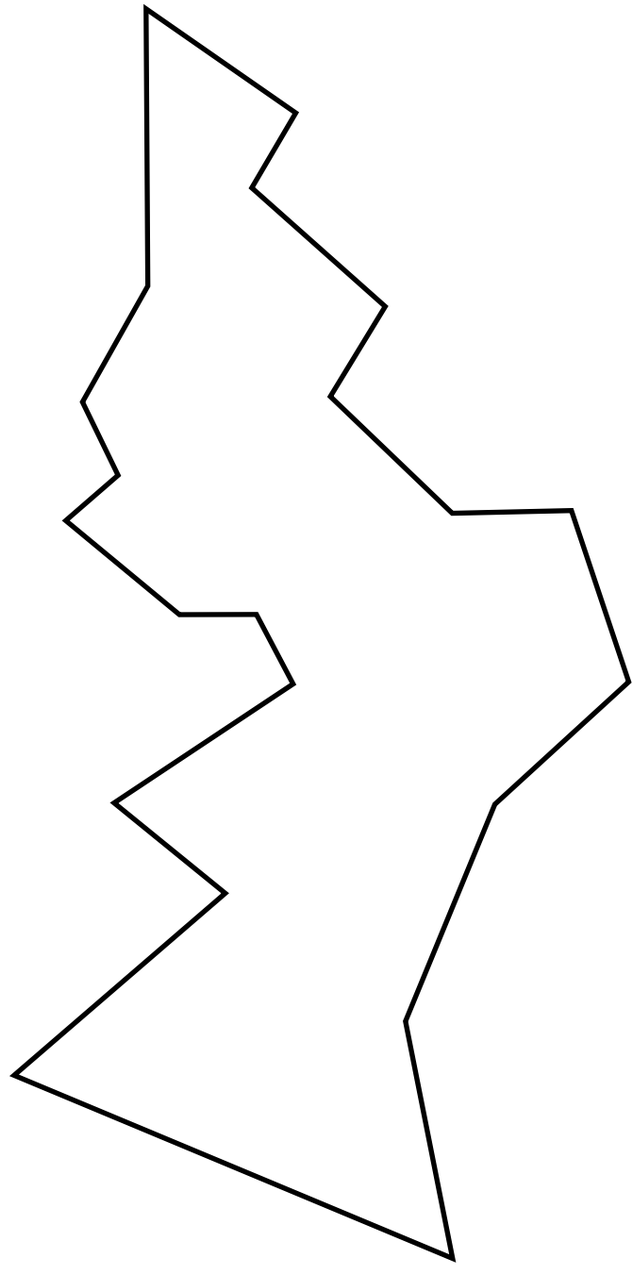
A polygon is x-monotone if its boundary can be split into two x-monotone chains.



Monotone polygons



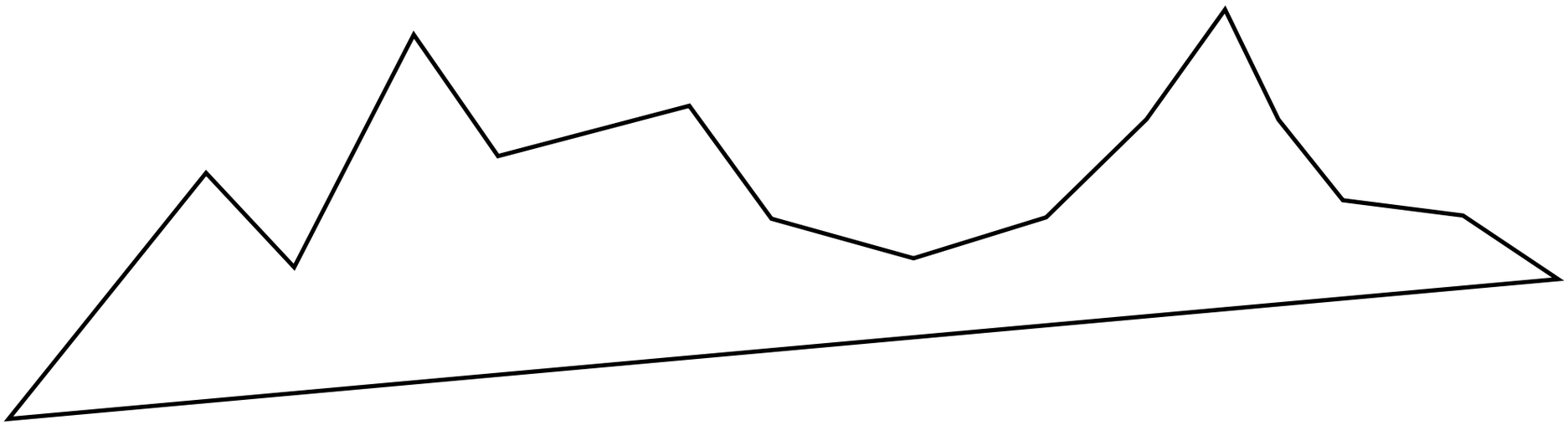
x-monotone



y-monotone

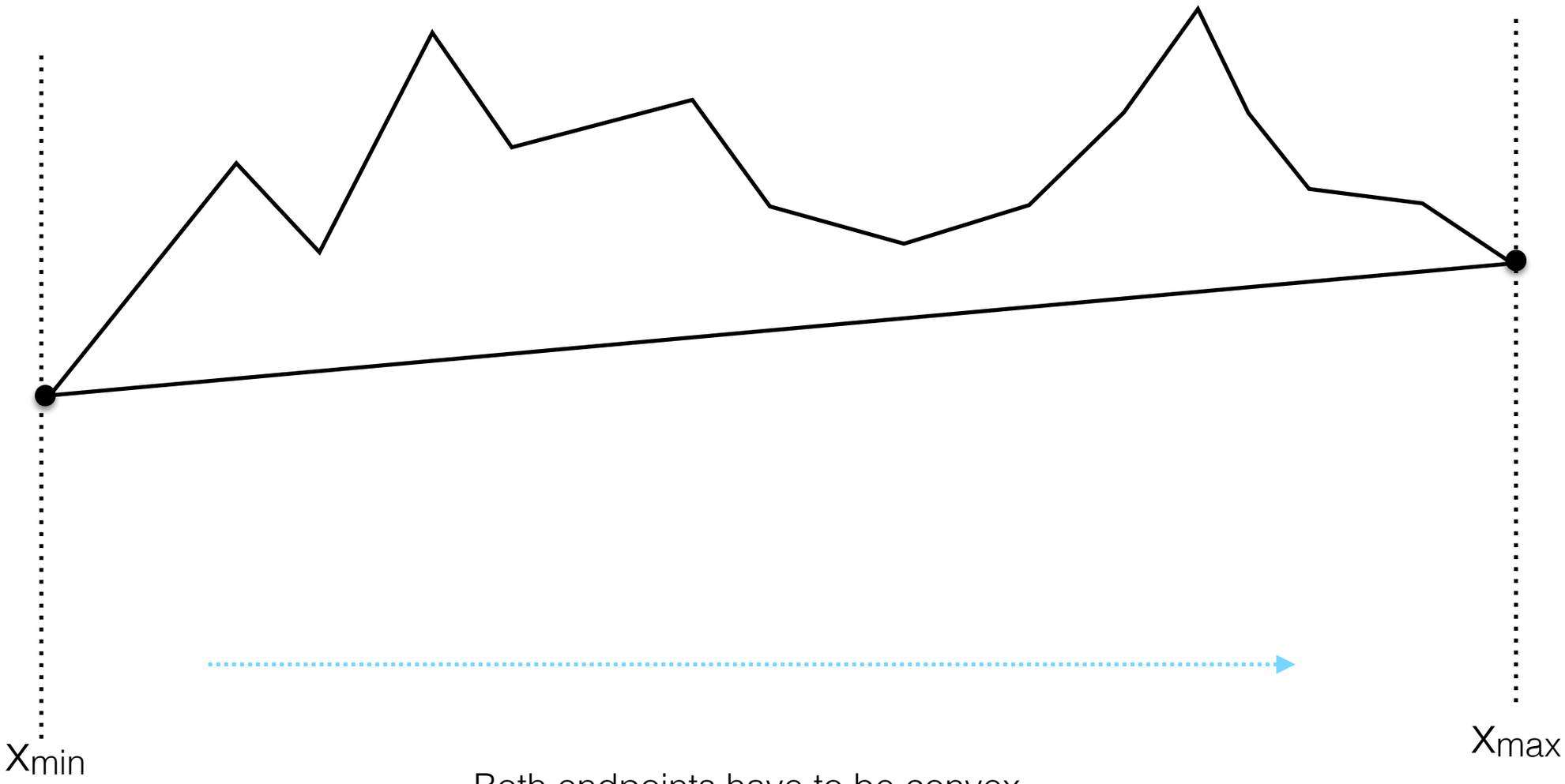
Monotone Mountains (unimonotone polygons)

A polygon is an **x-monotone mountain** if it is monotone and one of the two chains is a single segment.



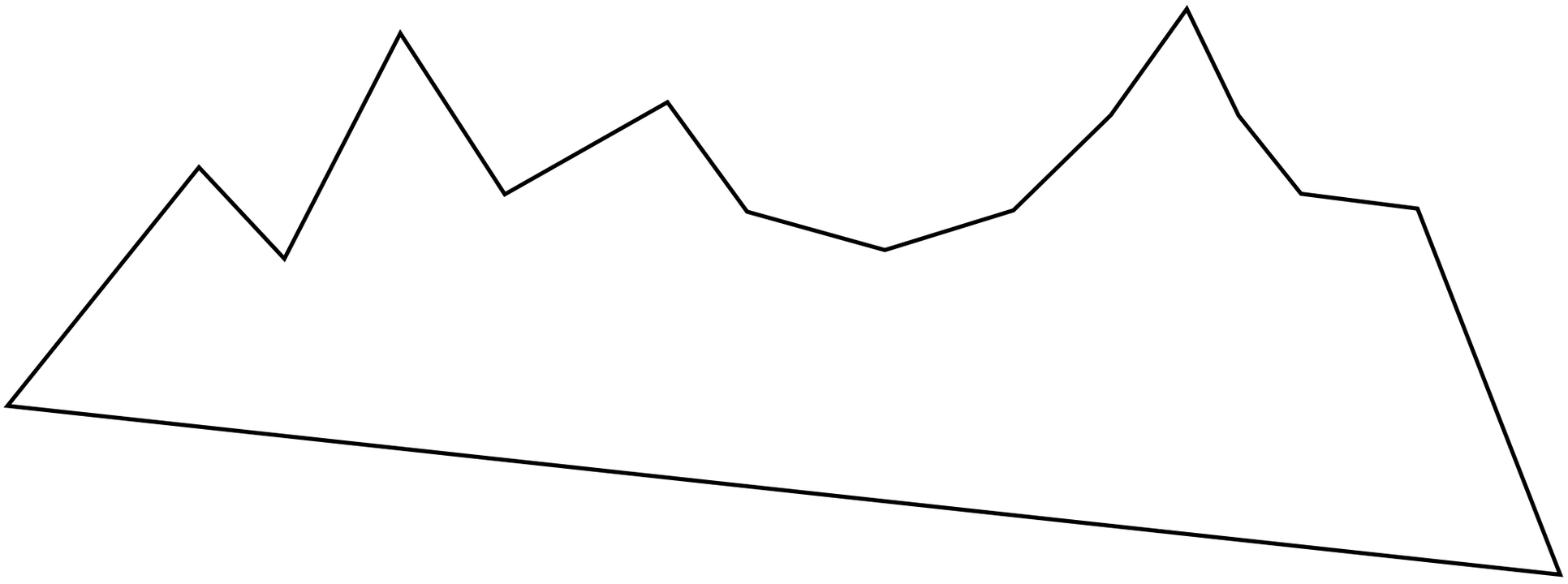
Monotone Mountains (unimonotone polygons)

A polygon is an **x-monotone mountain** if it is monotone and one of the two chains is a single segment.



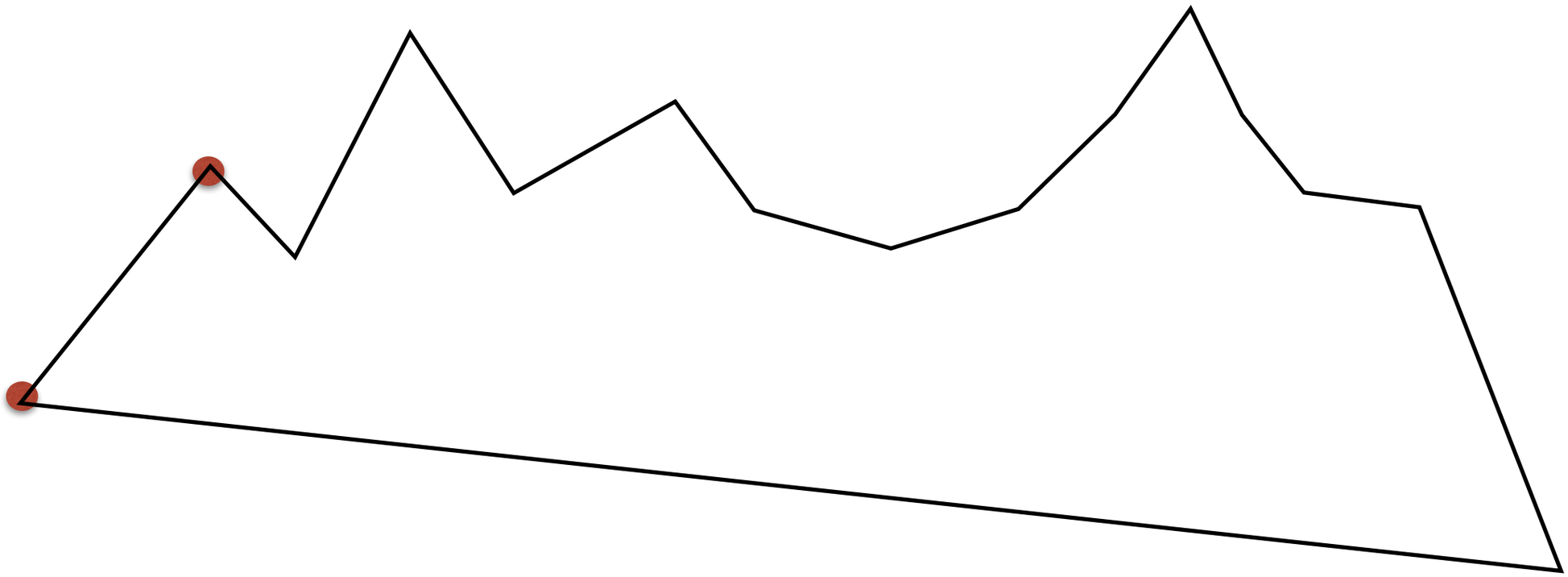
Both endpoints have to be convex.

Monotone mountains are easy to triangulate!

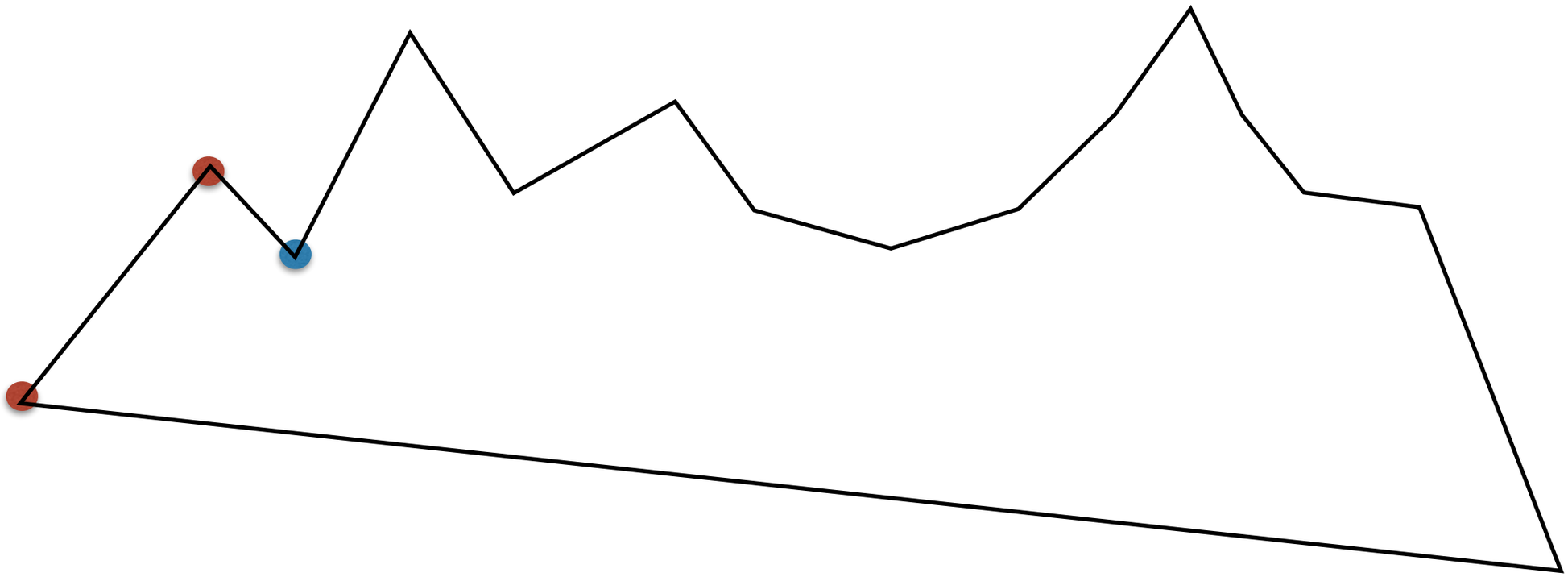


Class work: Let’s come up with an algorithm (and analyze it).

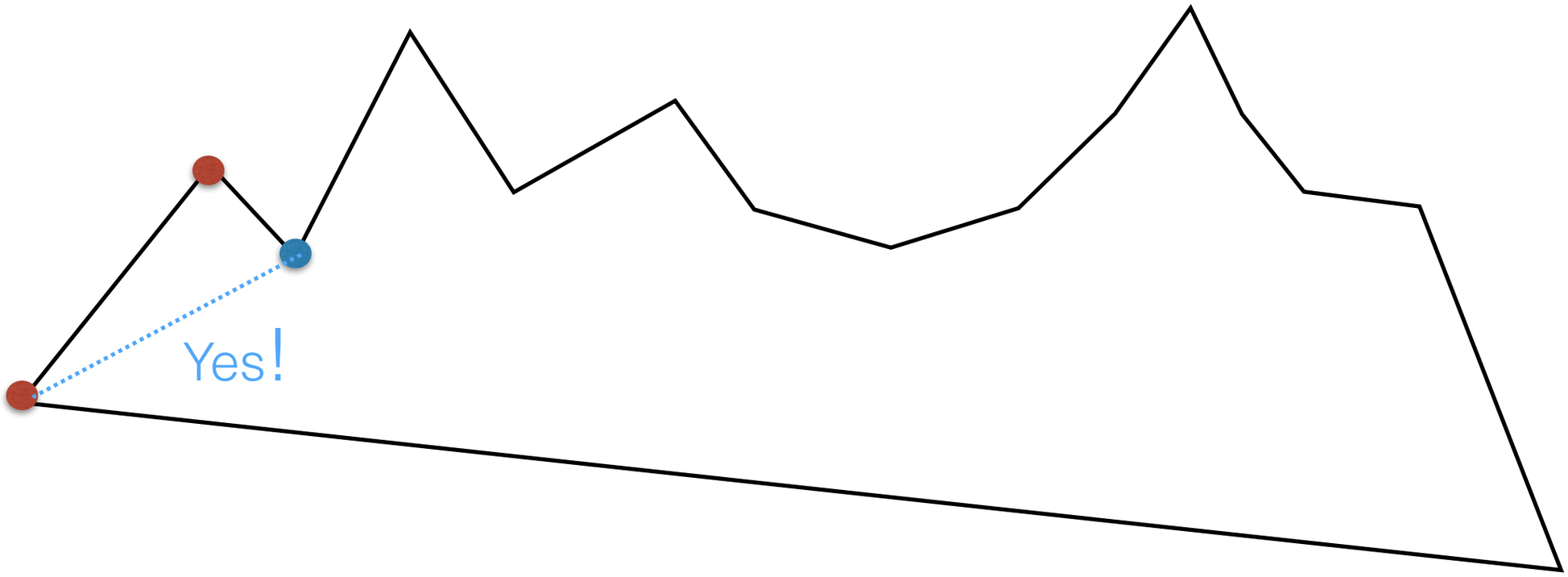
Monotone mountains are easy to triangulate!



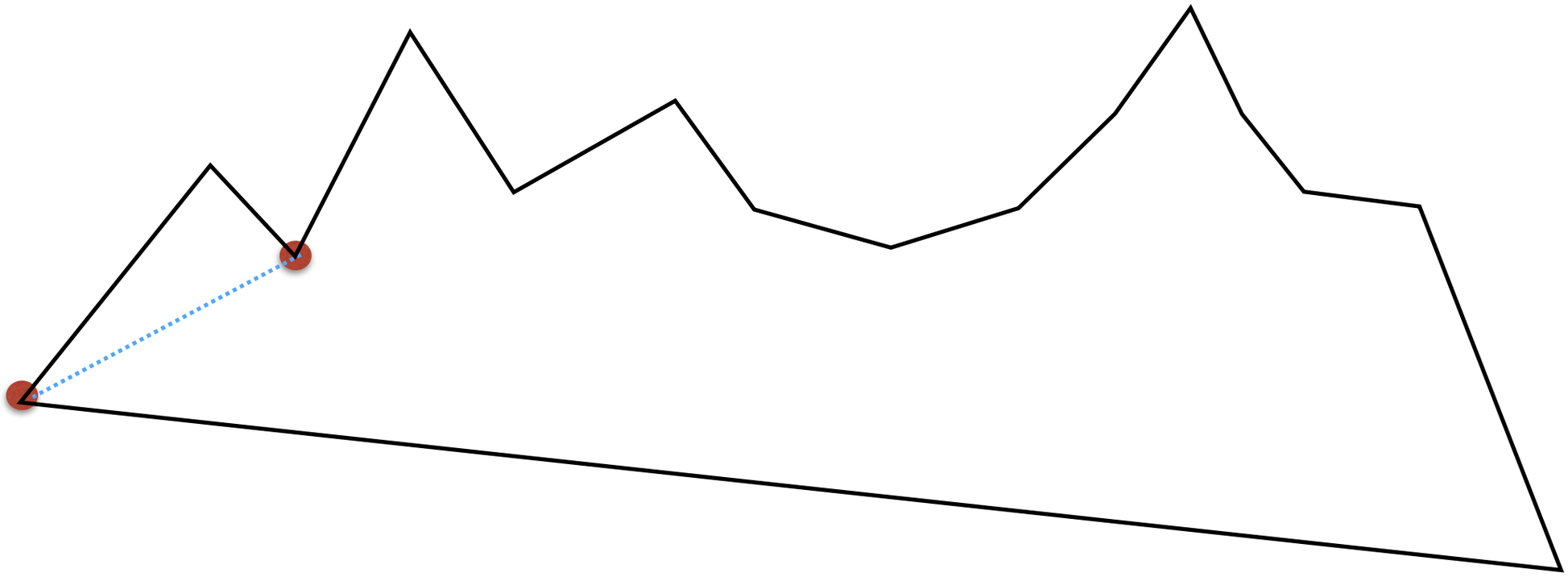
Monotone mountains are easy to triangulate!



Monotone mountains are easy to triangulate!

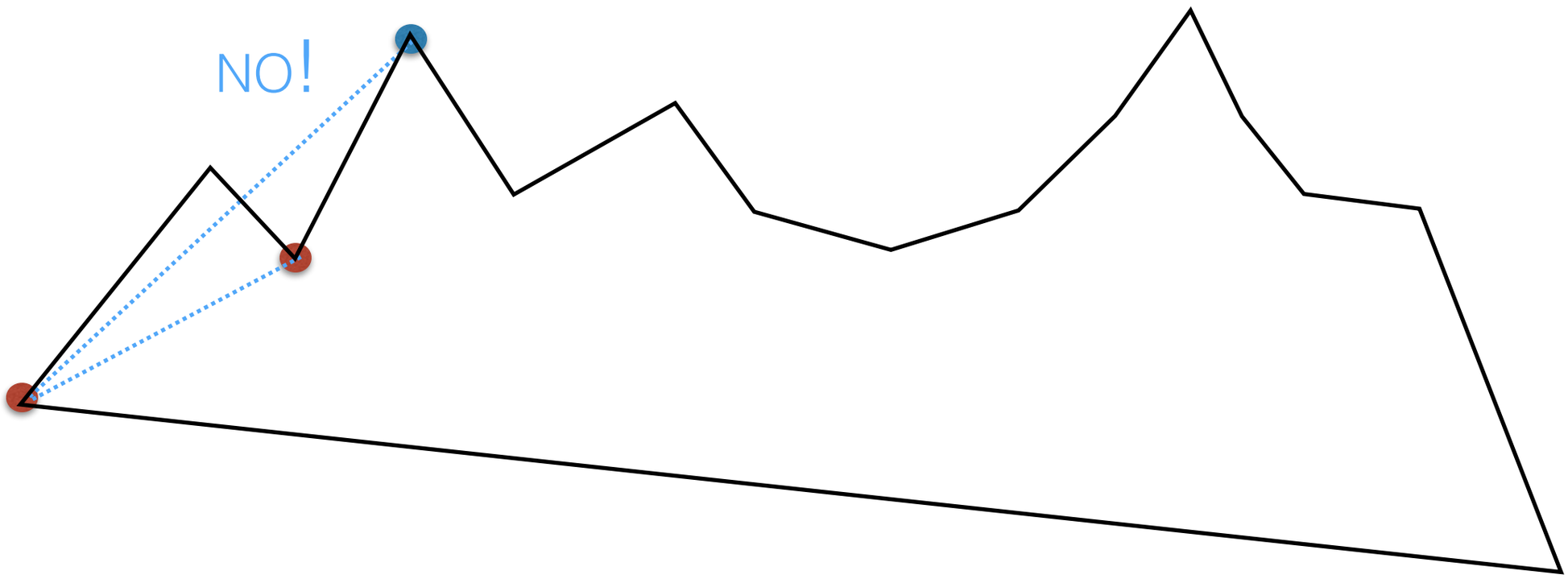


Monotone mountains are easy to triangulate!



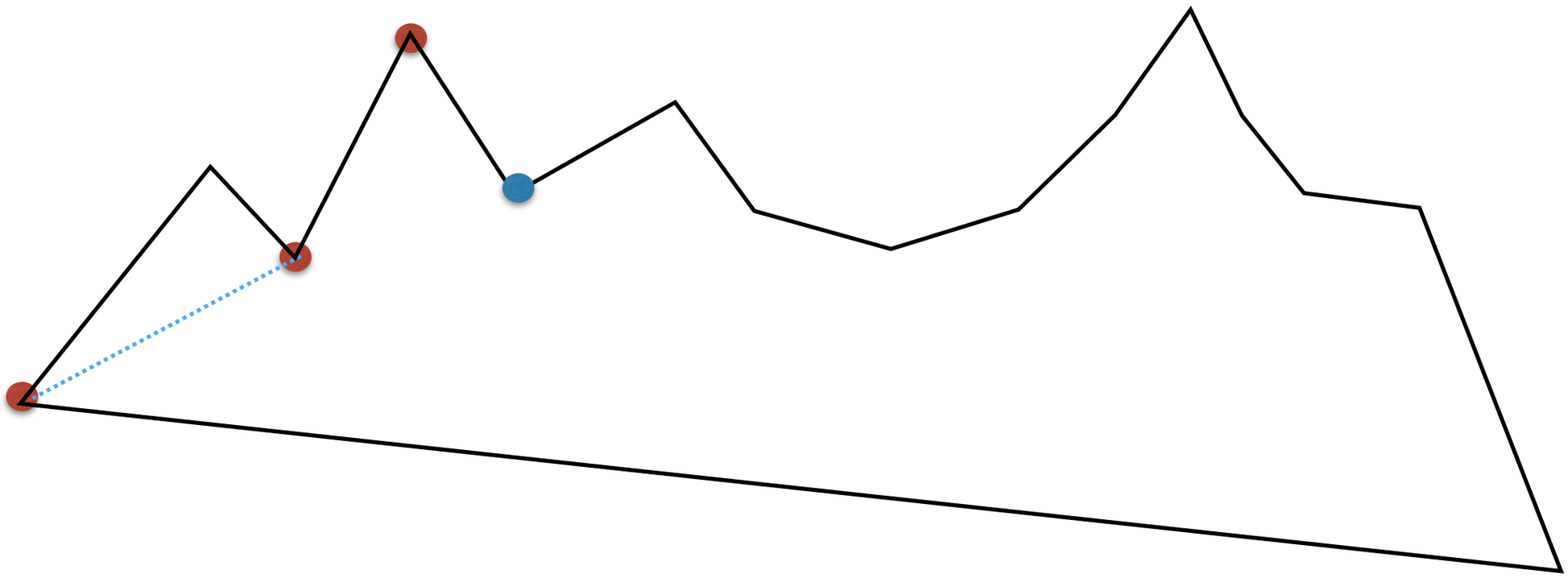
Monotone mountains are easy to triangulate!

Monotone mountains are easy to triangulate!

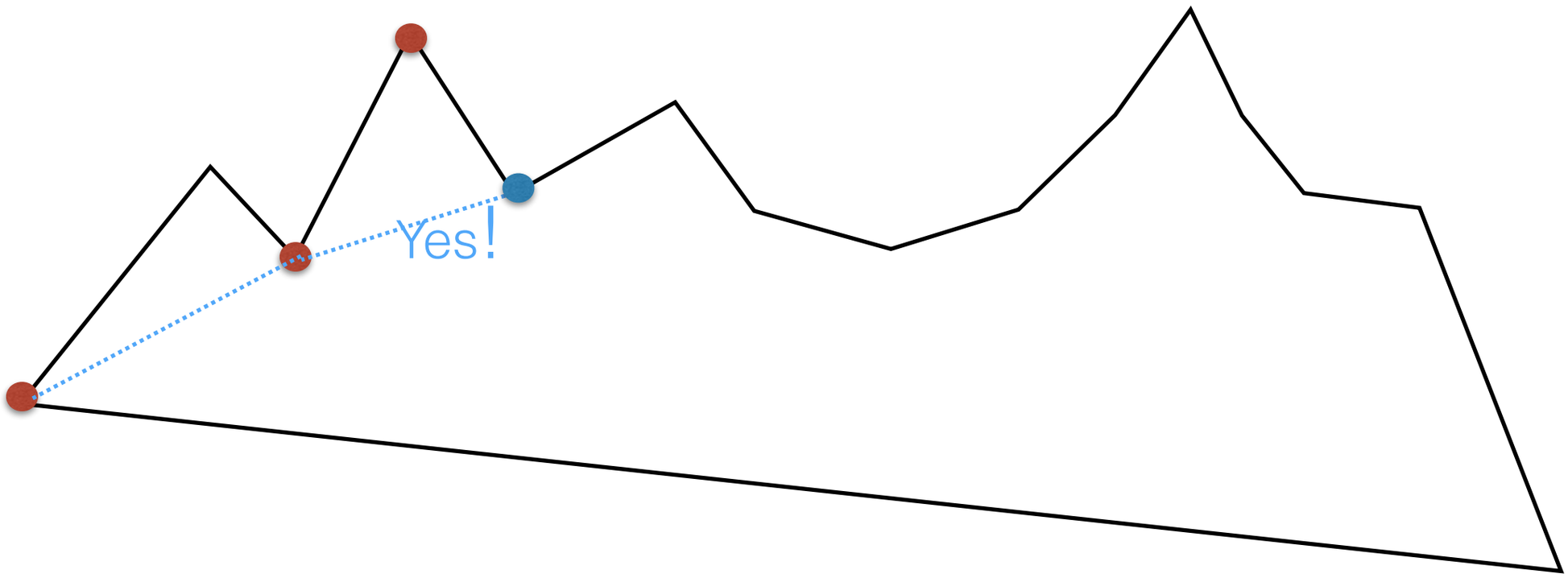


Monotone mountains are easy to triangulate!

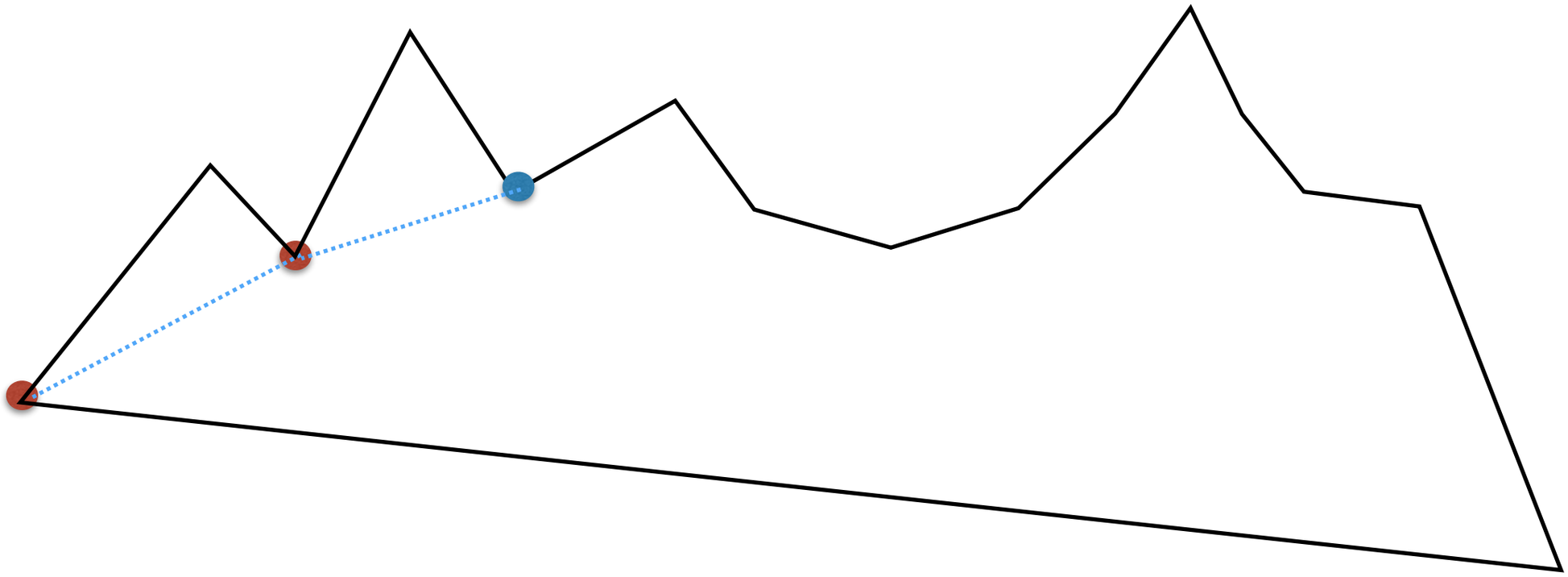
Monotone mountains are easy to triangulate!



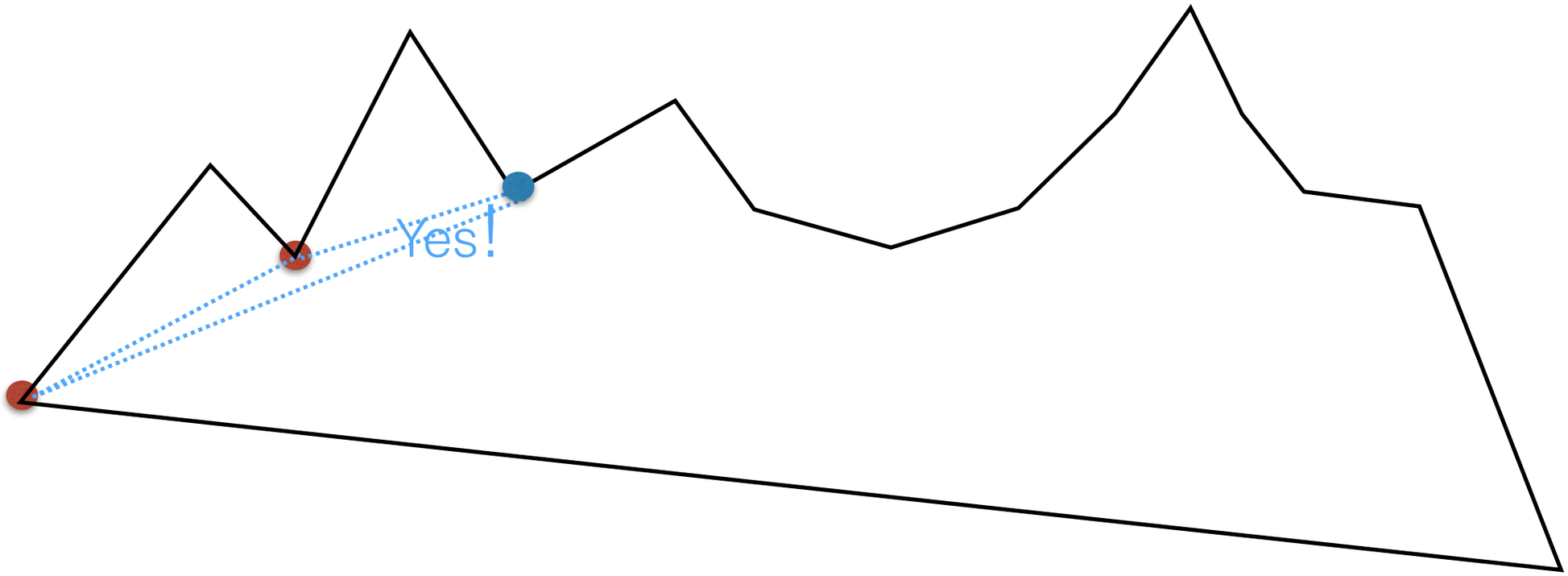
Monotone mountains are easy to triangulate!



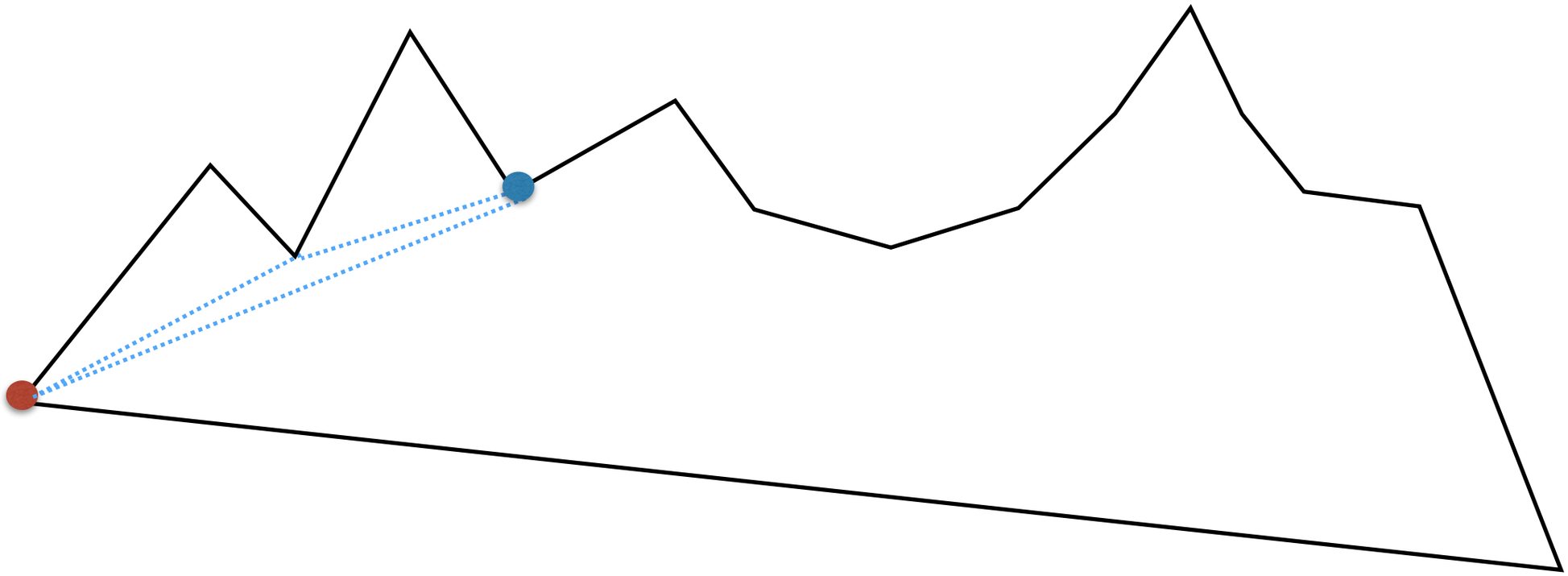
Monotone mountains are easy to triangulate!



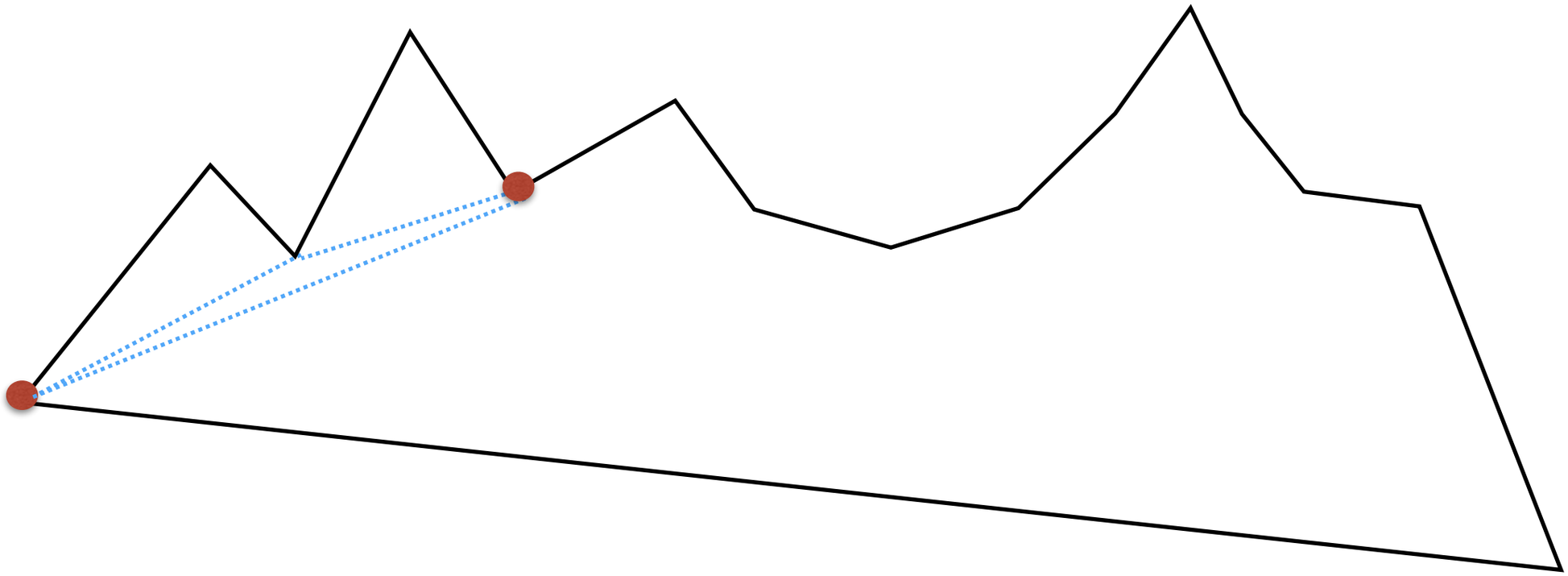
Monotone mountains are easy to triangulate!



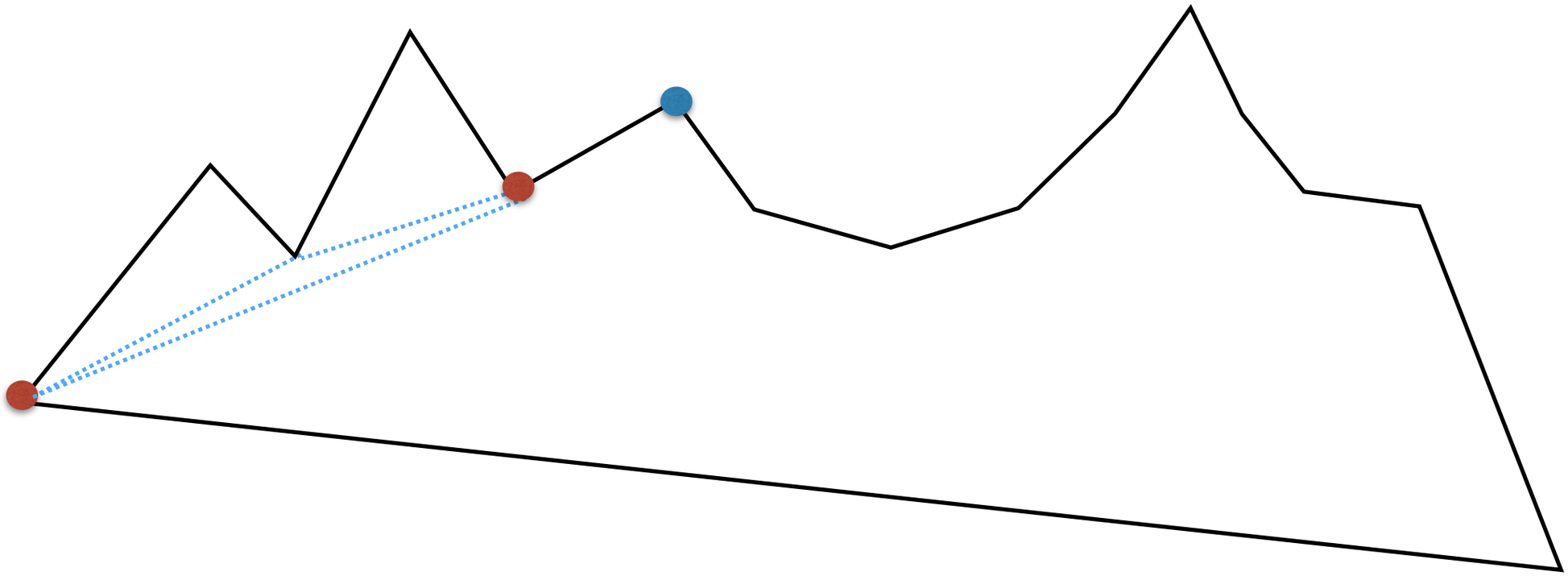
Monotone mountains are easy to triangulate!



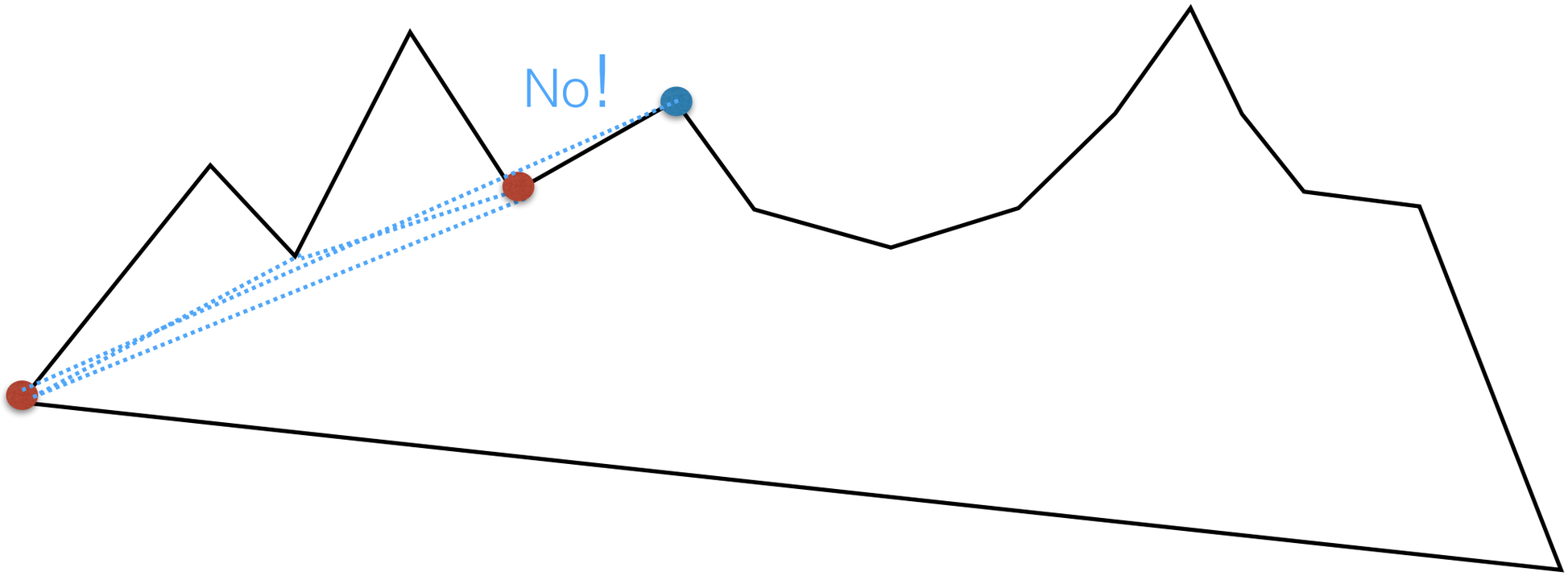
Monotone mountains are easy to triangulate!



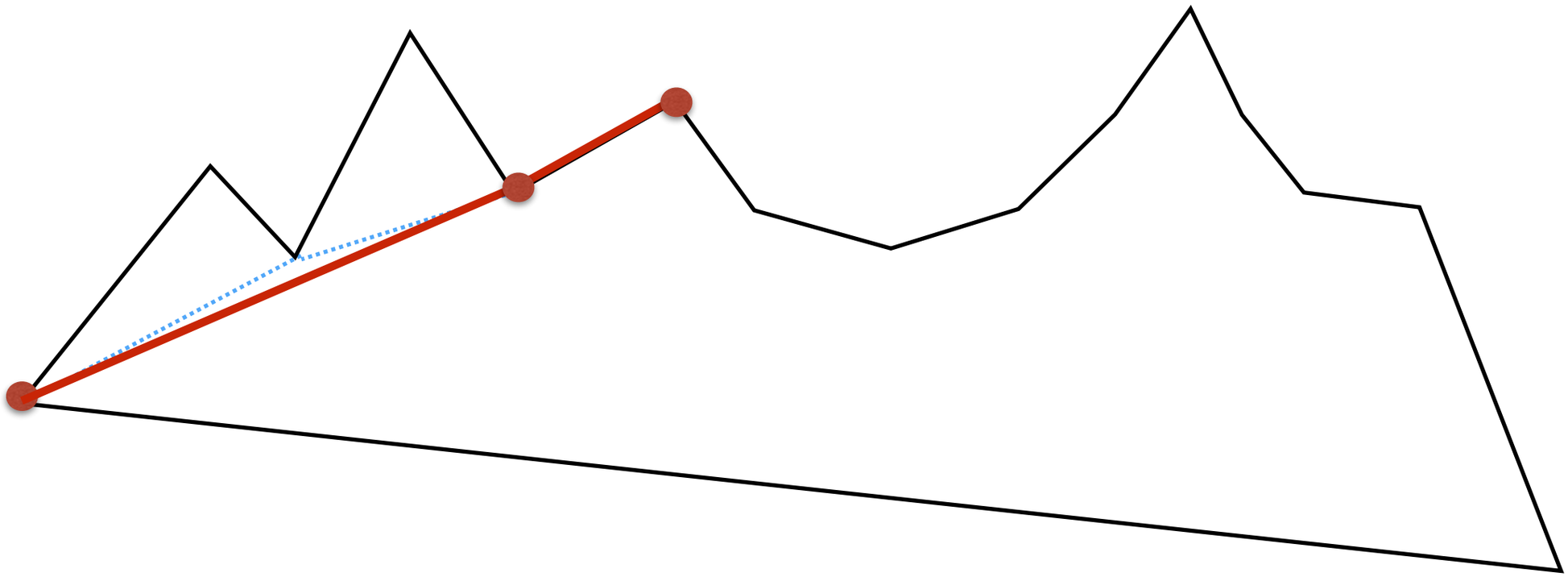
Monotone mountains are easy to triangulate!



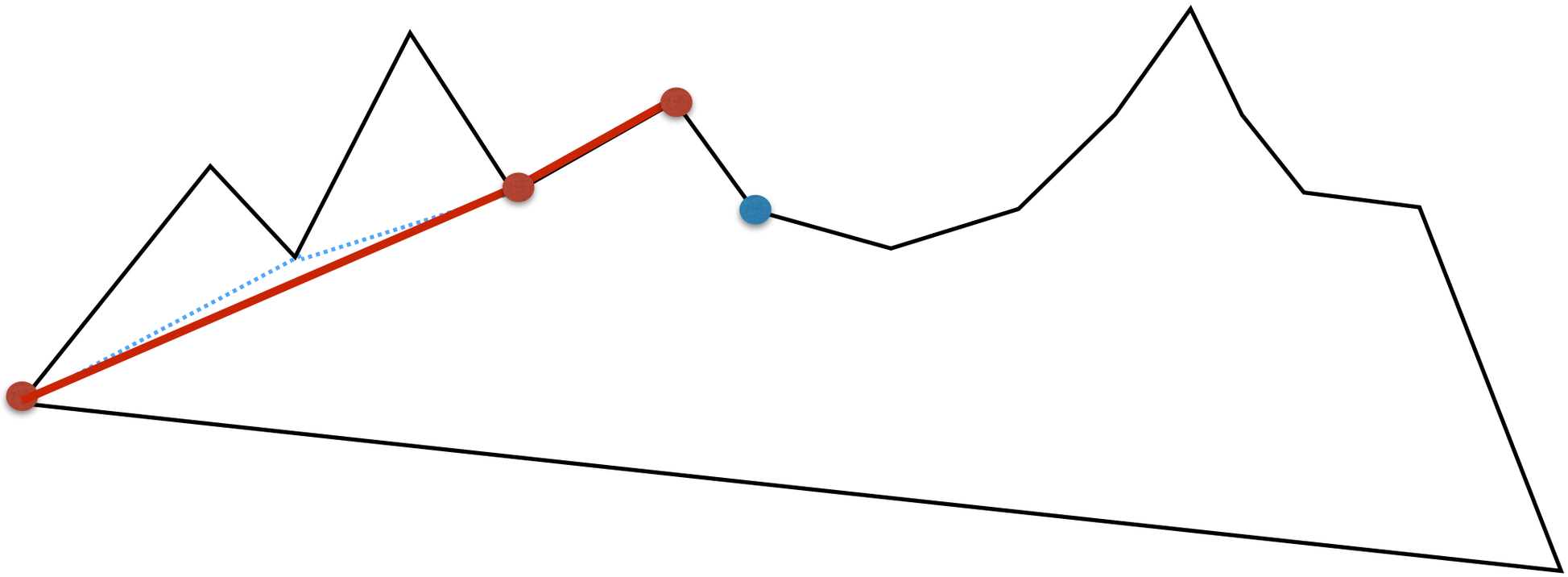
Monotone mountains are easy to triangulate!



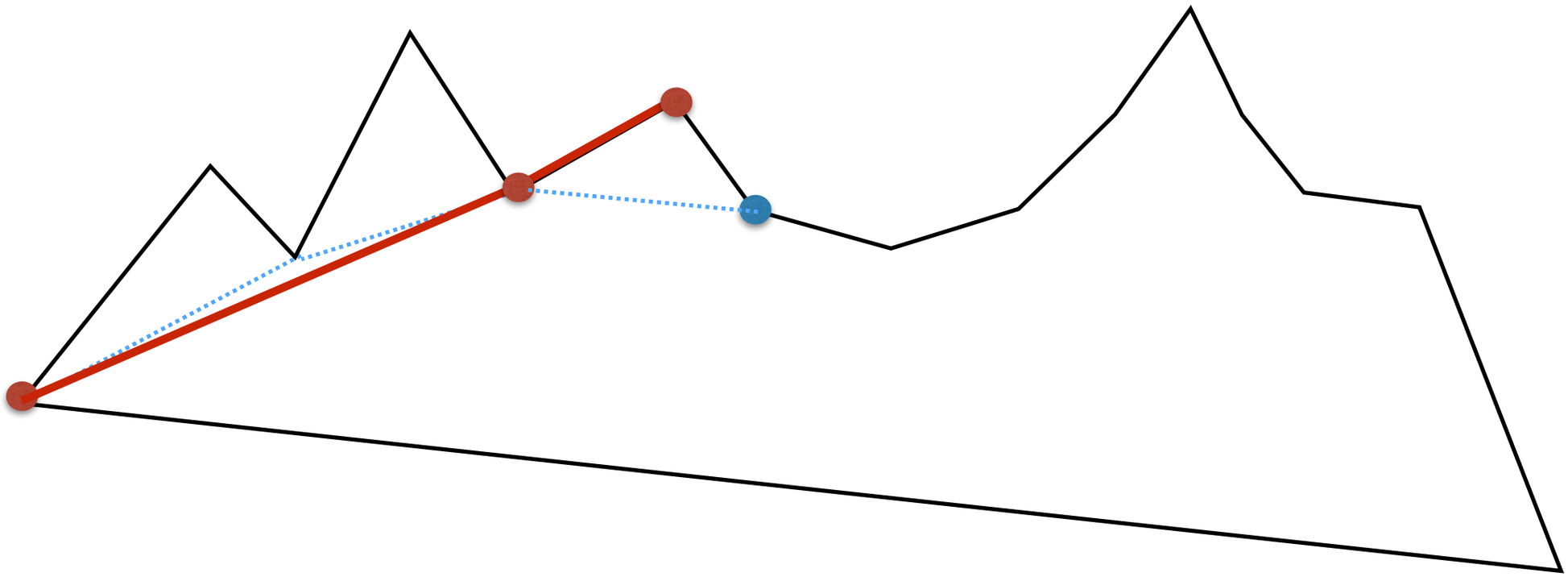
Monotone mountains are easy to triangulate!



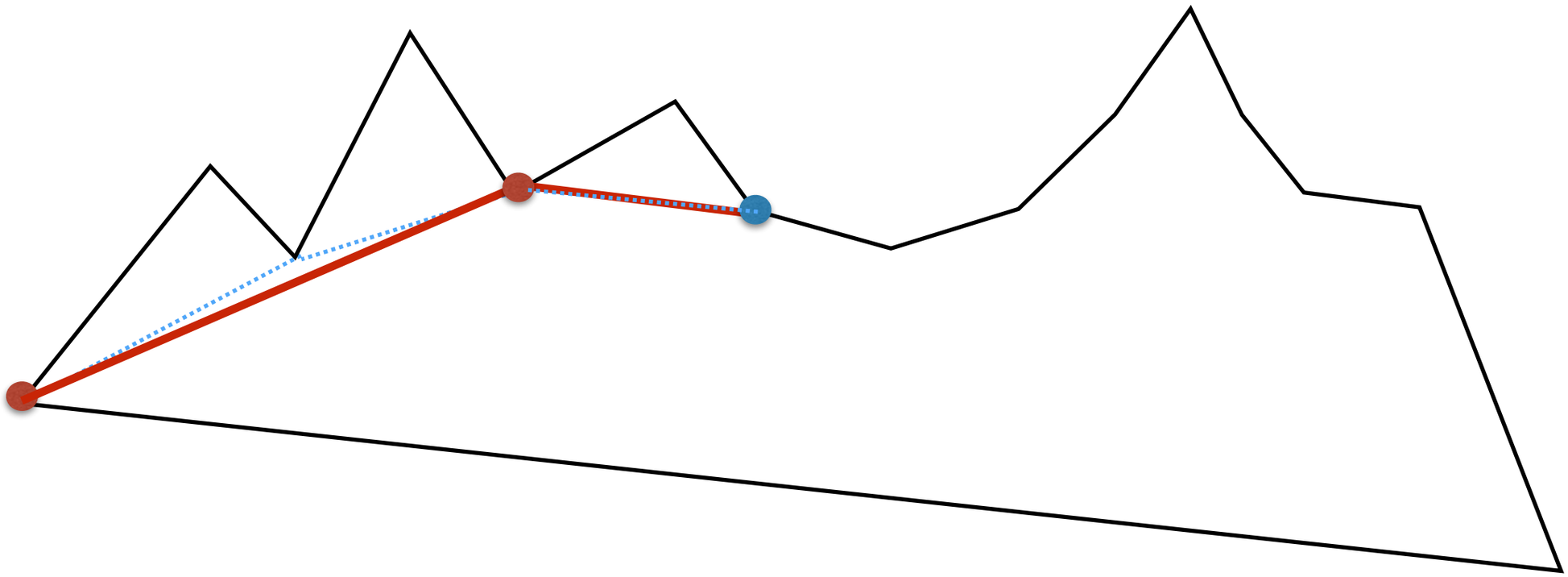
Monotone mountains are easy to triangulate!



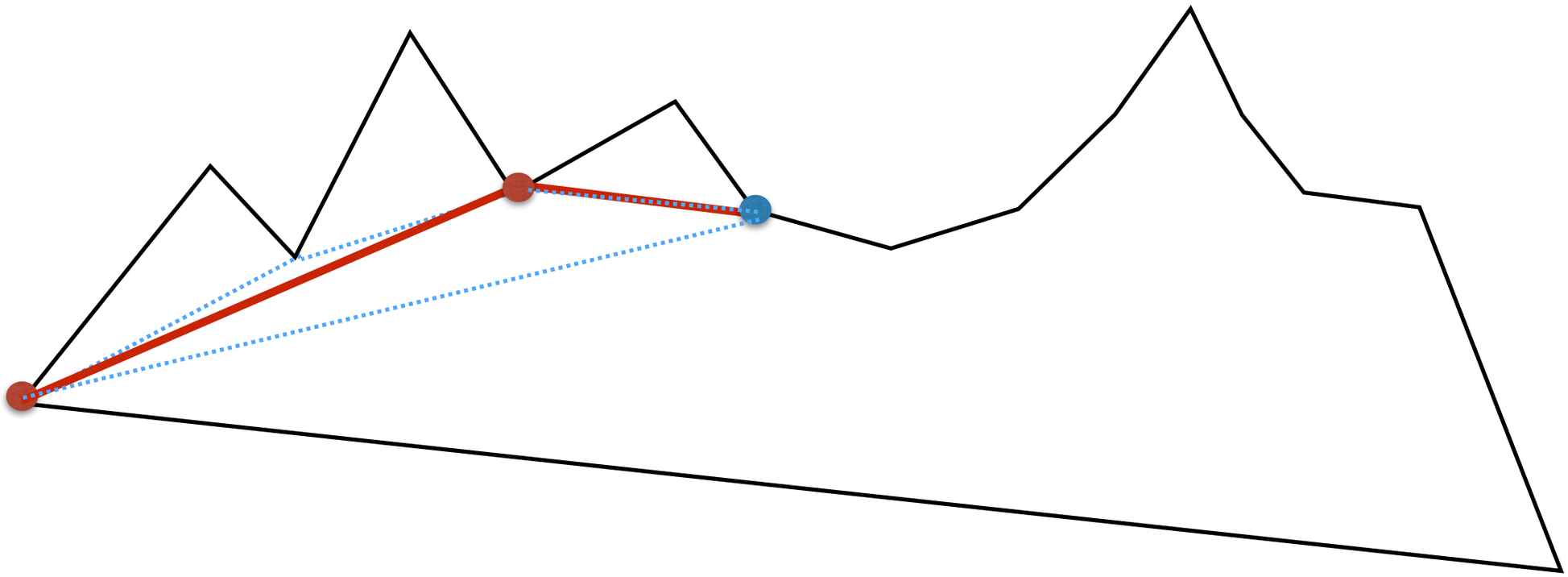
Monotone mountains are easy to triangulate!



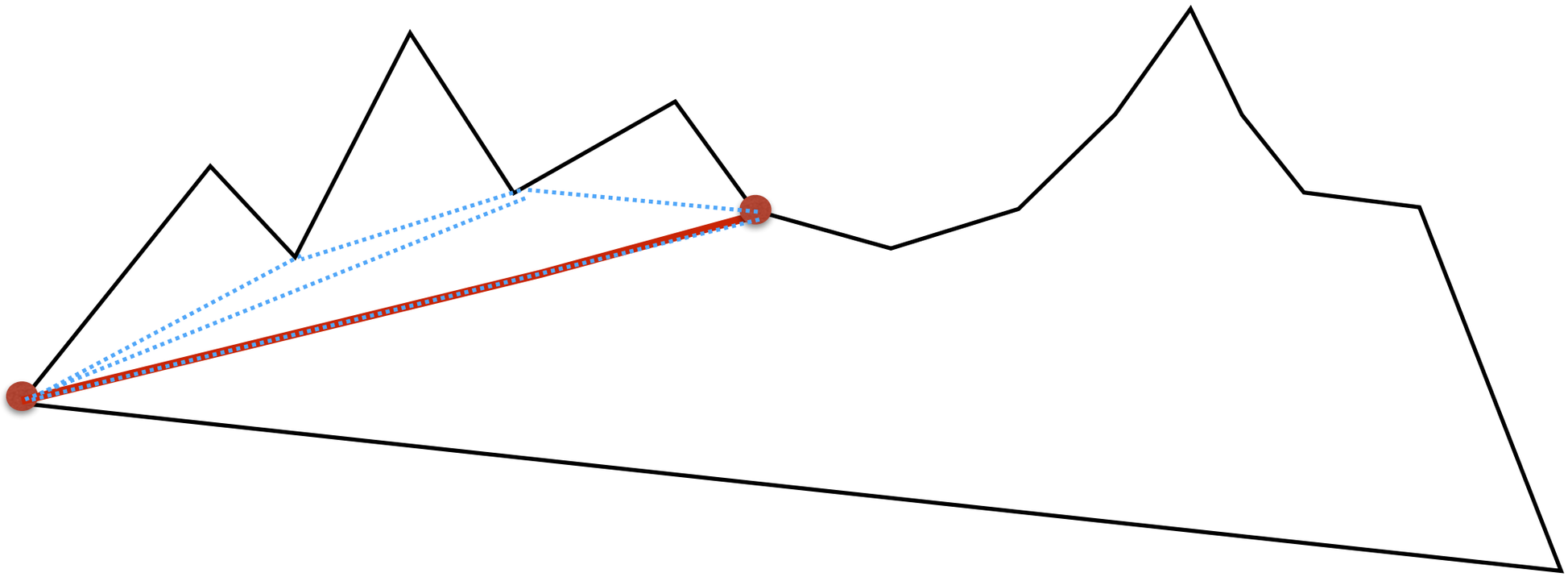
Monotone mountains are easy to triangulate!



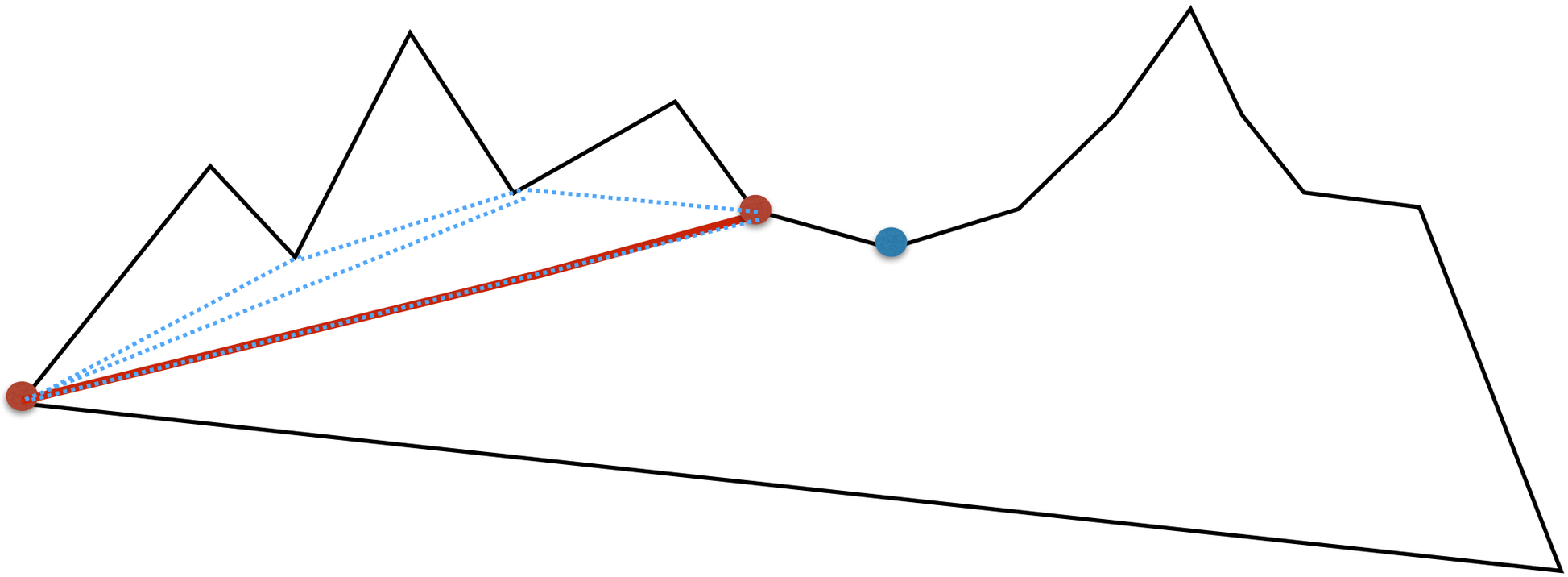
Monotone mountains are easy to triangulate!



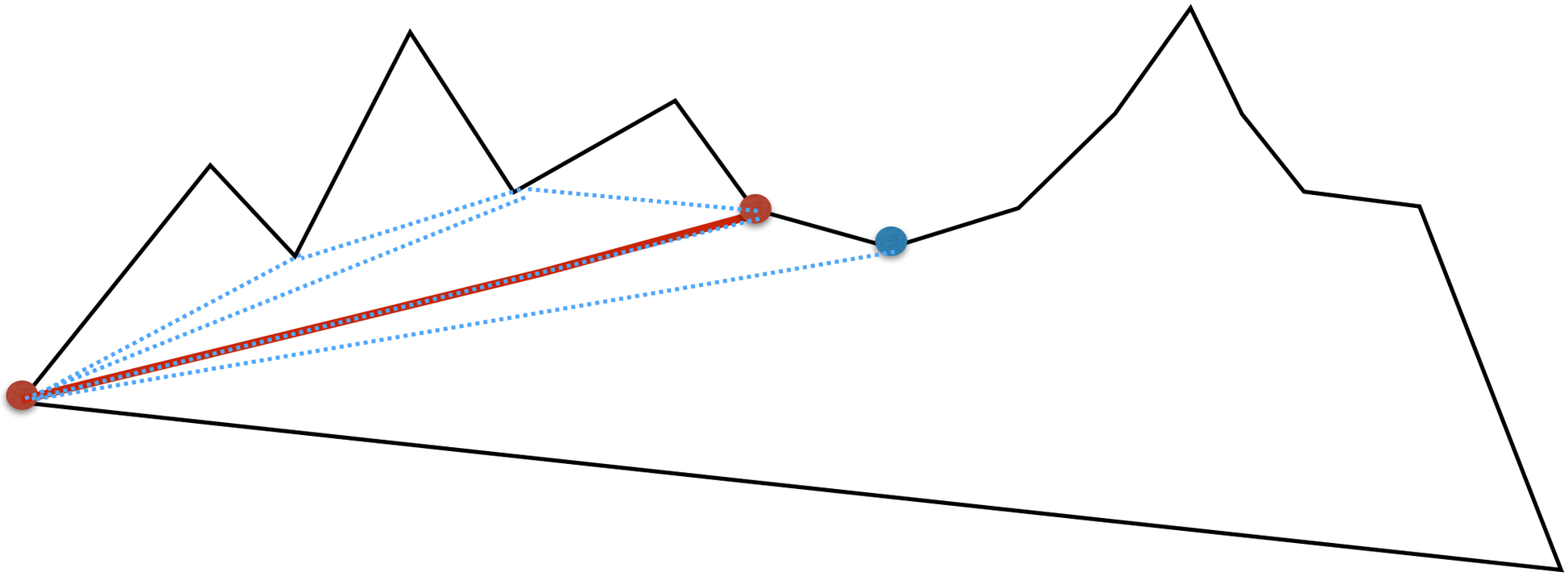
Monotone mountains are easy to triangulate!



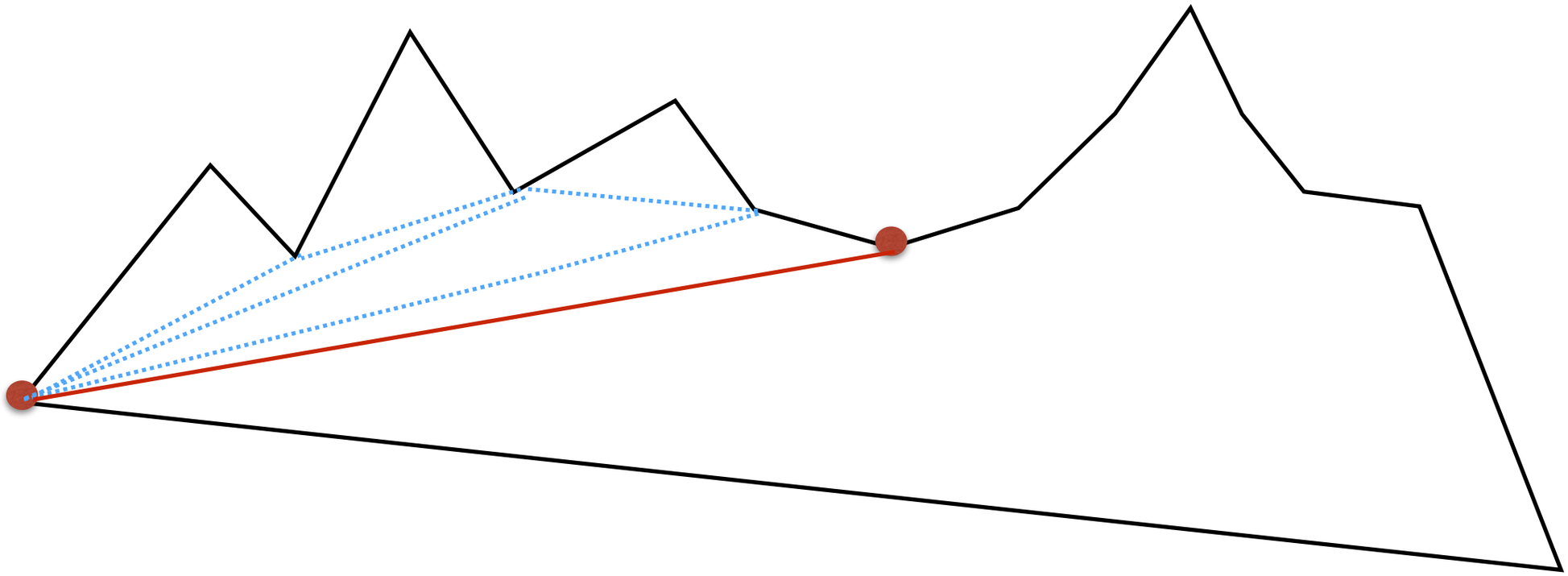
Monotone mountains are easy to triangulate!



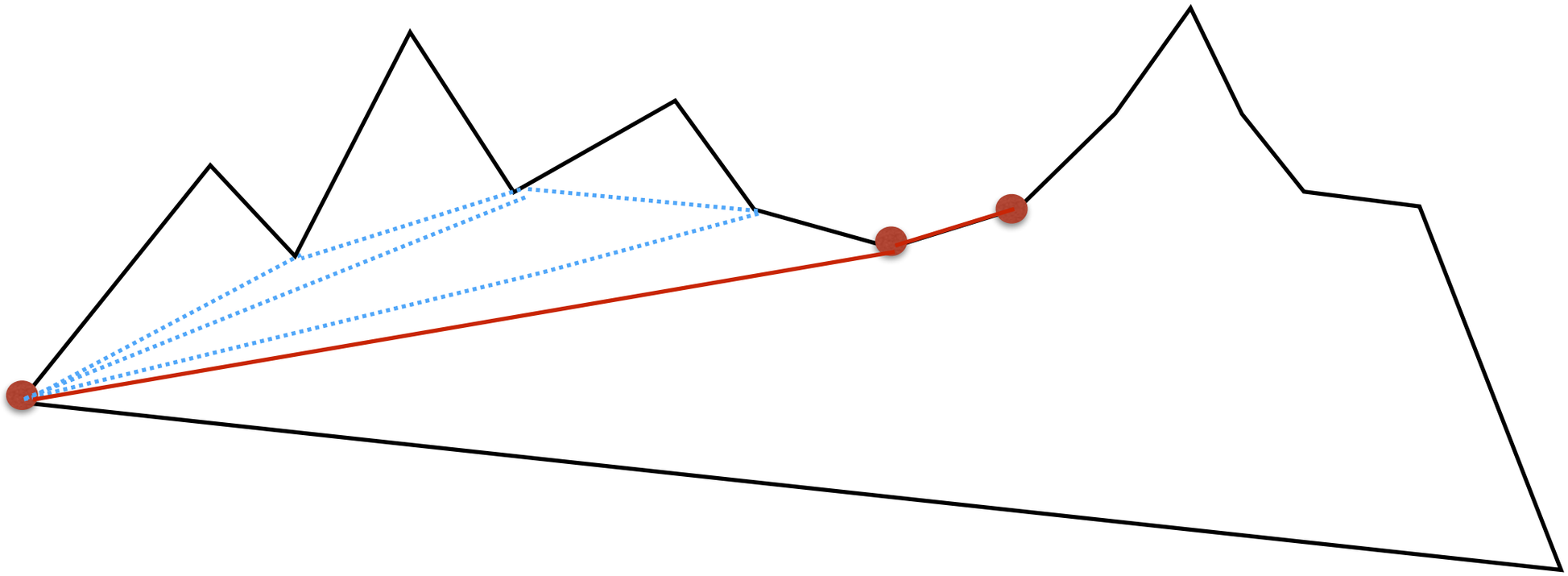
Monotone mountains are easy to triangulate!



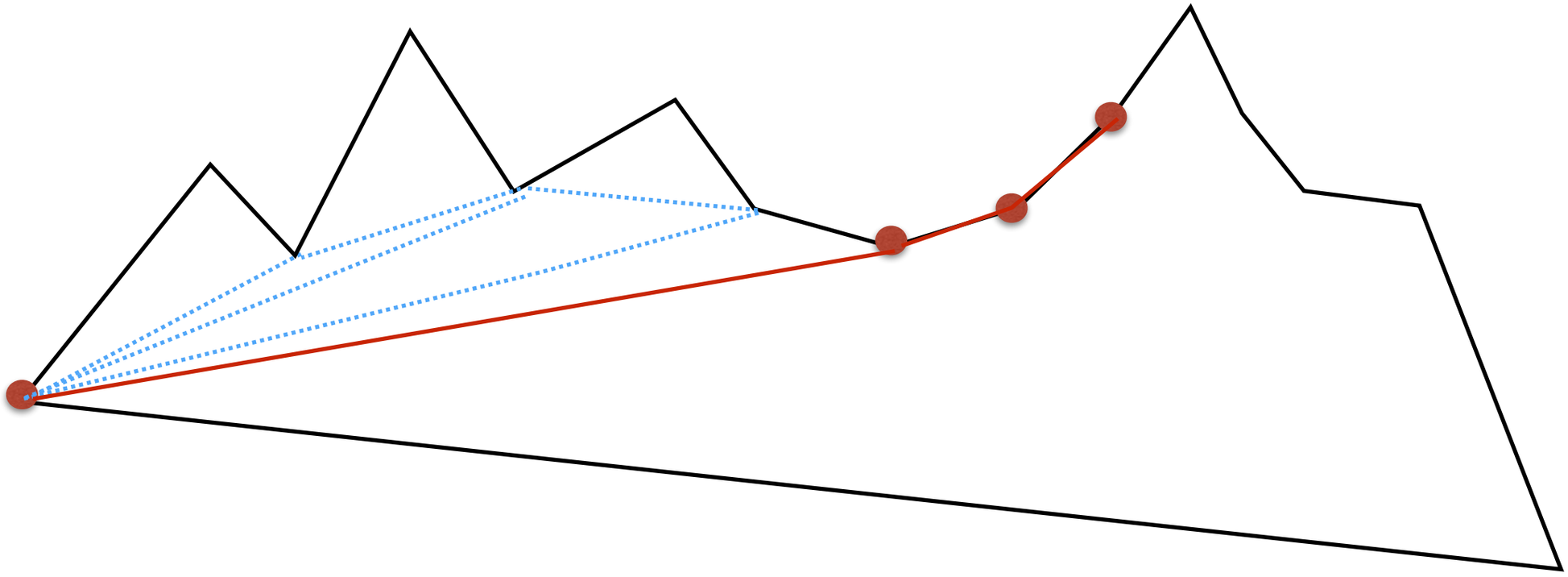
Monotone mountains are easy to triangulate!



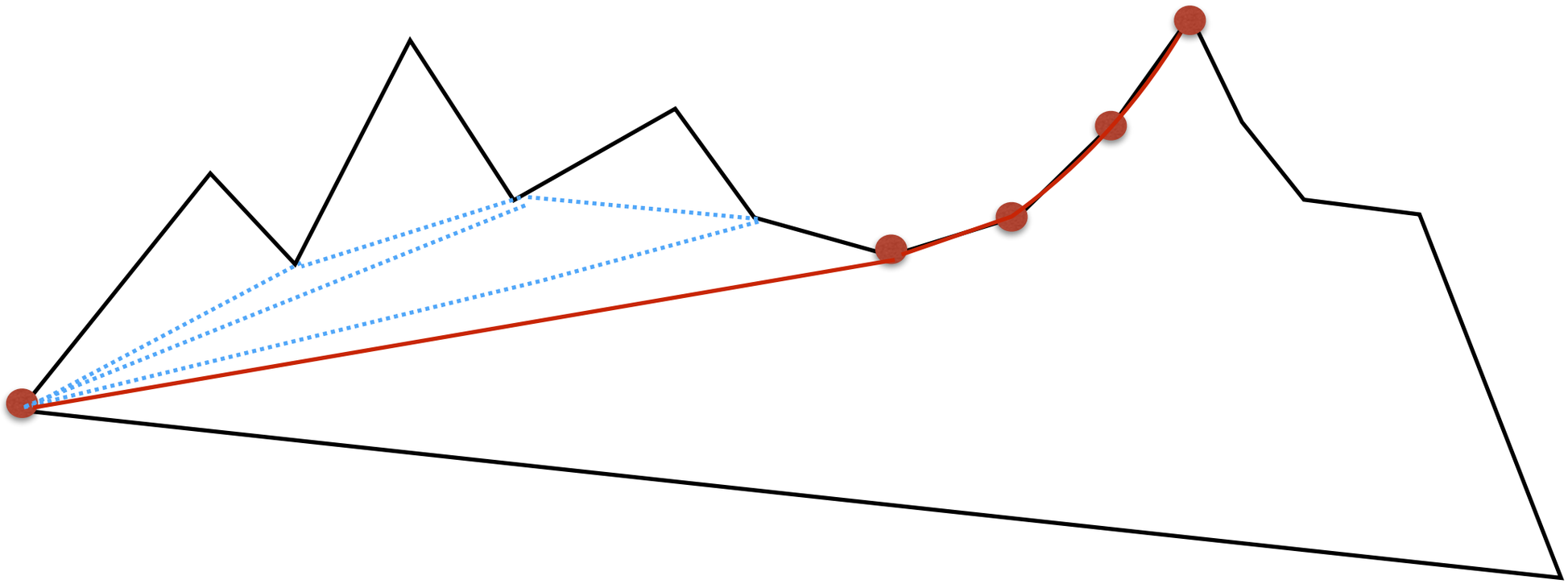
Monotone mountains are easy to triangulate!



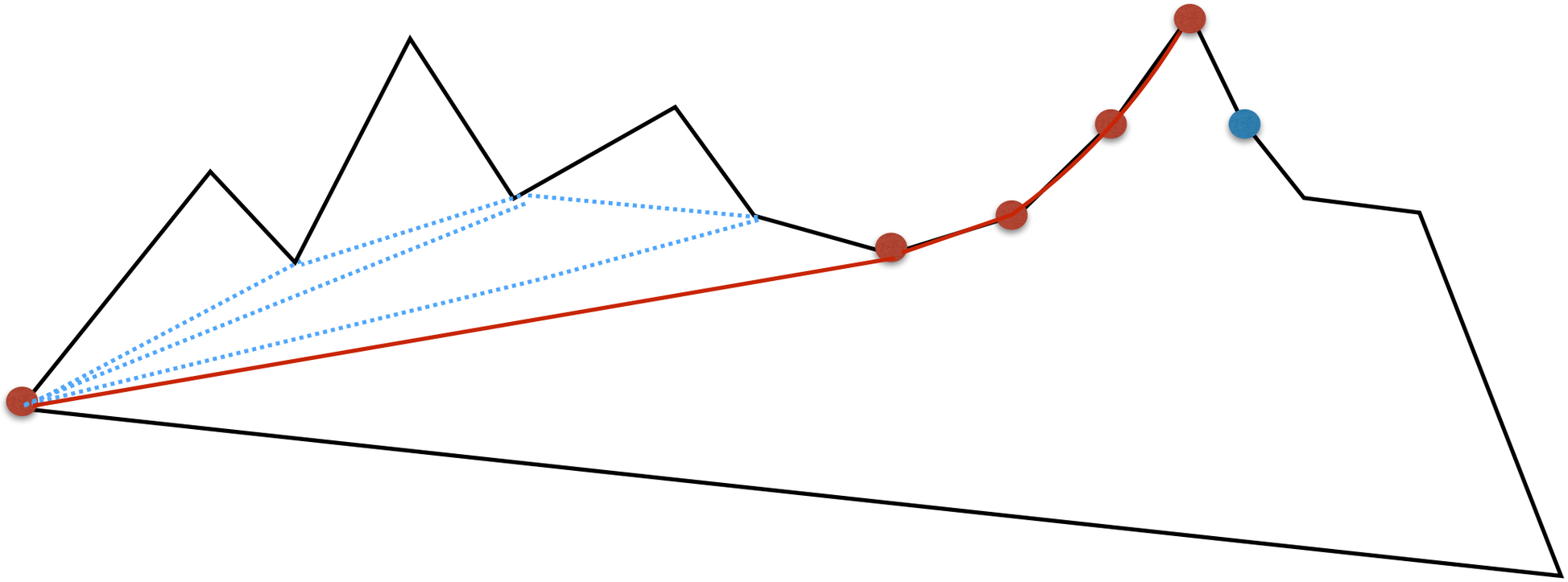
Monotone mountains are easy to triangulate!



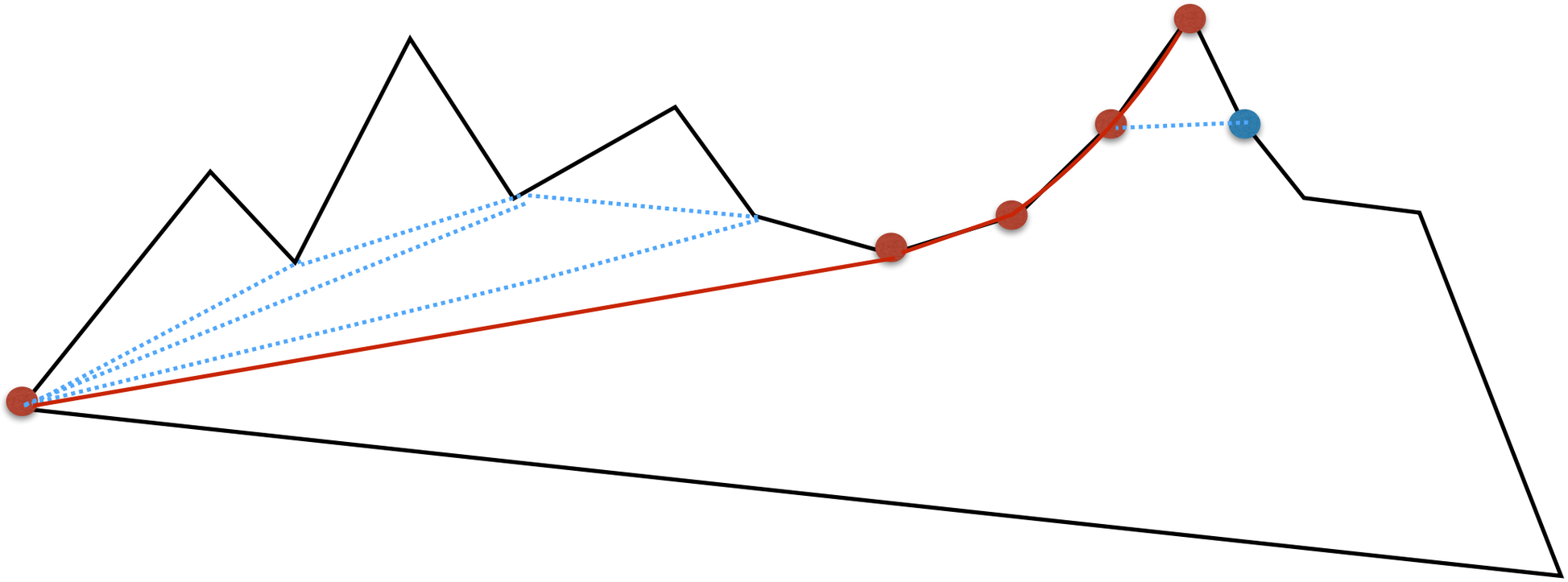
Monotone mountains are easy to triangulate!



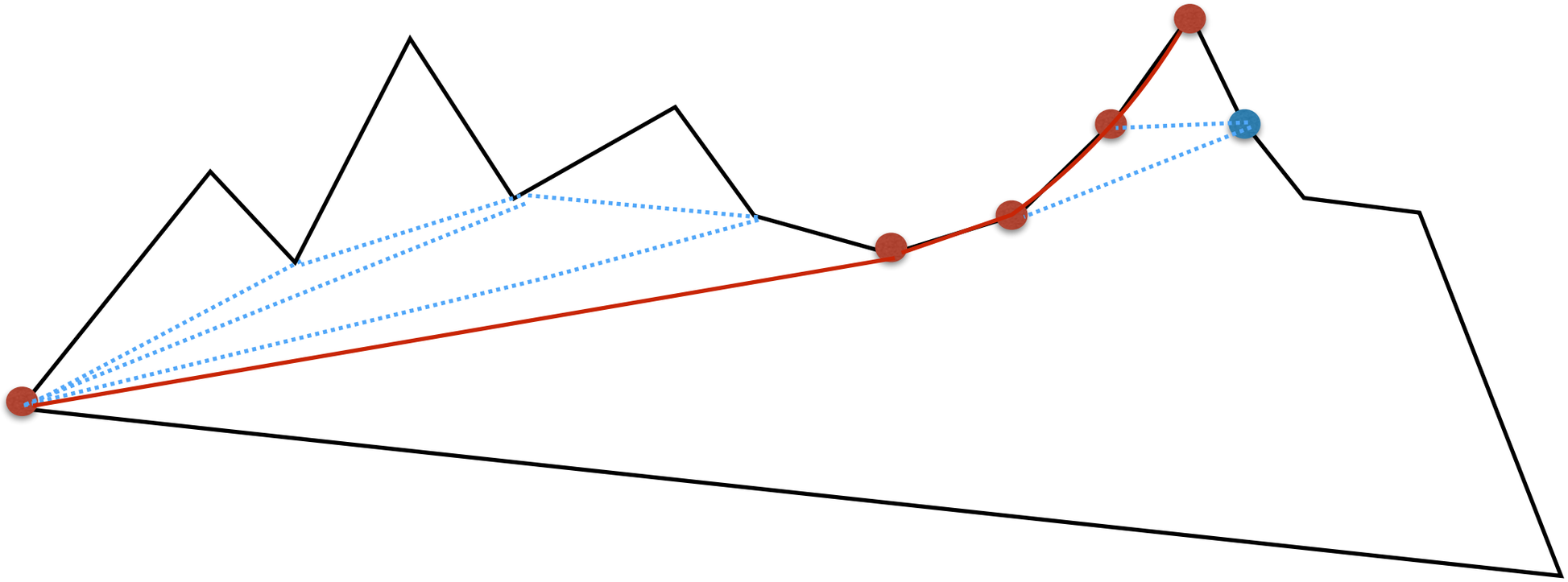
Monotone mountains are easy to triangulate!



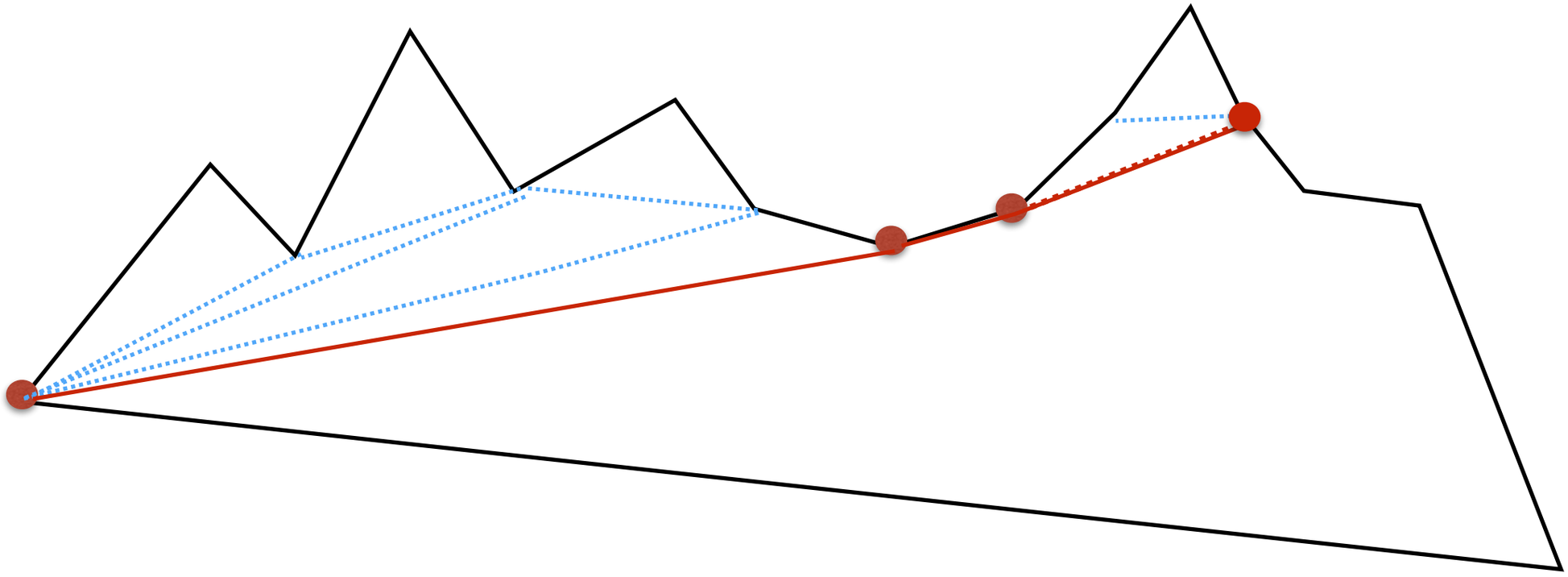
Monotone mountains are easy to triangulate!



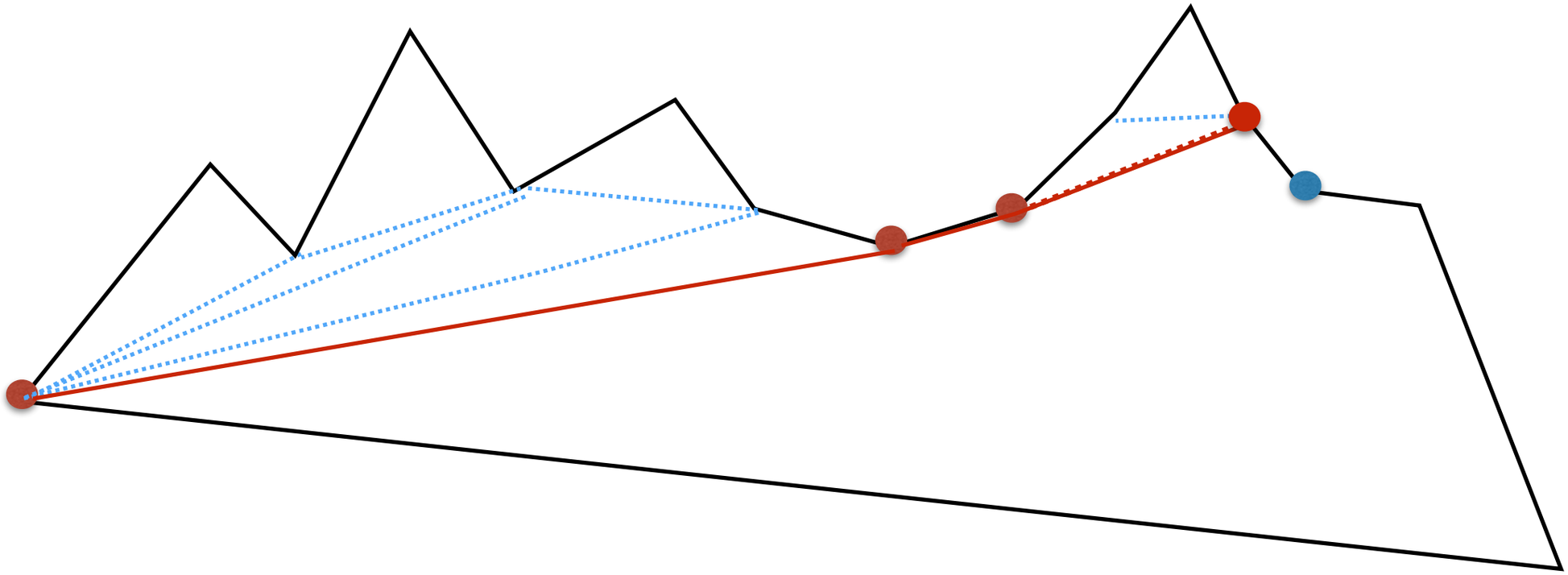
Monotone mountains are easy to triangulate!



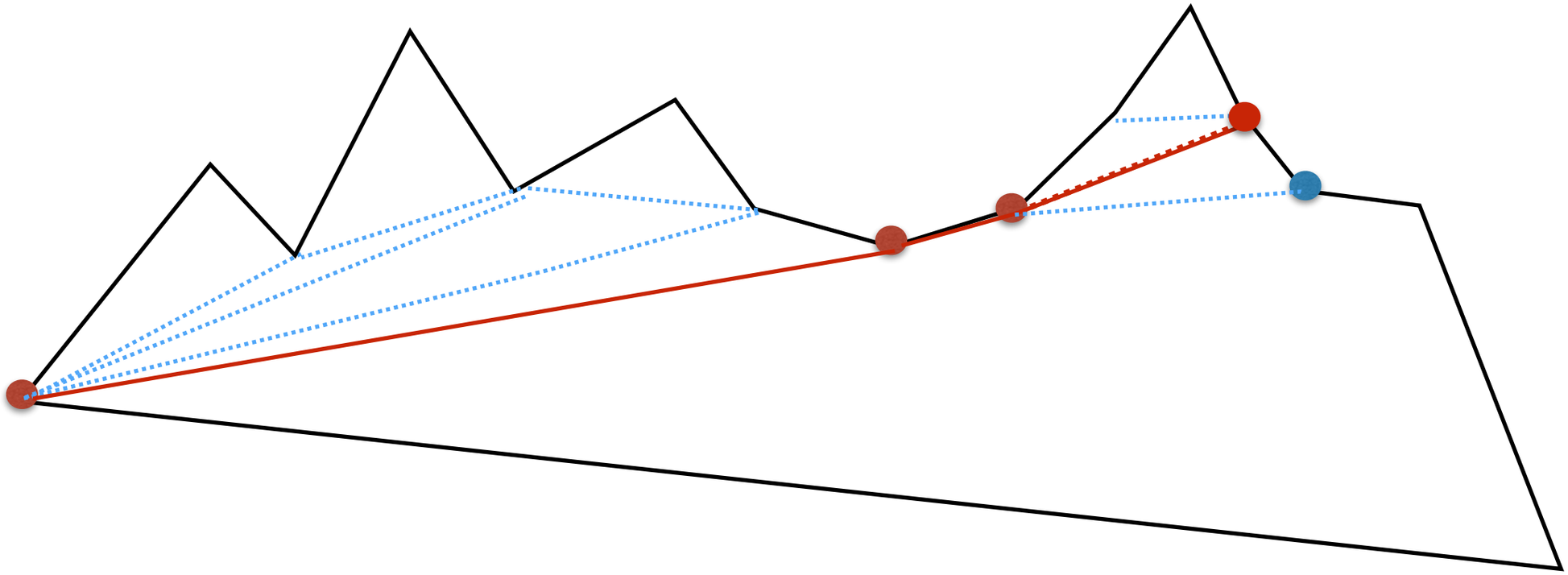
Monotone mountains are easy to triangulate!



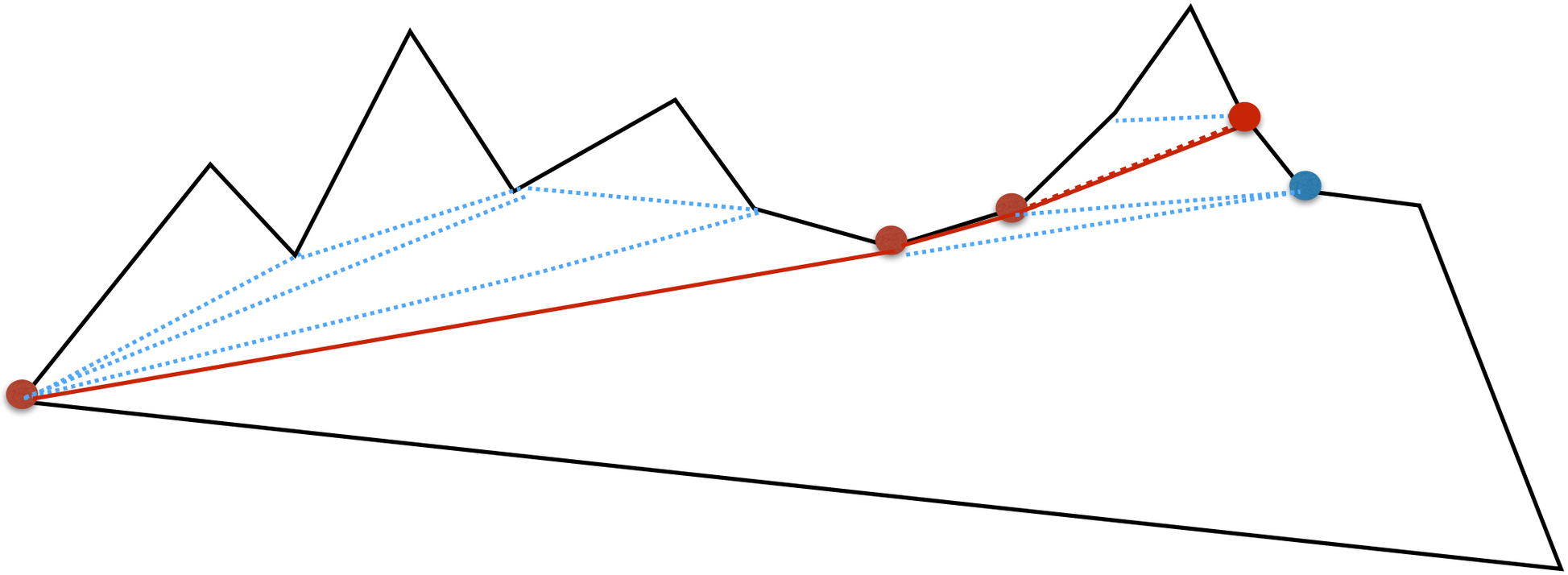
Monotone mountains are easy to triangulate!



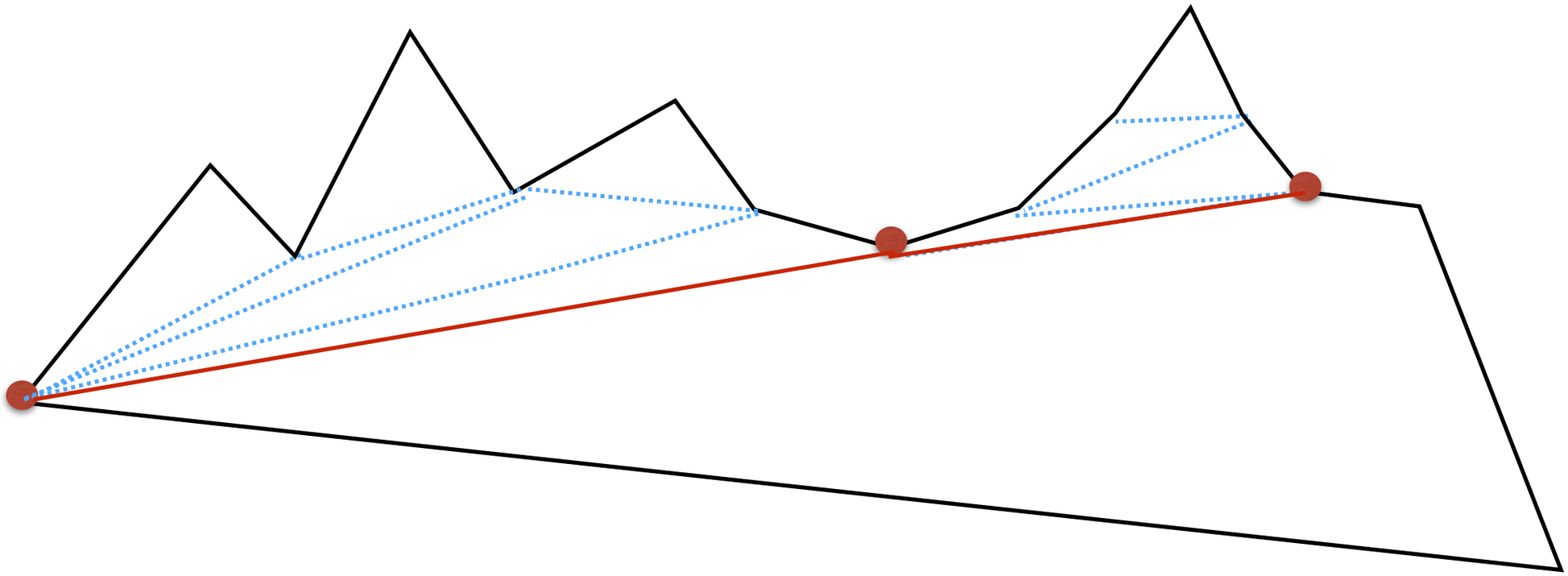
Monotone mountains are easy to triangulate!



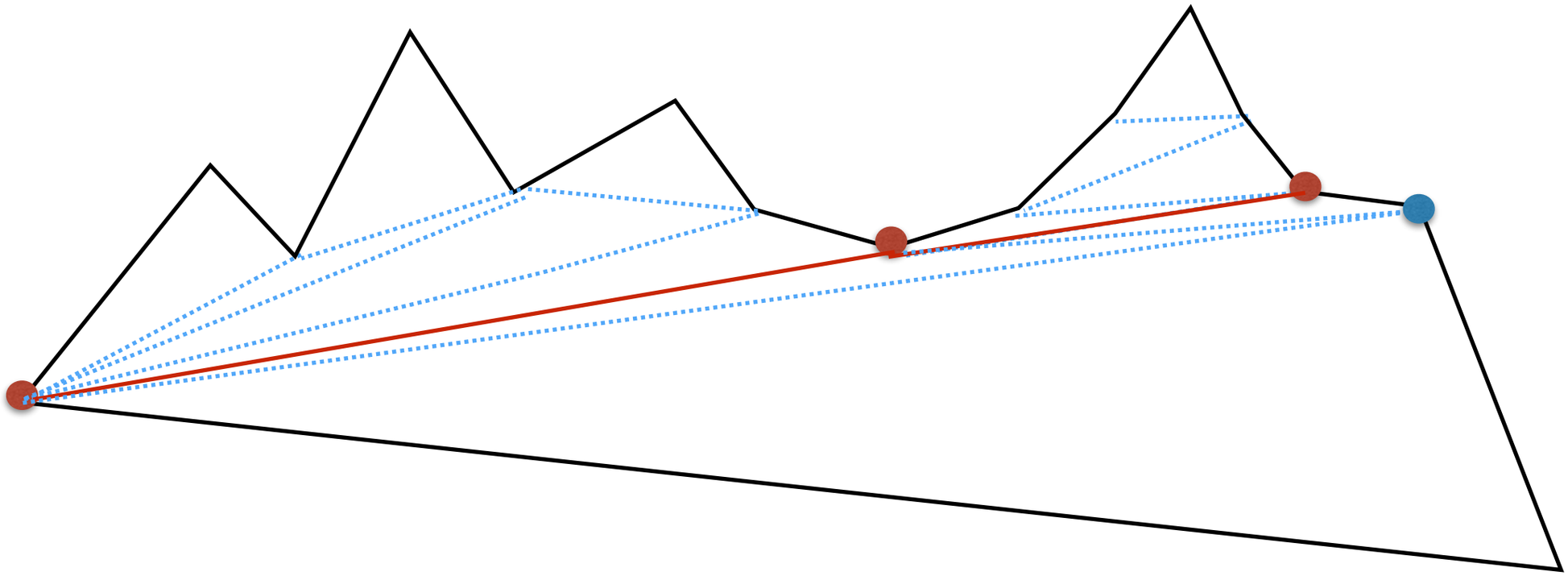
Monotone mountains are easy to triangulate!



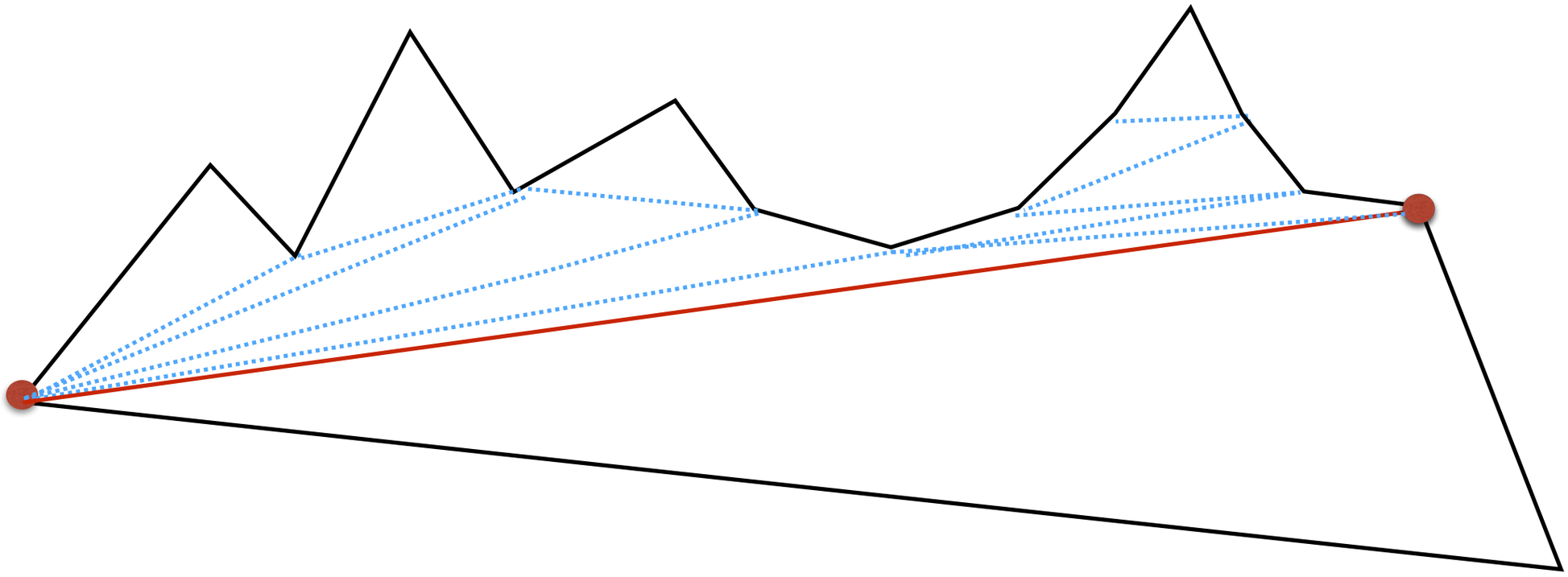
Monotone mountains are easy to triangulate!



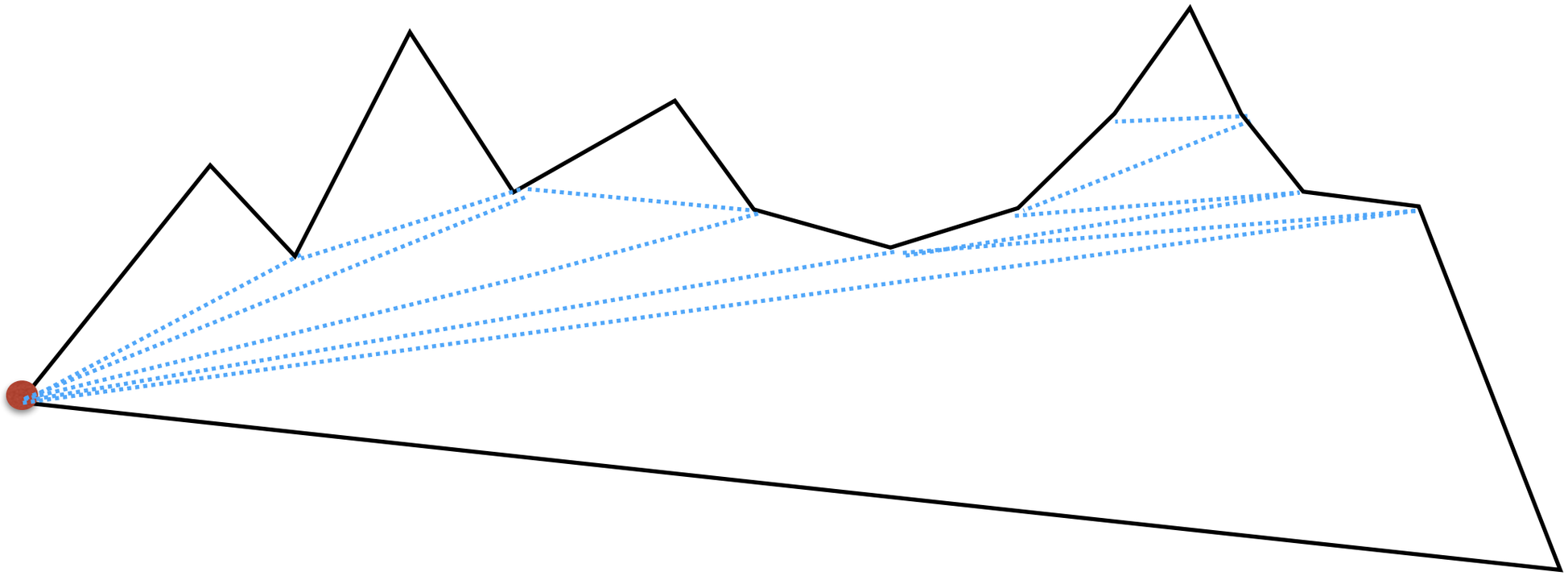
Monotone mountains are easy to triangulate!



Monotone mountains are easy to triangulate!

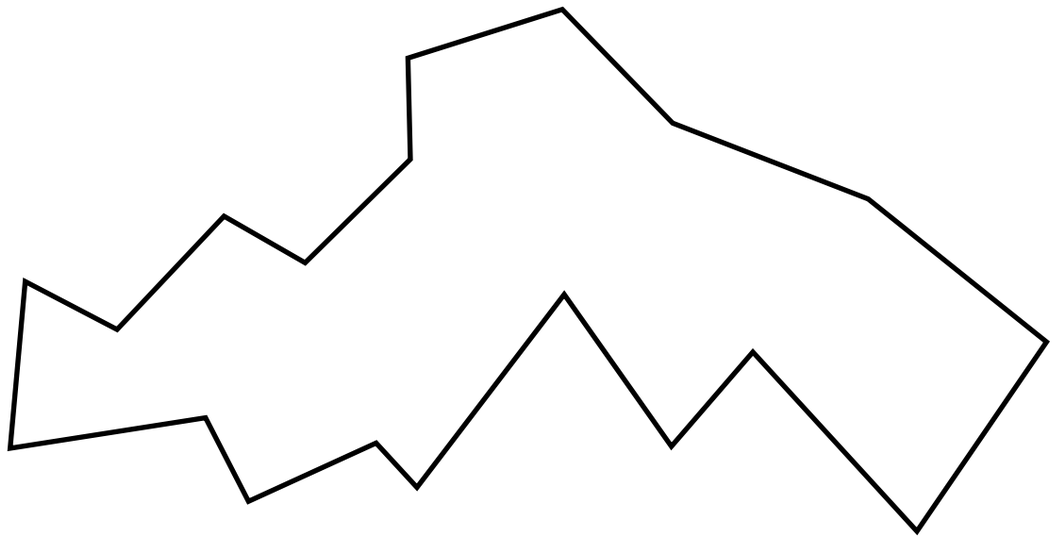


Monotone mountains are easy to triangulate!



Analysis: O(n) time

Monotone polygons

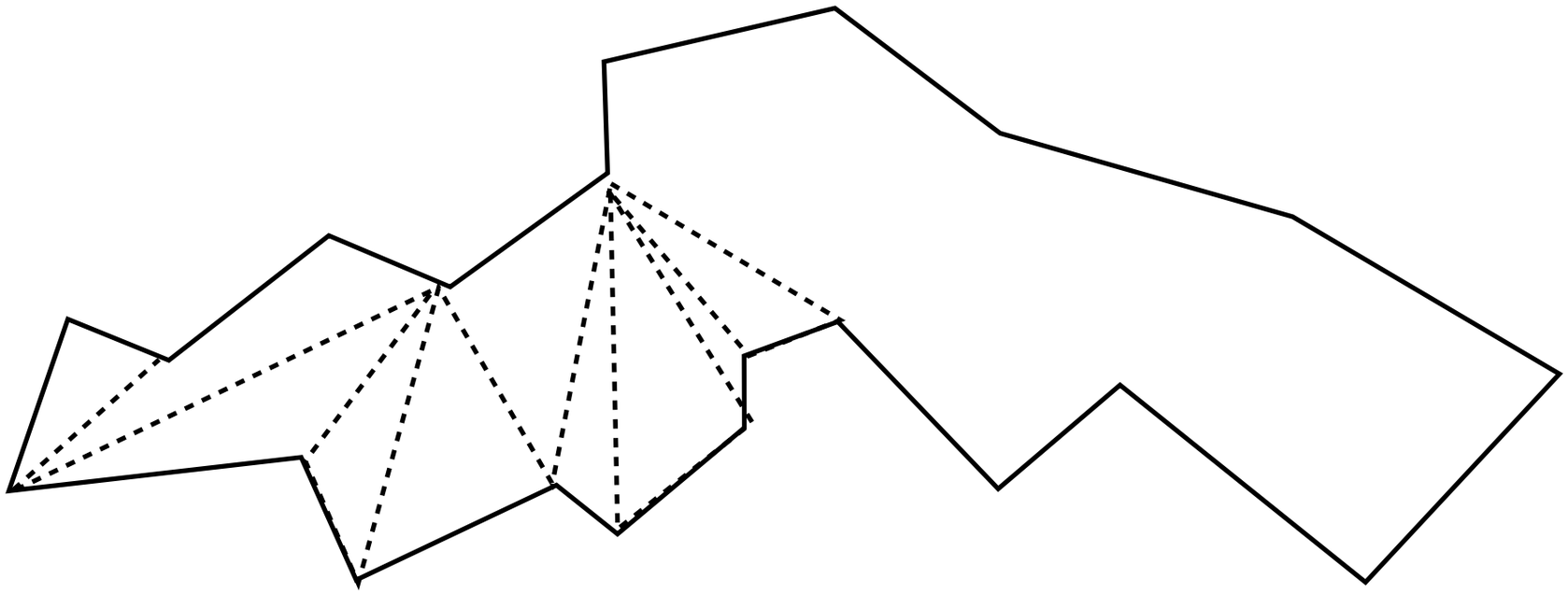


Monotone polygons can (also) be triangulated in O(n) time

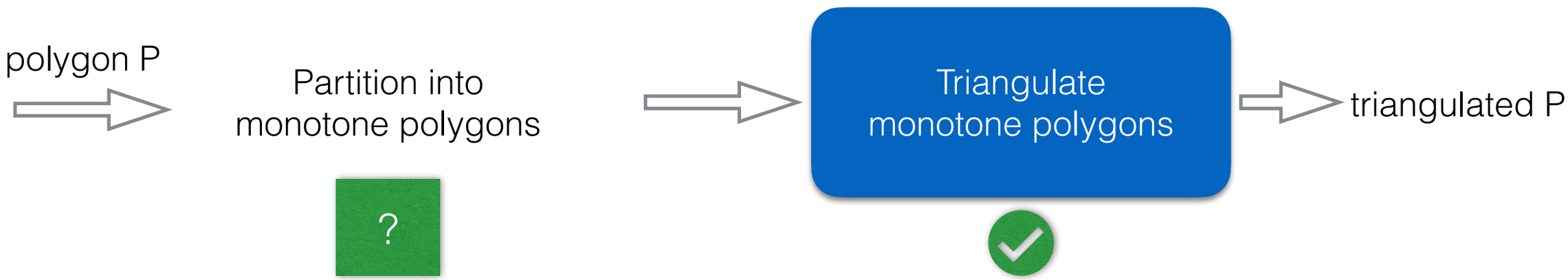
Monotone polygons

Same idea: pick the next vertex in x-order (on upper or lower chain). Connect it to all vertices after it that are visible via a diagonal. Continue to the next vertex.

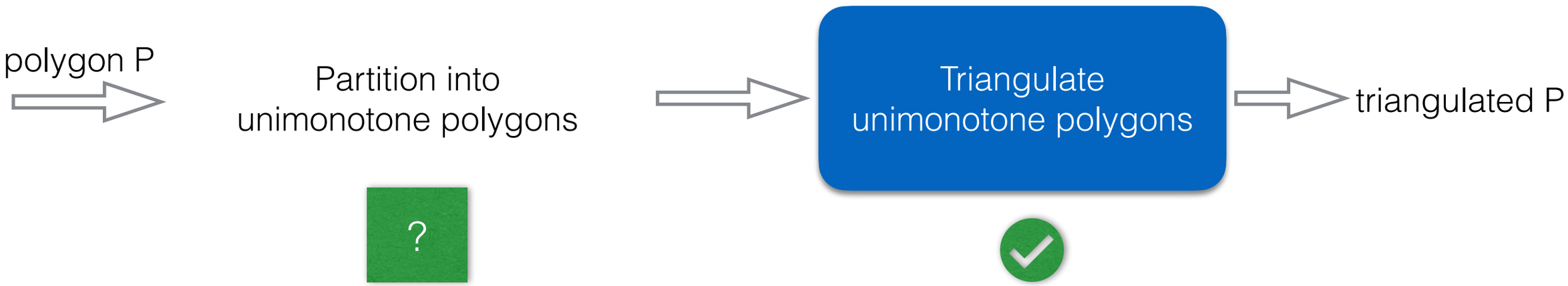
O(n) time



Towards an O(n lg n) Polygon Triangulation Algorithm



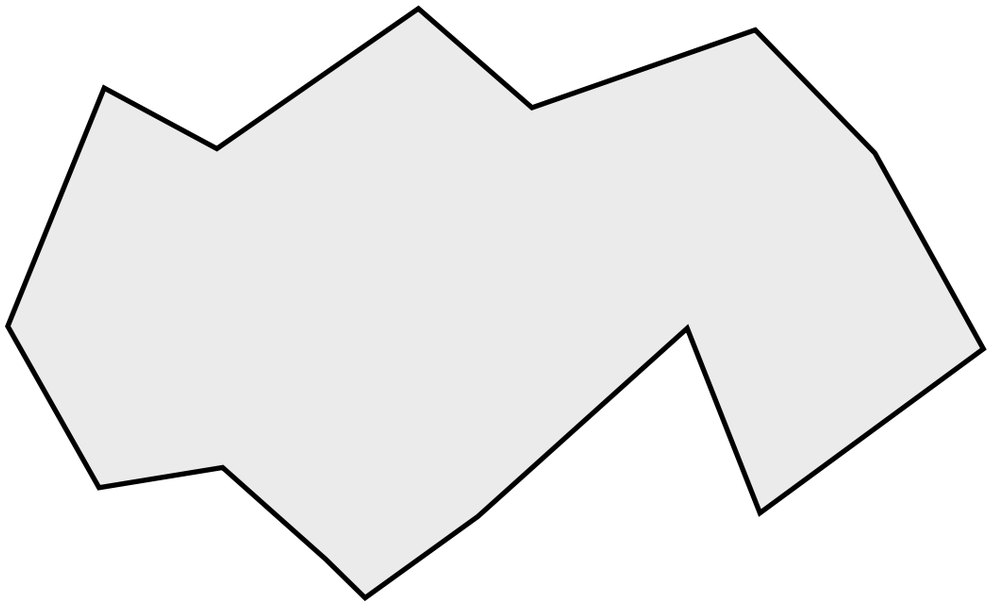
OR



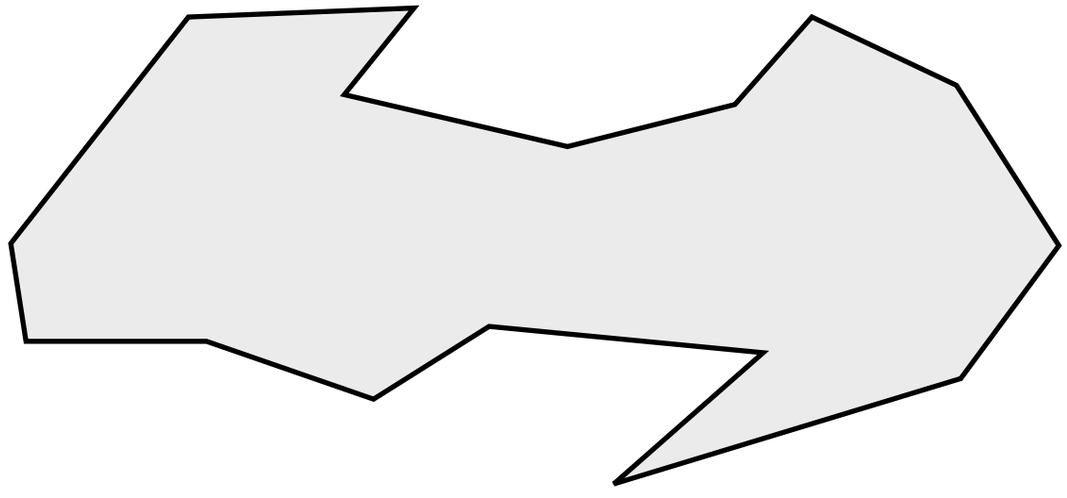
Want: partition a polygon into monotone pieces

(or: monotone mountains)

Intuition



x-monotone

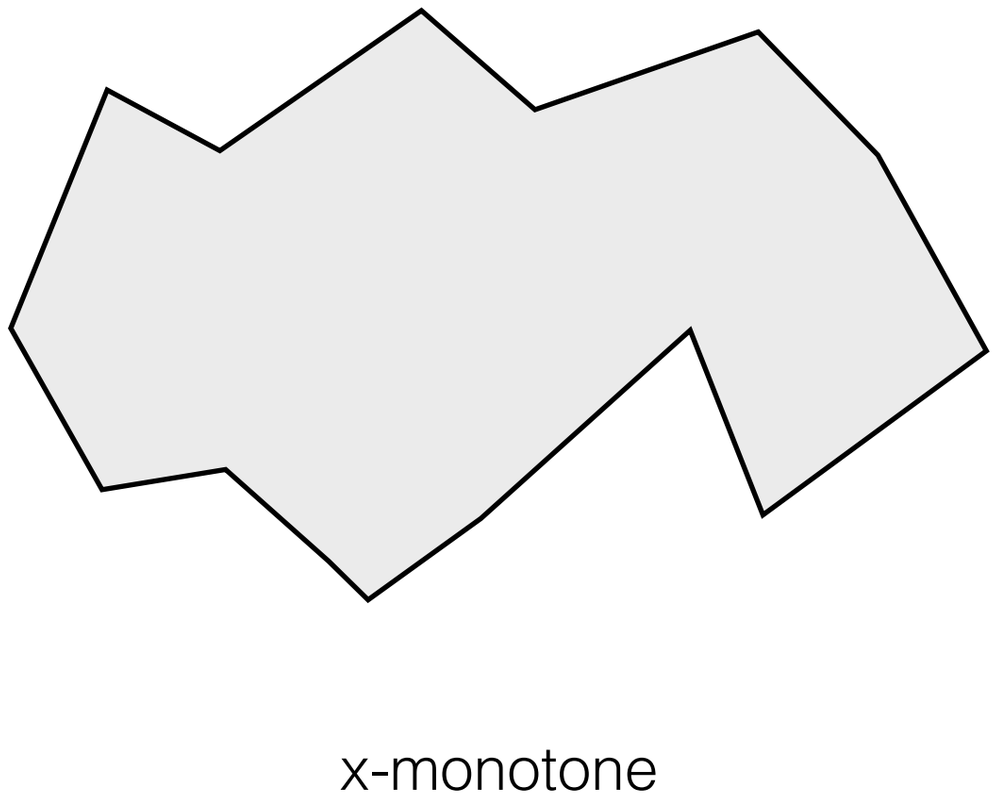


not x-monotone

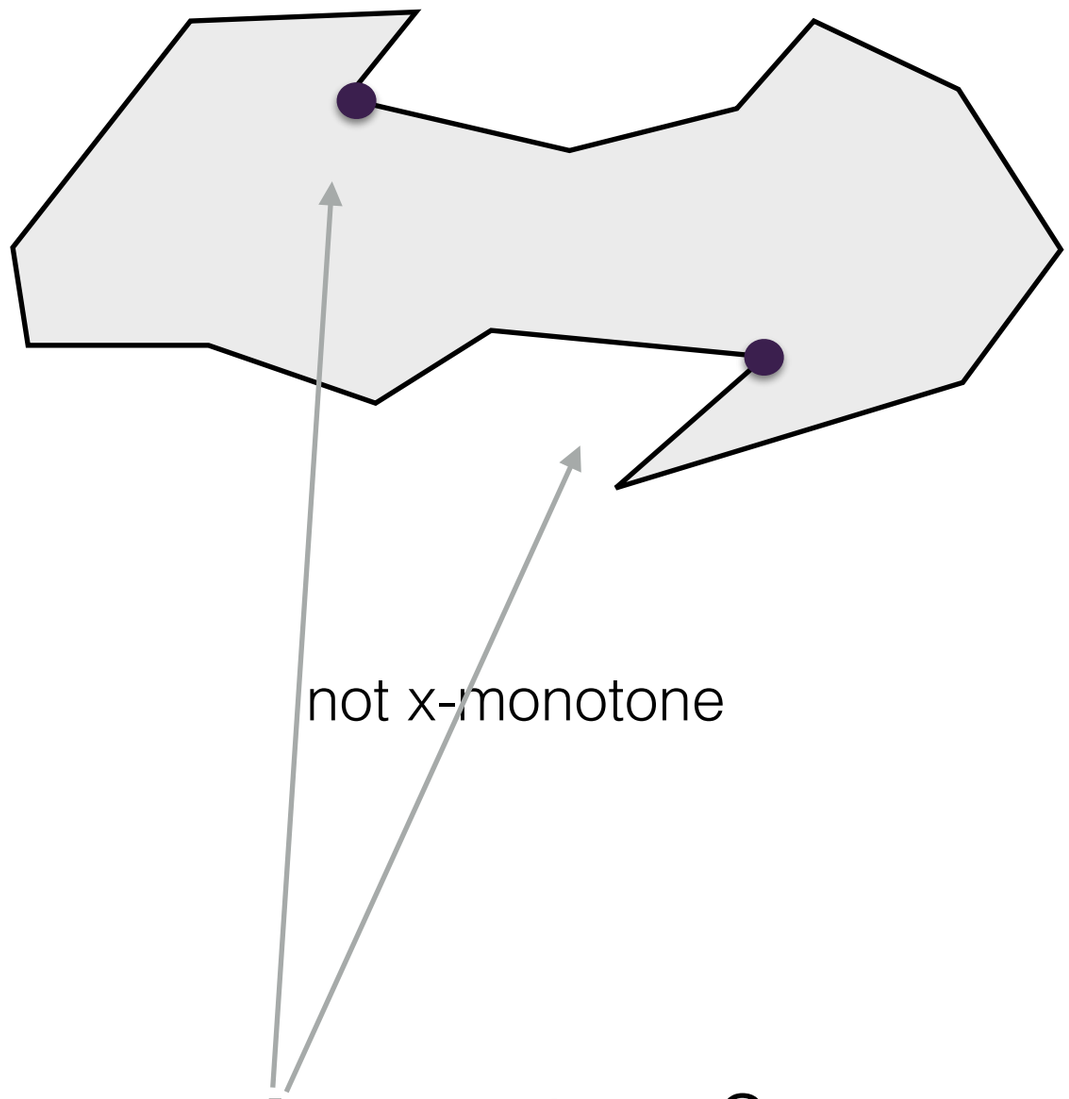
What makes a polygon **not** monotone?

Intuition

Cusp: a reflex vertex v such that the vertices before and after are both smaller or both larger than v (in terms of x-coords).



x-monotone



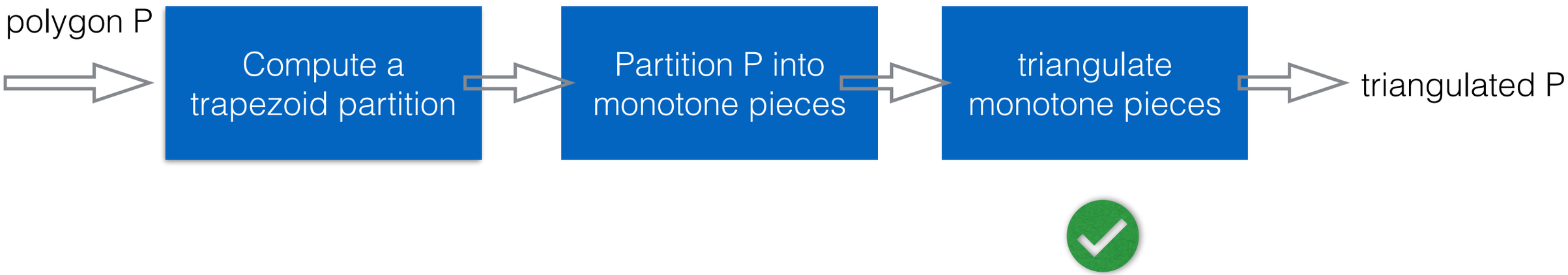
not x-monotone

What makes a polygon **not** monotone?

- Theorem: If a polygon has no cusps, then it's monotone.
- (Intuitively clear, but proof is tedious)

Overview: An O(n lg n) polygon triangulation algorithm

- We'll partition a polygon into monotone pieces by getting rid of cusps
- We'll do this by partitioning P into trapezoids (trapezoidation)



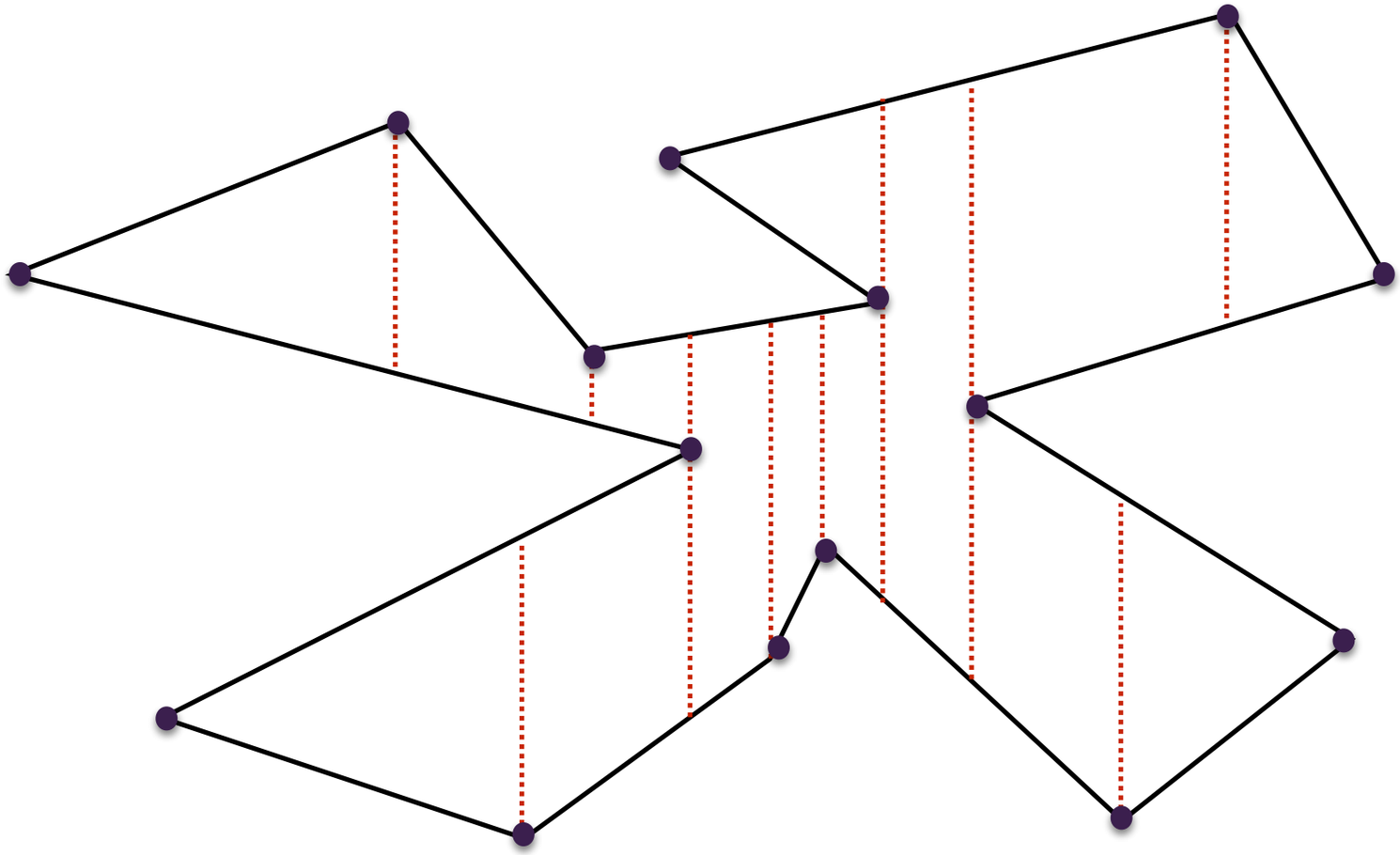
What on earth is a trapezoid partition?

- So, we want to partition a polygon into simpler pieces
- What is the simplest sort of partition that you can think of?
    - Shoot vertical rays and partition the interior of the polygon into vertical strips
    - That's exactly a trapezoid partition

Trapezoid partitions

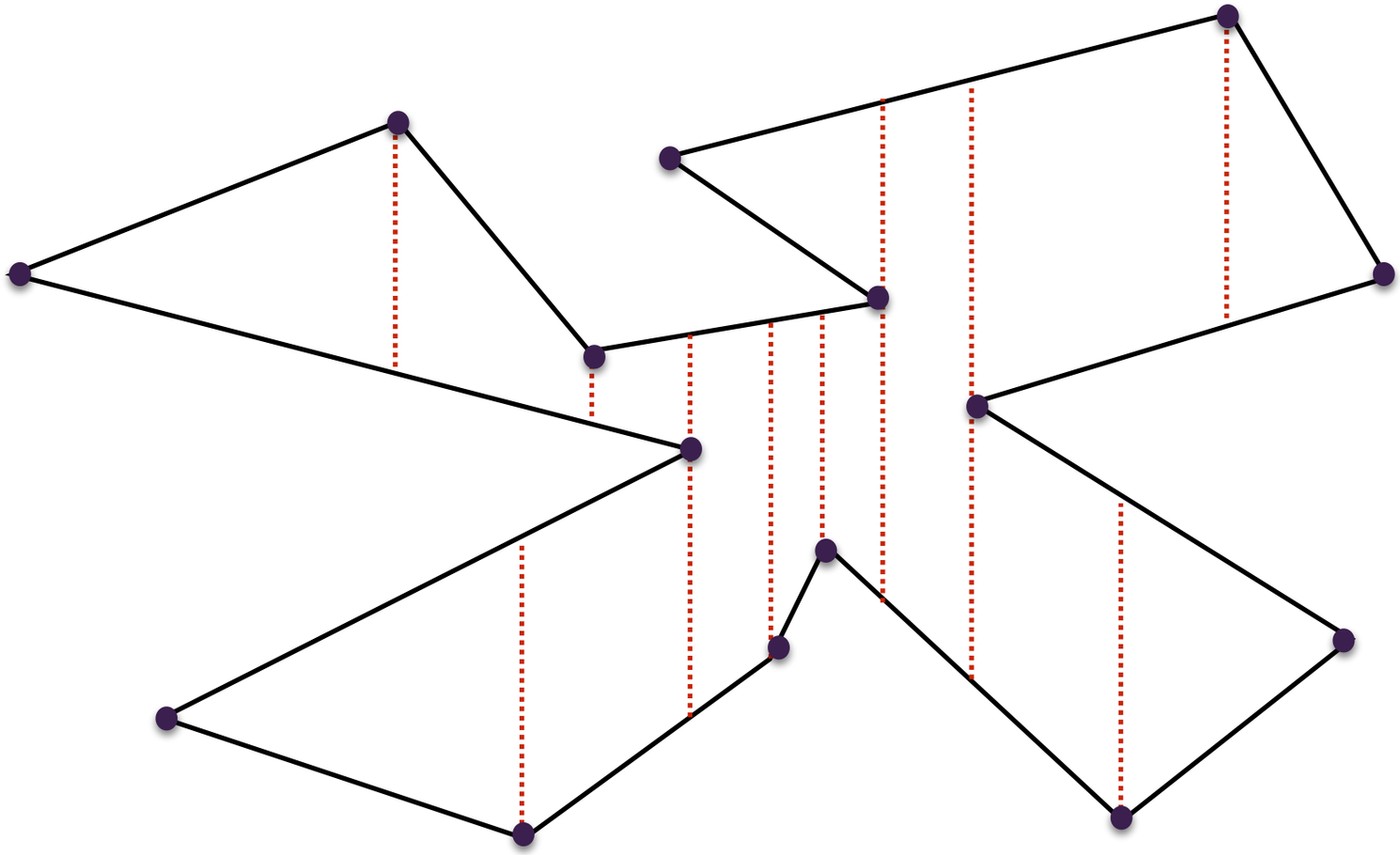
Shoot vertical rays from each vertex

- If polygon is above vertex, shoot vertical ray up (until reaches boundary)
- If f polygon is below vertex, shoot down
- If polygon is above and below vertex, shoot both up and down



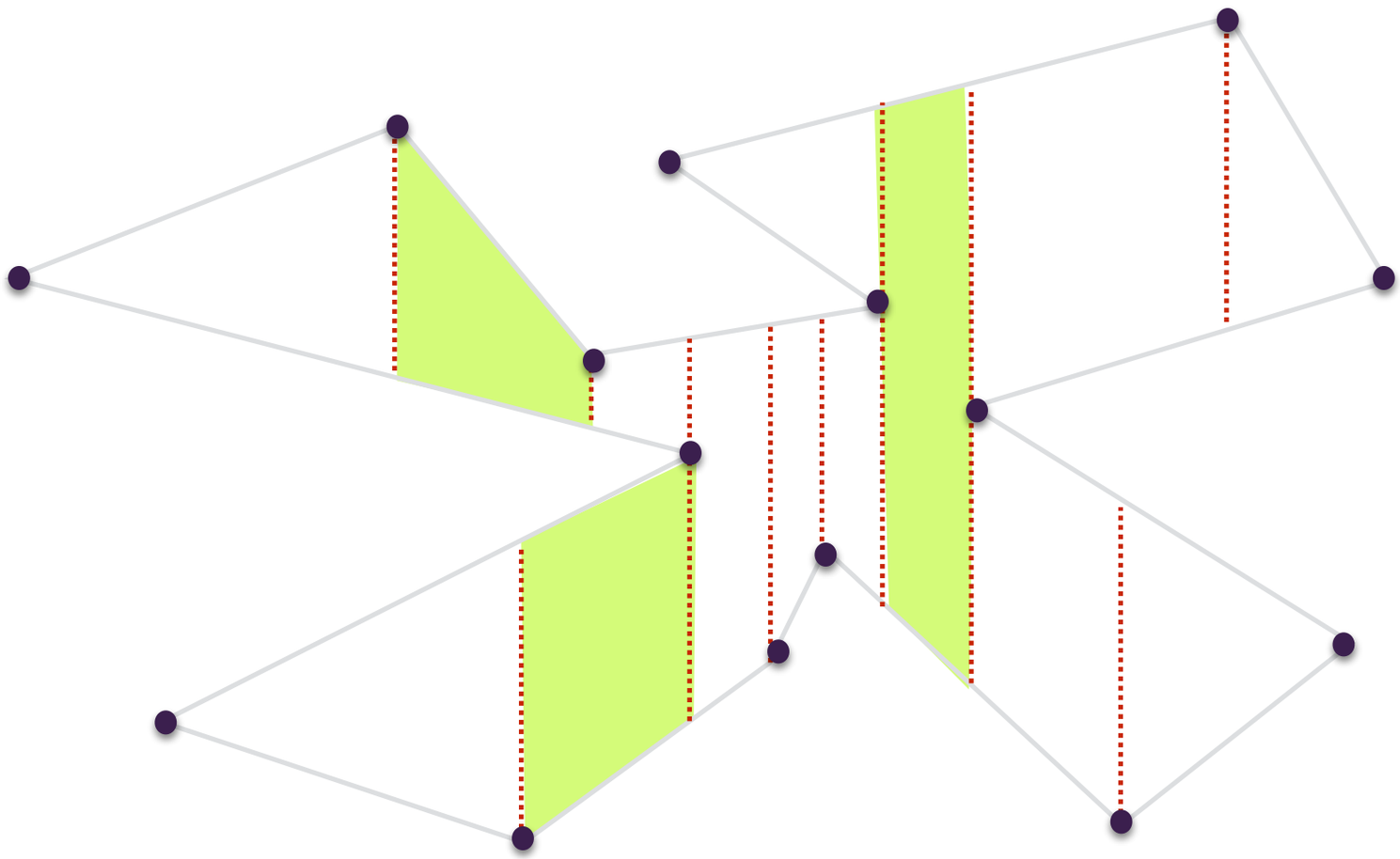
Trapezoid partitions: Properties

- Each polygon in the partition is a trapezoid, because:
    - It has one or two rays as sides.
    - If it has two, they must both hit the same edge above, and the same edge below.
- Size of trapezoid partition?
    - At most one ray through each vertex => O(n) threads => O(n) trapezoids



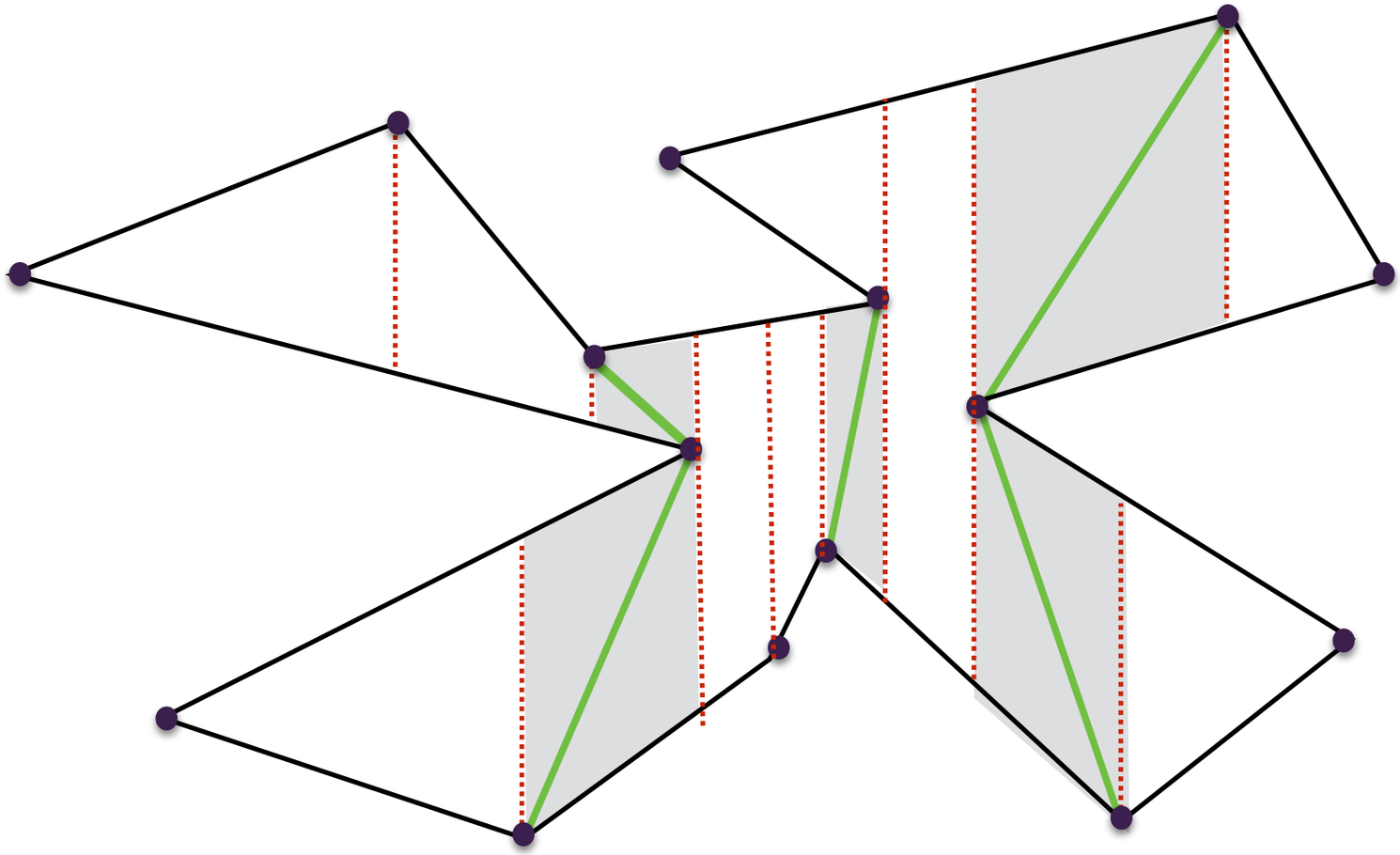
Trapezoid partitions

- Each trapezoid has precisely two vertices of the polygon, one on the left and one on the right. They can be on the top, bottom or middle of the trapezoid.



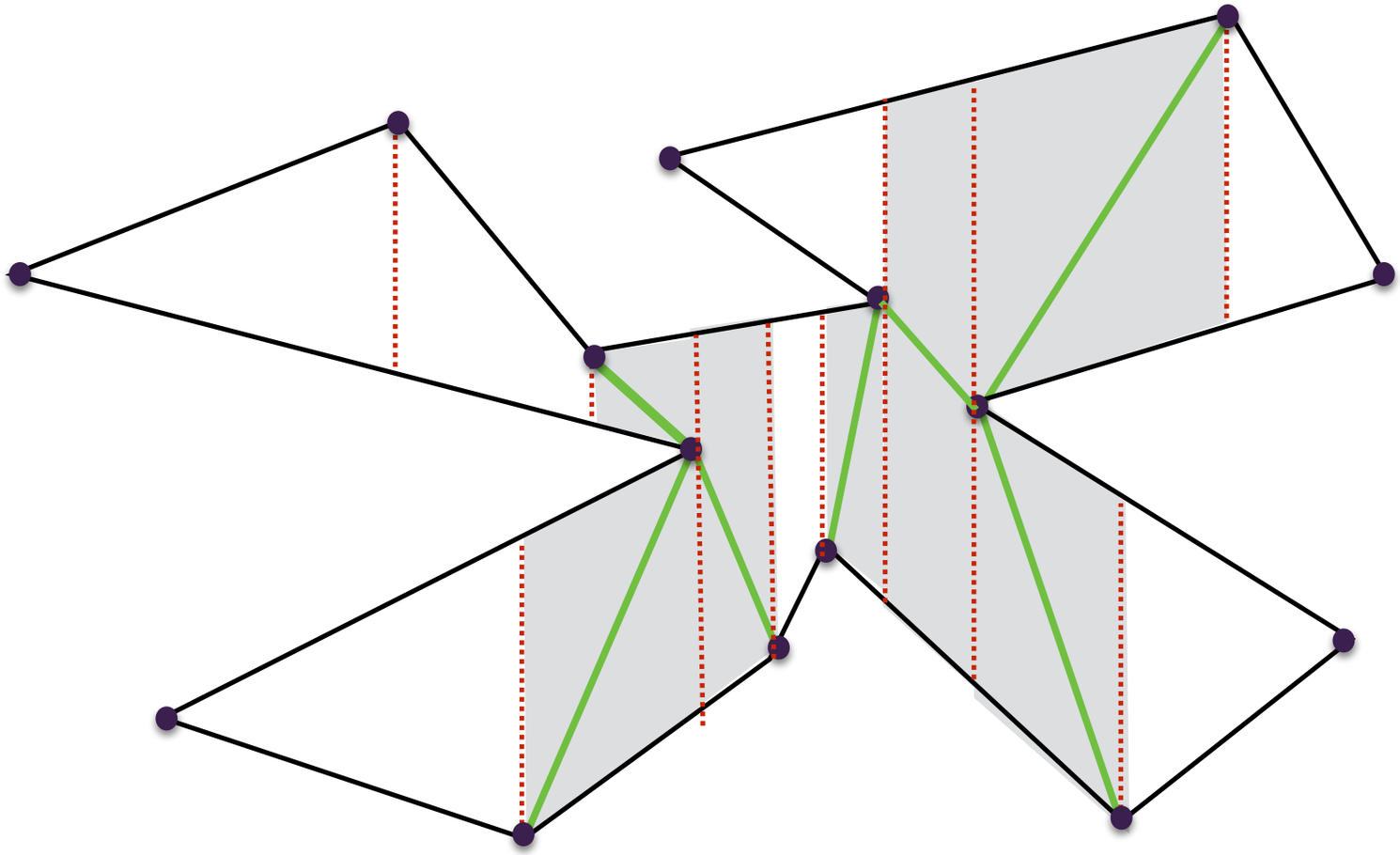
Diagonals

- In each trapezoid: if its two vertices are **not** on the same edge, they define a **diagonal**.
    - Why? The interior of a trapezoid cannot have any vertices/edges in its it, so the vertices of P on its boundary must be visible to each other.
    - If we know the trapezoid partition, this gives us (some) diagonals of P



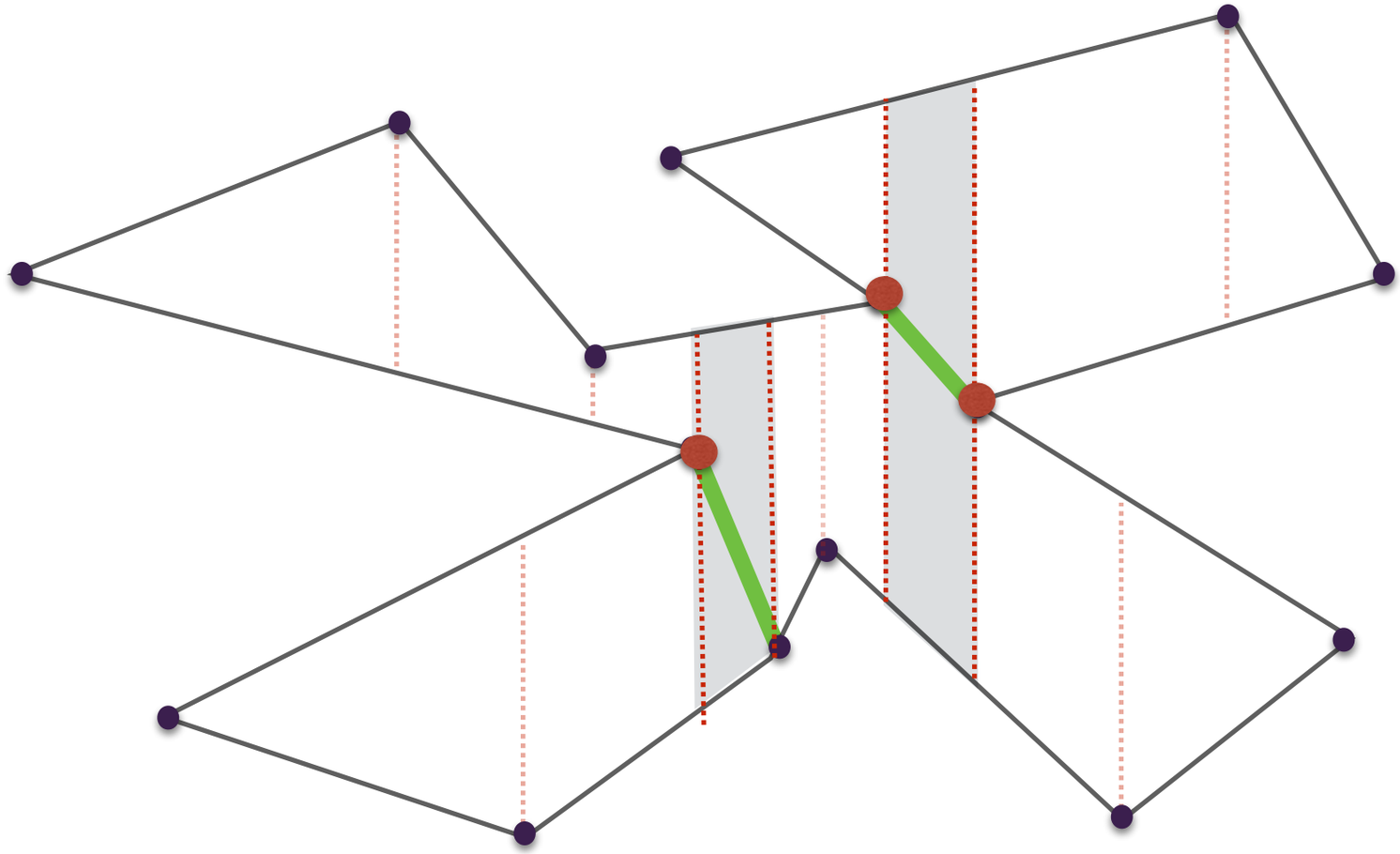
Diagonals

- In each trapezoid: if its two vertices are **not** on the same edge, they define a **diagonal**.
    - Why? The interior of a trapezoid cannot have any vertices/edges in its it, so the vertices of P on its boundary must be visible to each other.
    - If we know the trapezoid partition, this gives us (some) diagonals of P



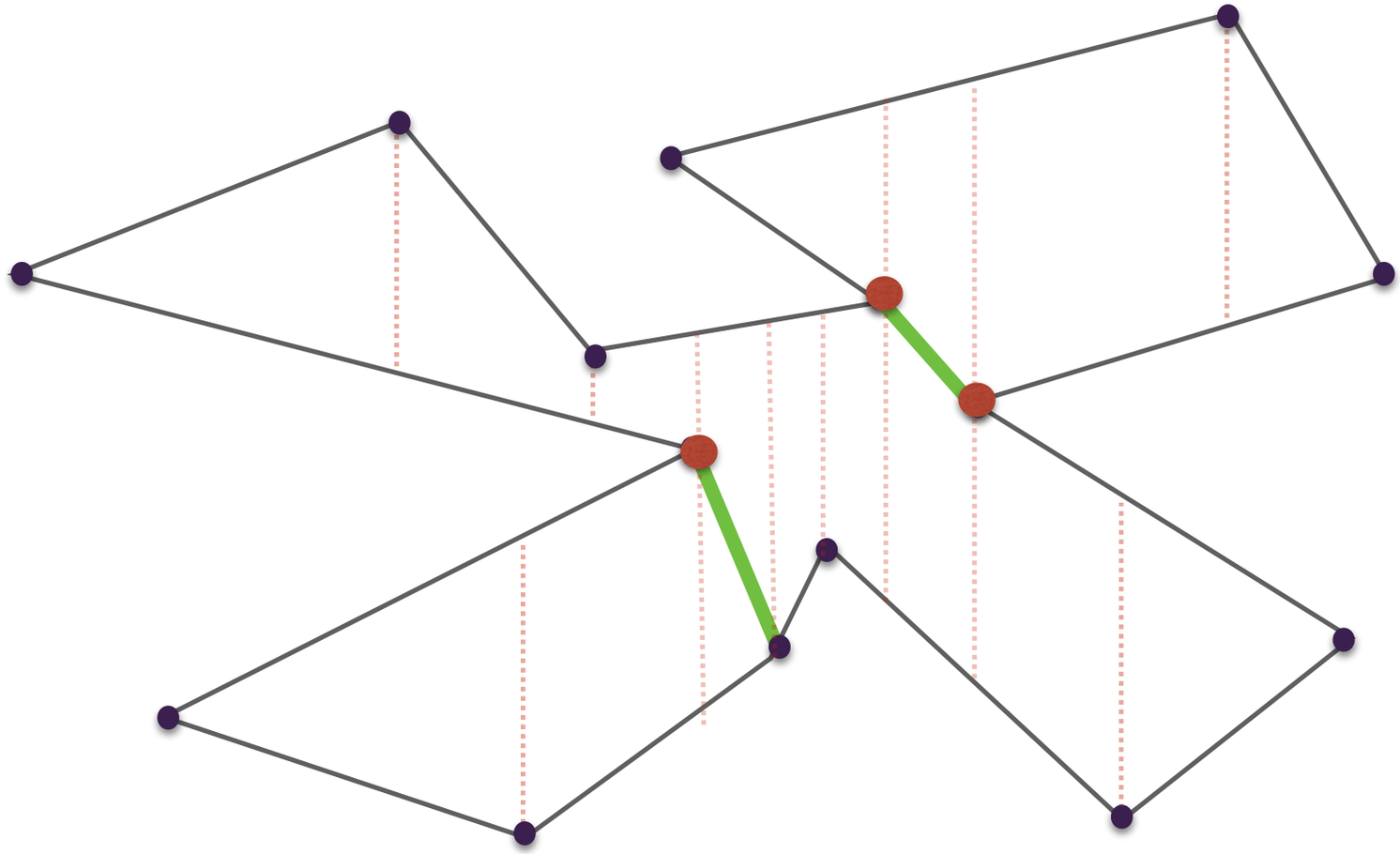
Remember our problem: cusp vertices

- For a leftward cusp vertex: consider the trapezoid to its right
- For a rightward cusp: consider the trapezoid to its left
- Consider those diagonals.



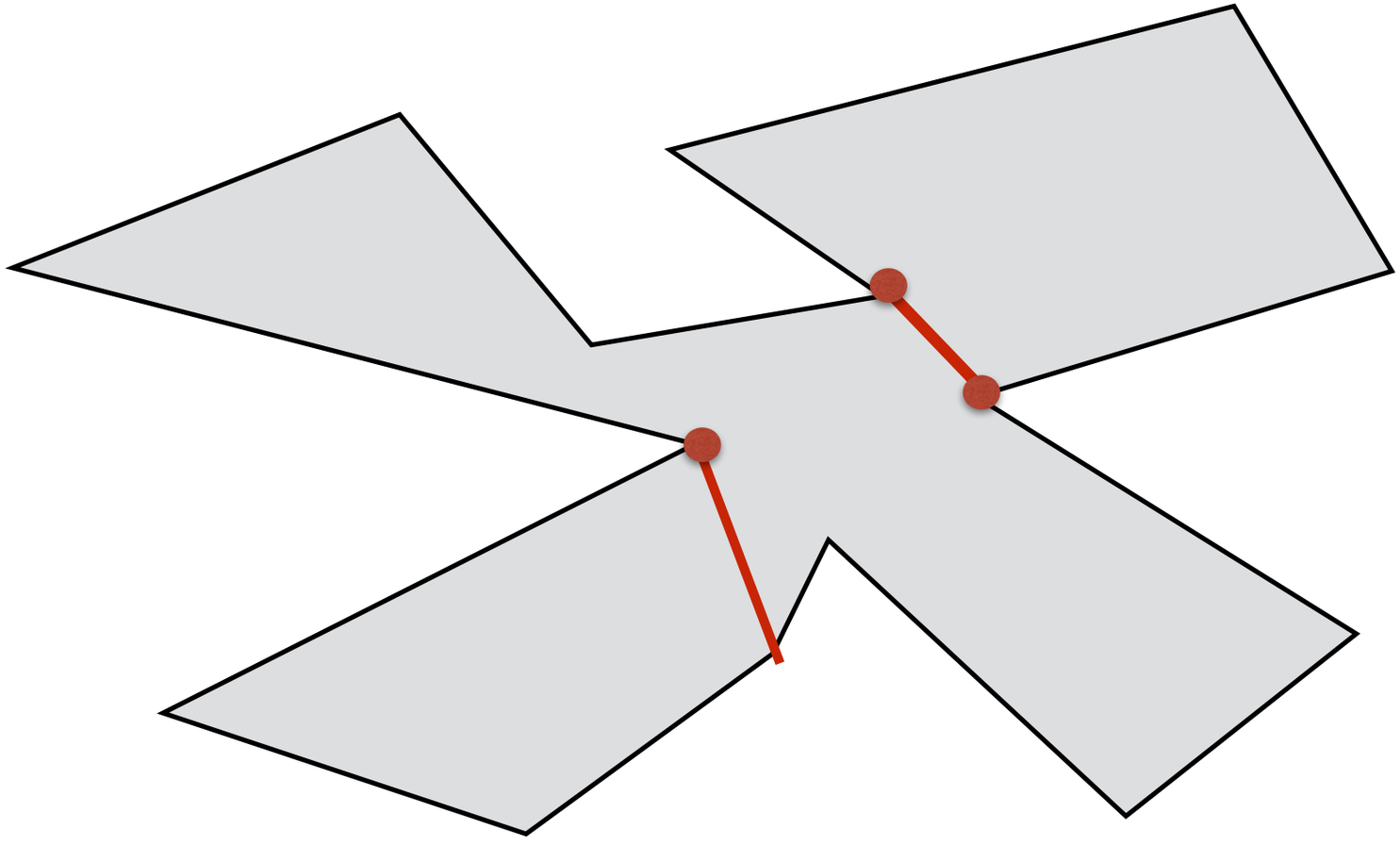
Remember our problem: cusp vertices

- For a leftward cusp vertex: consider the trapezoid to its right
- For a rightward cusp: consider the trapezoid to its left
- Consider those diagonals. They partition P into (3) pieces.



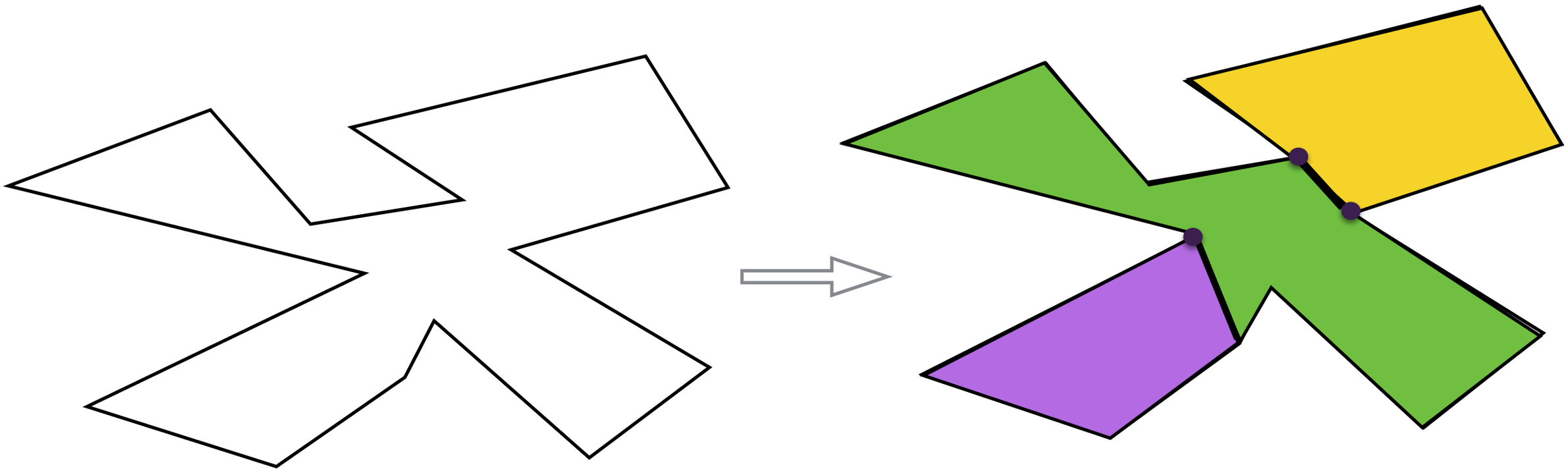
Remember our problem: cusp vertices

- For a leftward cusp vertex: consider the trapezoid to its right
- For a rightward cusp: consider the trapezoid to its left
- Consider those diagonals. They partition P into (3) pieces.
- Claim: These pieces are monotone!



What have we done?

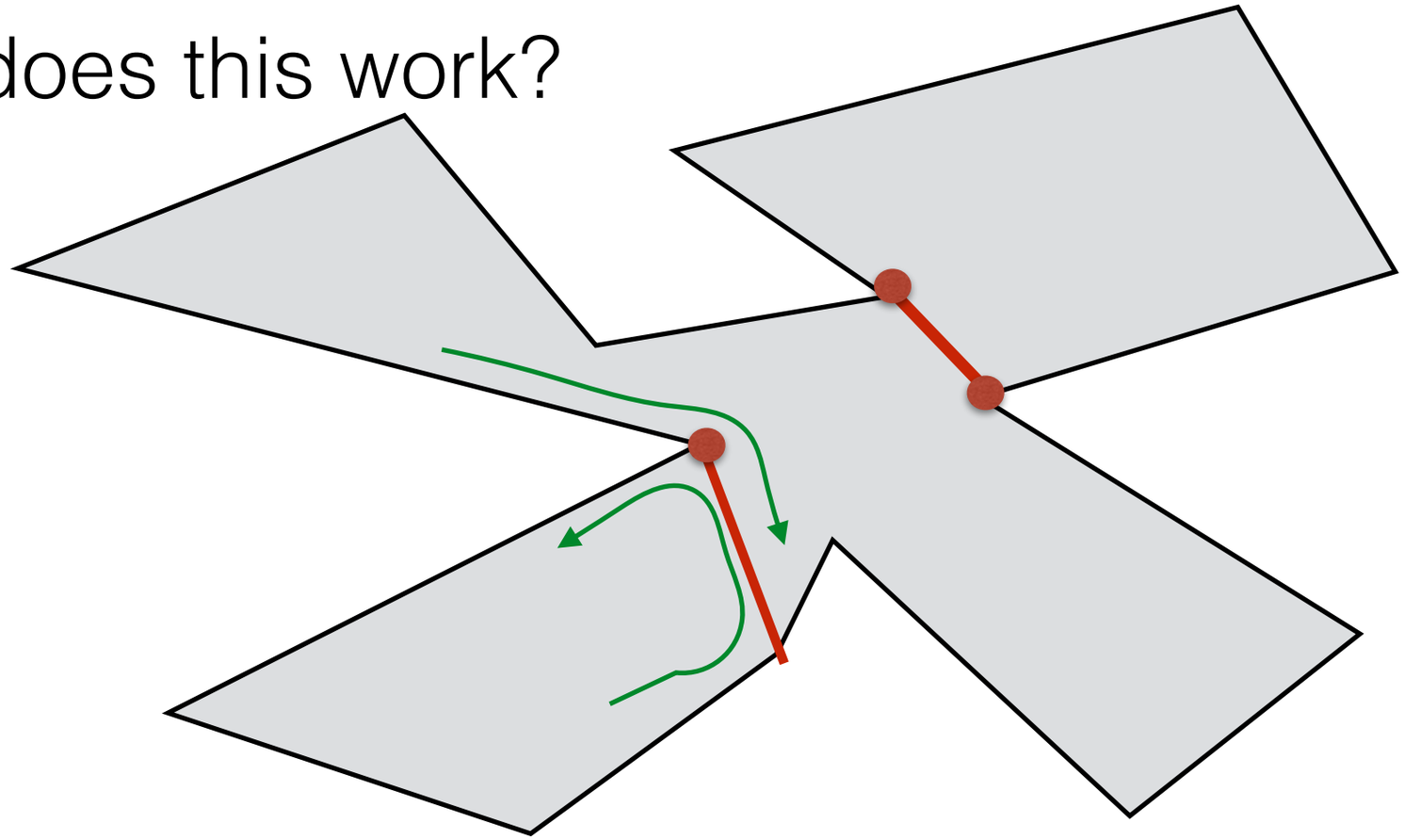
1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For each cusp: Add diagonal before/after

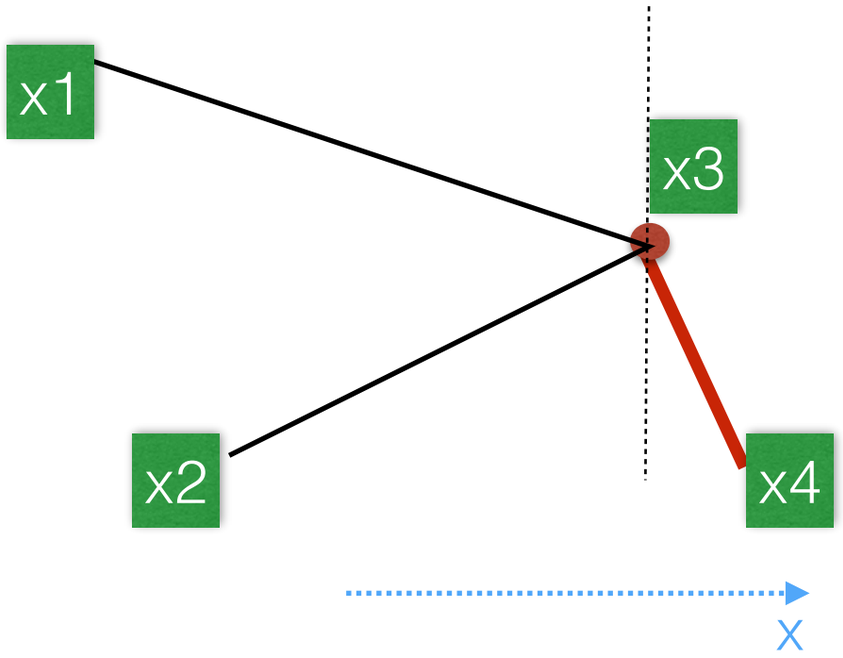


x

Claim: these pieces are x-monotone

Why does this work?



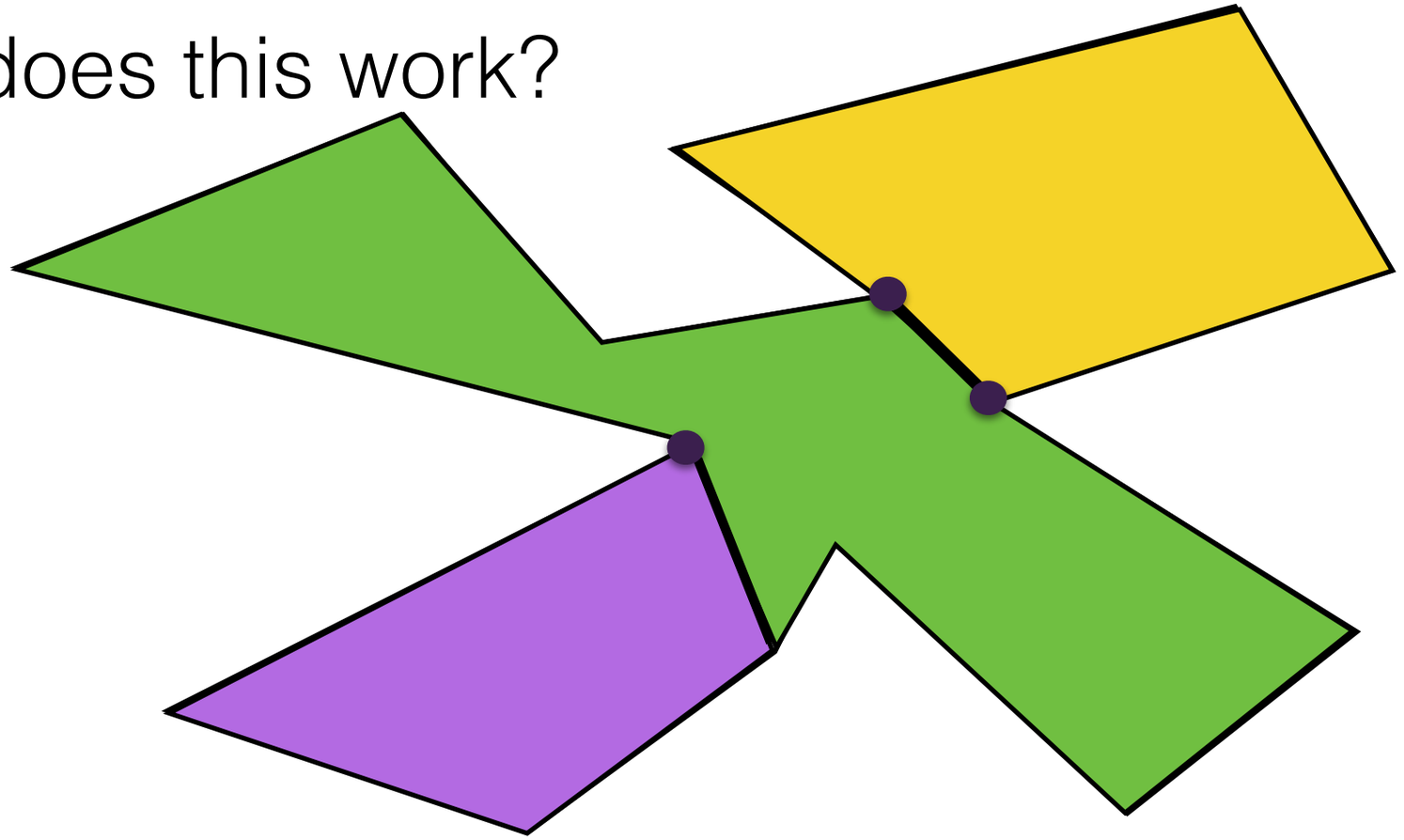


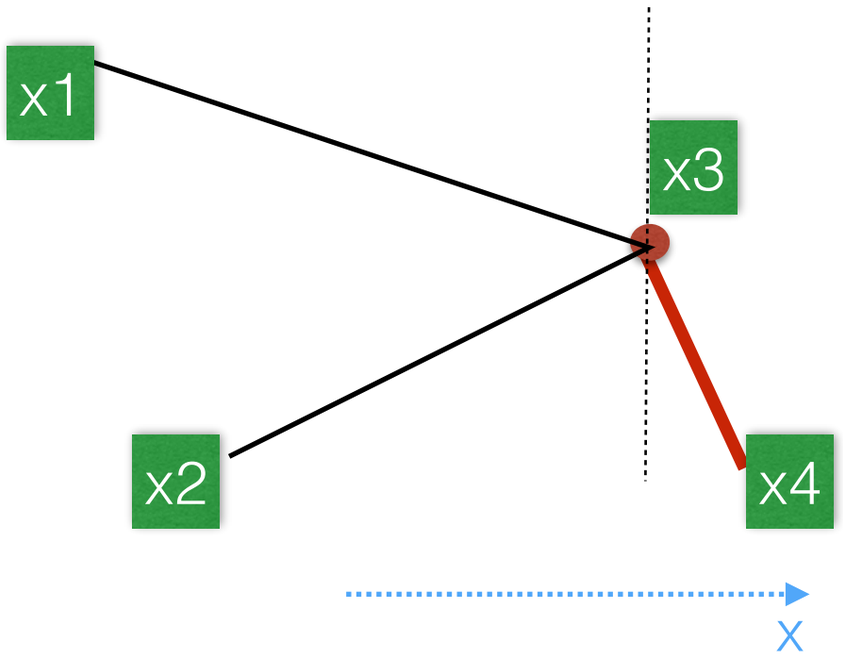
$$x_1 < x_3 < x_4$$

$$x_2 < x_3 < x_4$$

diagonal is after the cusp: $x_3 < x_4$

Why does this work?





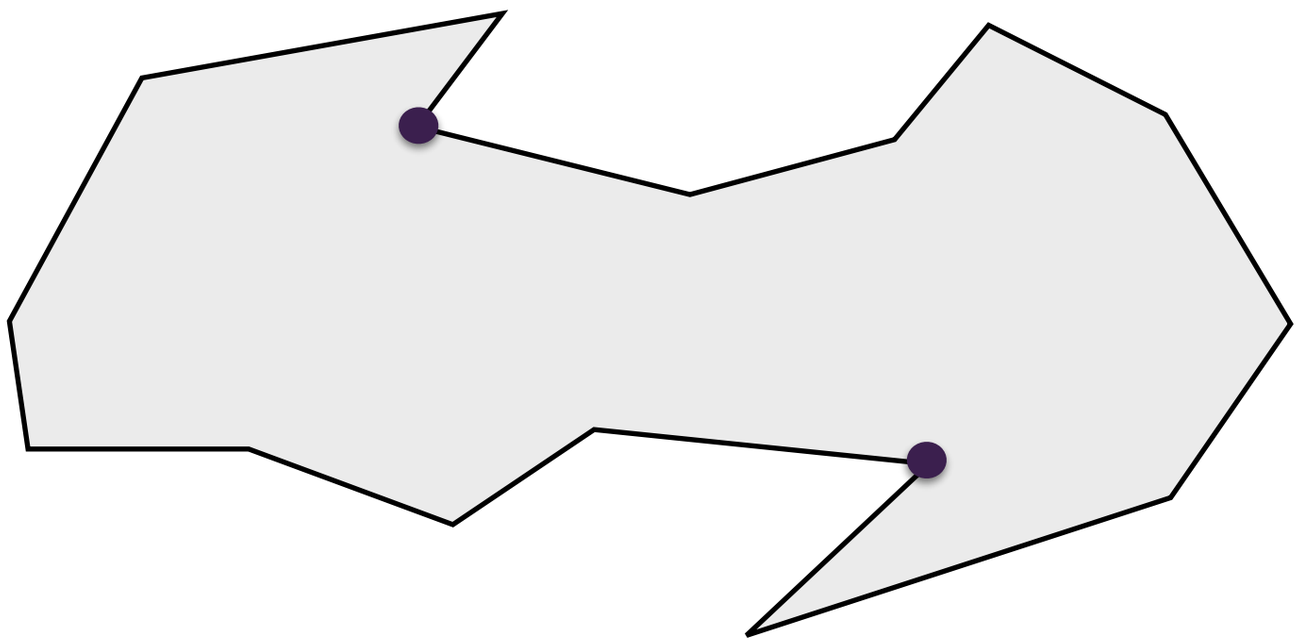
$$x_1 < x_3 < x_4$$

$$x_2 < x_3 < x_4$$

diagonal is after the cusp: $x_3 < x_4$

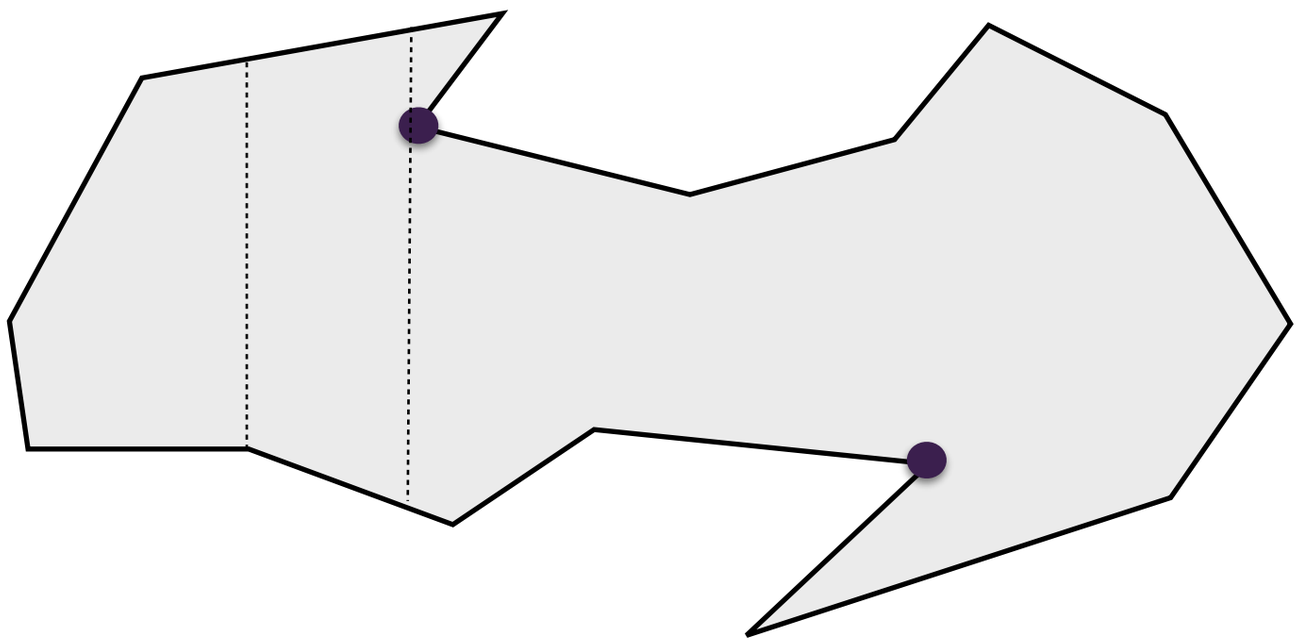
Another example

Partitioning a polygon into monotone pieces

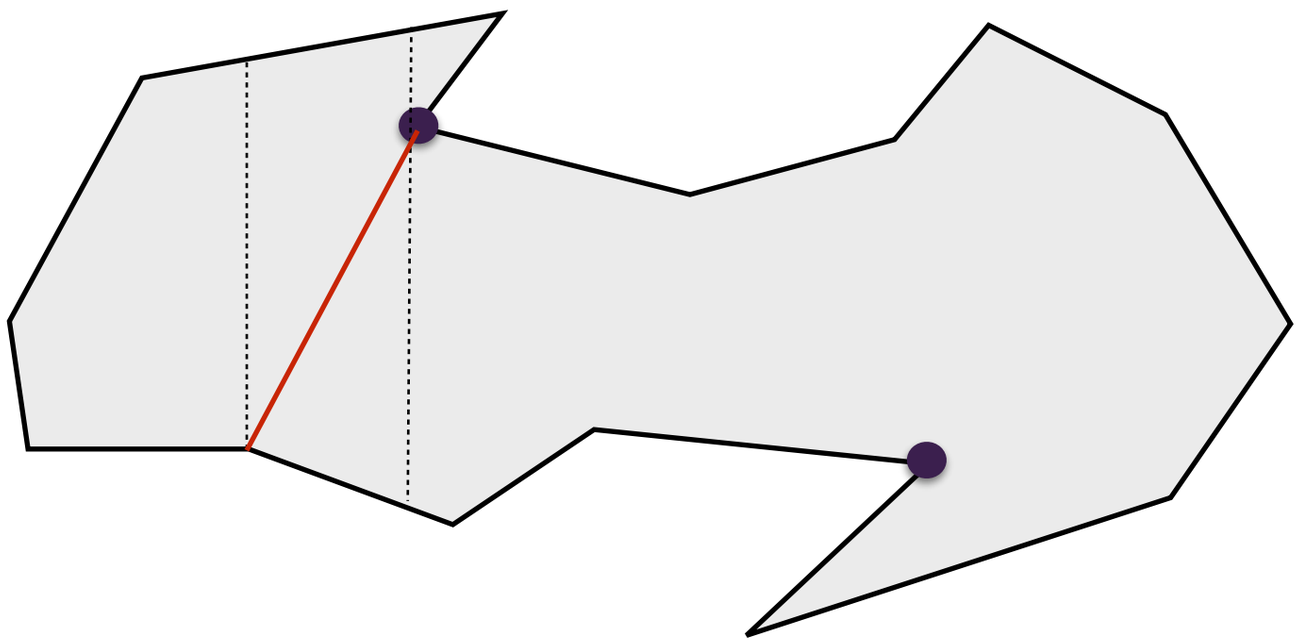


1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For left cusp vertices: add diagonal in trapezoid before the cusp
4. For right cusp vertices: add diagonal in trapezoid after the cusp

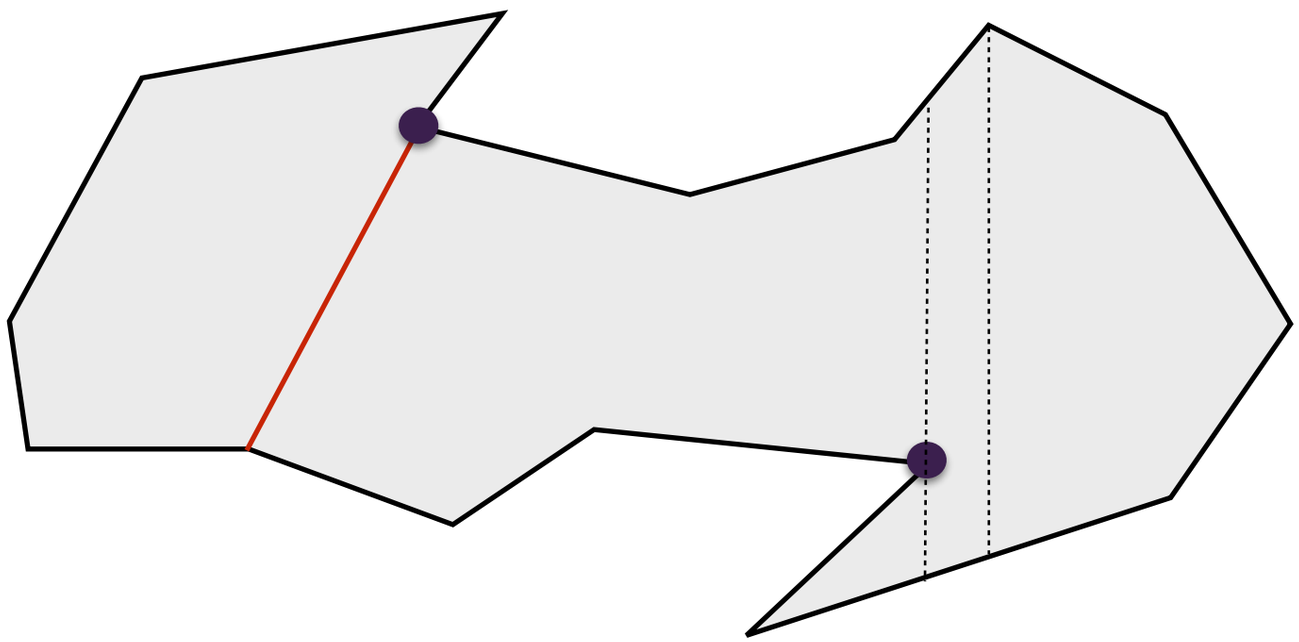
Partitioning a polygon into monotone pieces



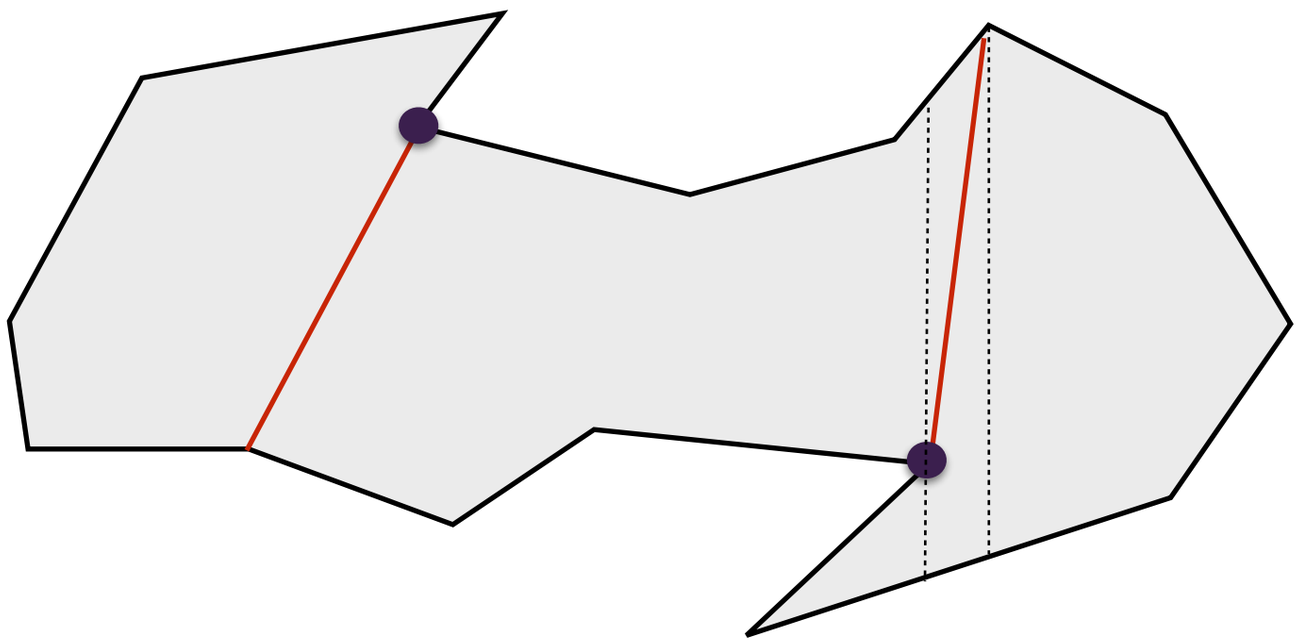
Partitioning a polygon into monotone pieces



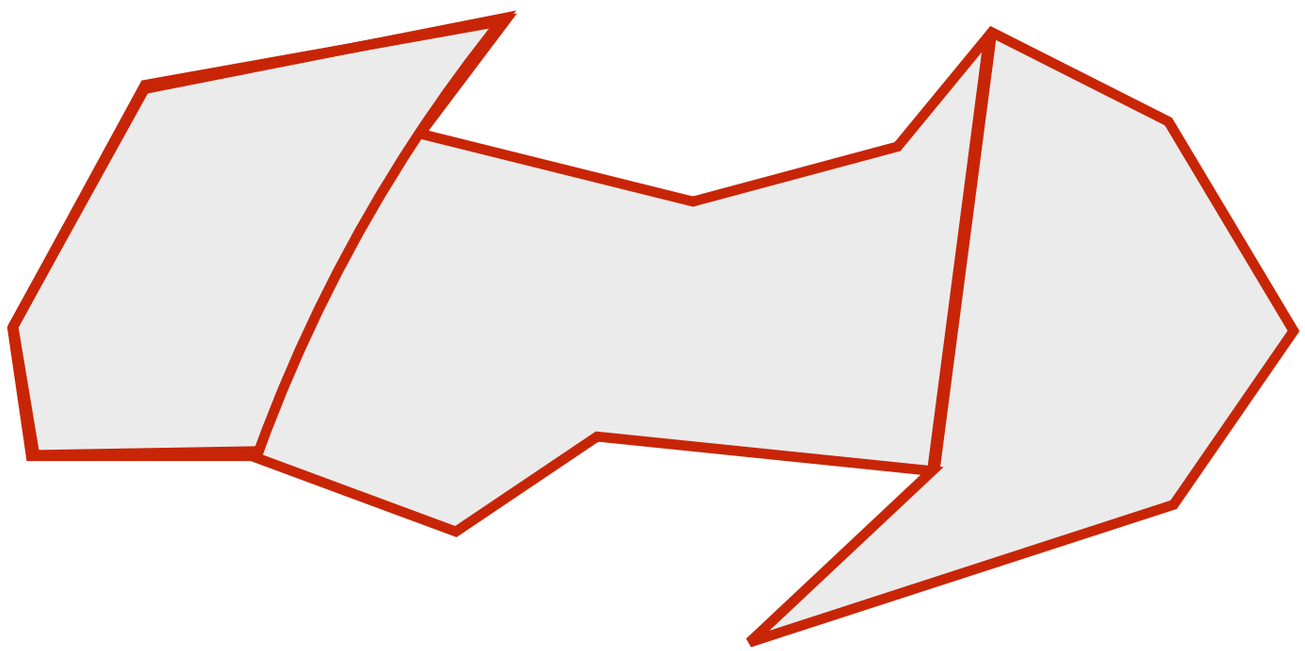
Partitioning a polygon into monotone pieces



Partitioning a polygon into monotone pieces



Partitioning a polygon into monotone pieces

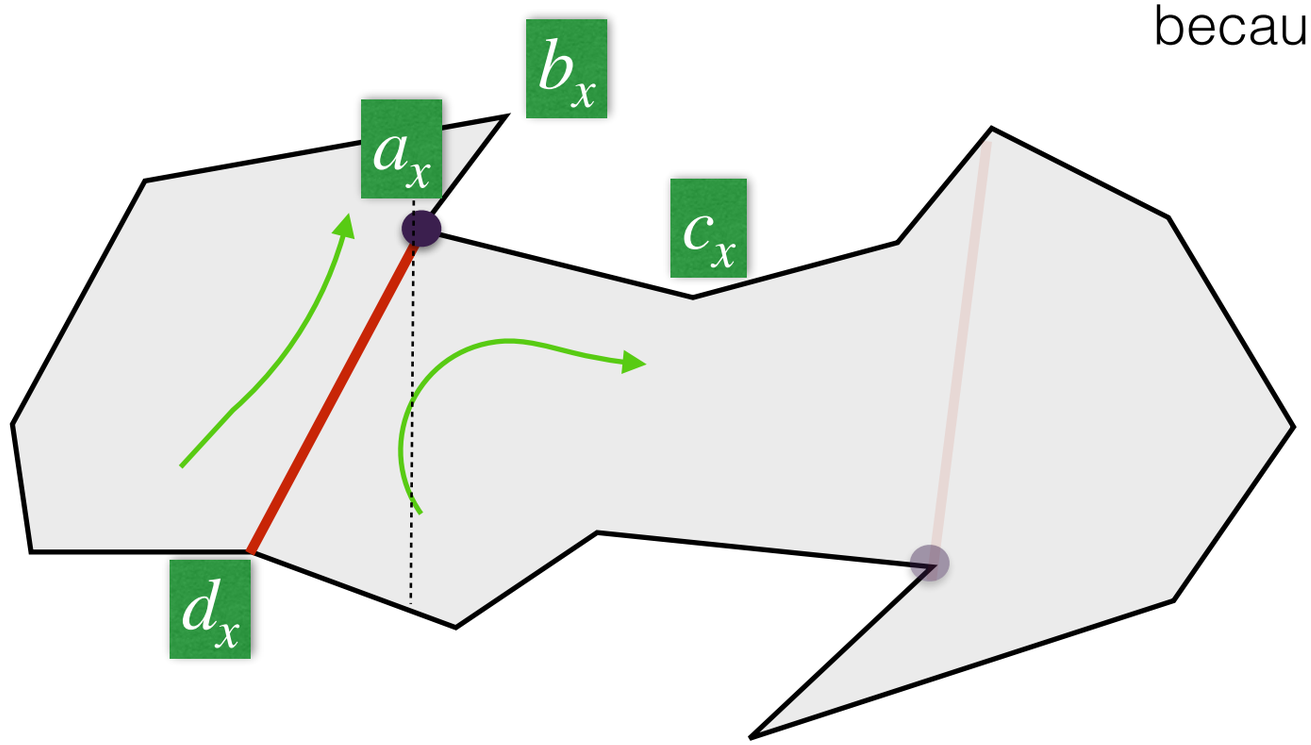


1. Identify cusp vertices
2. Compute a trapezoid partition of P
3. For left cusp vertices: add diagonal in trapezoid before the cusp
4. For right cusp vertices: add diagonal in trapezoid after the cusp

This partition P into pieces, and these pieces are monotone.

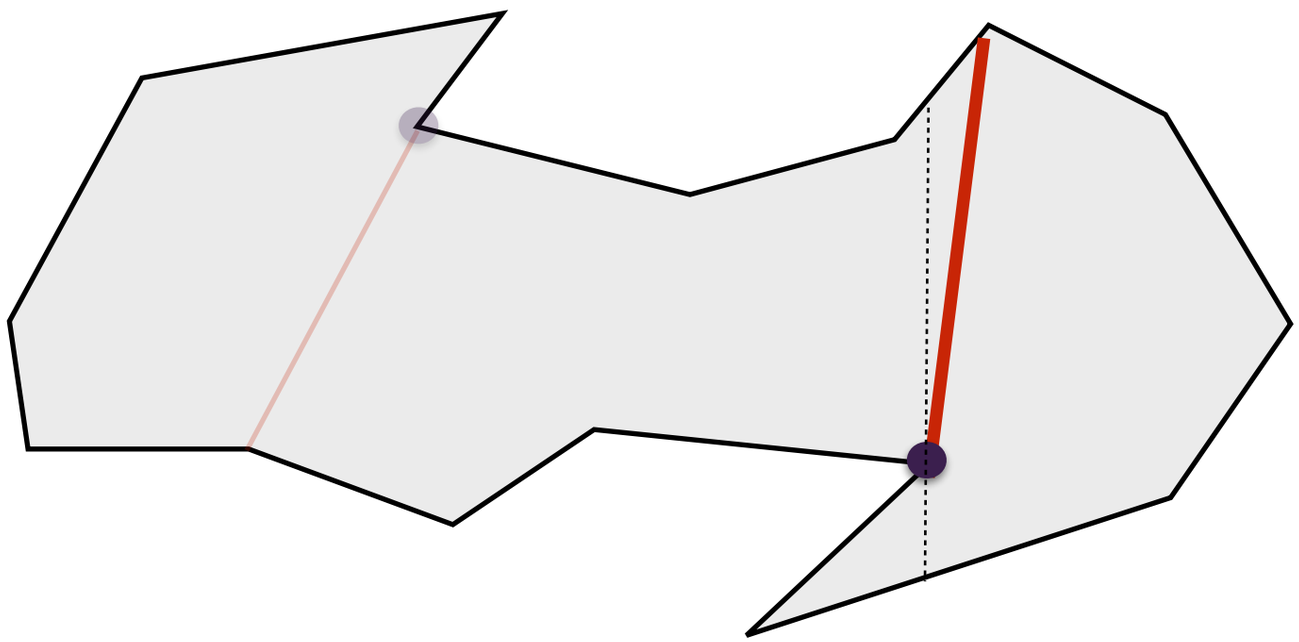
Why are the pieces monotone?

We know: $a_x < \{b_x, d_x\}$
because vertex a is a cusp



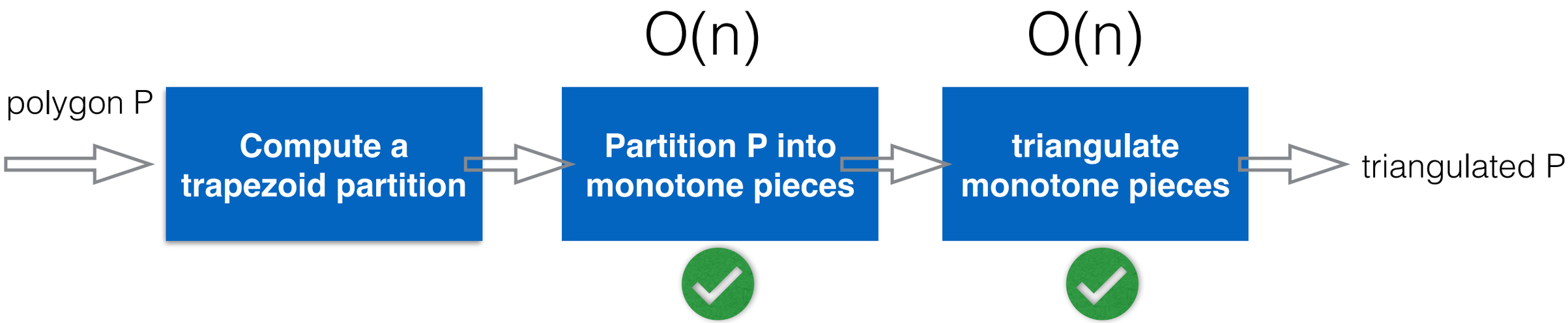
We know: $d_x < a_x$ because ad
is a diagonal in the trapezoid before the cusp

Why does this work?



Where we are:

Given a trapezoid partition of P, we can partition into monotone polygons and triangulate it in O(n) time.



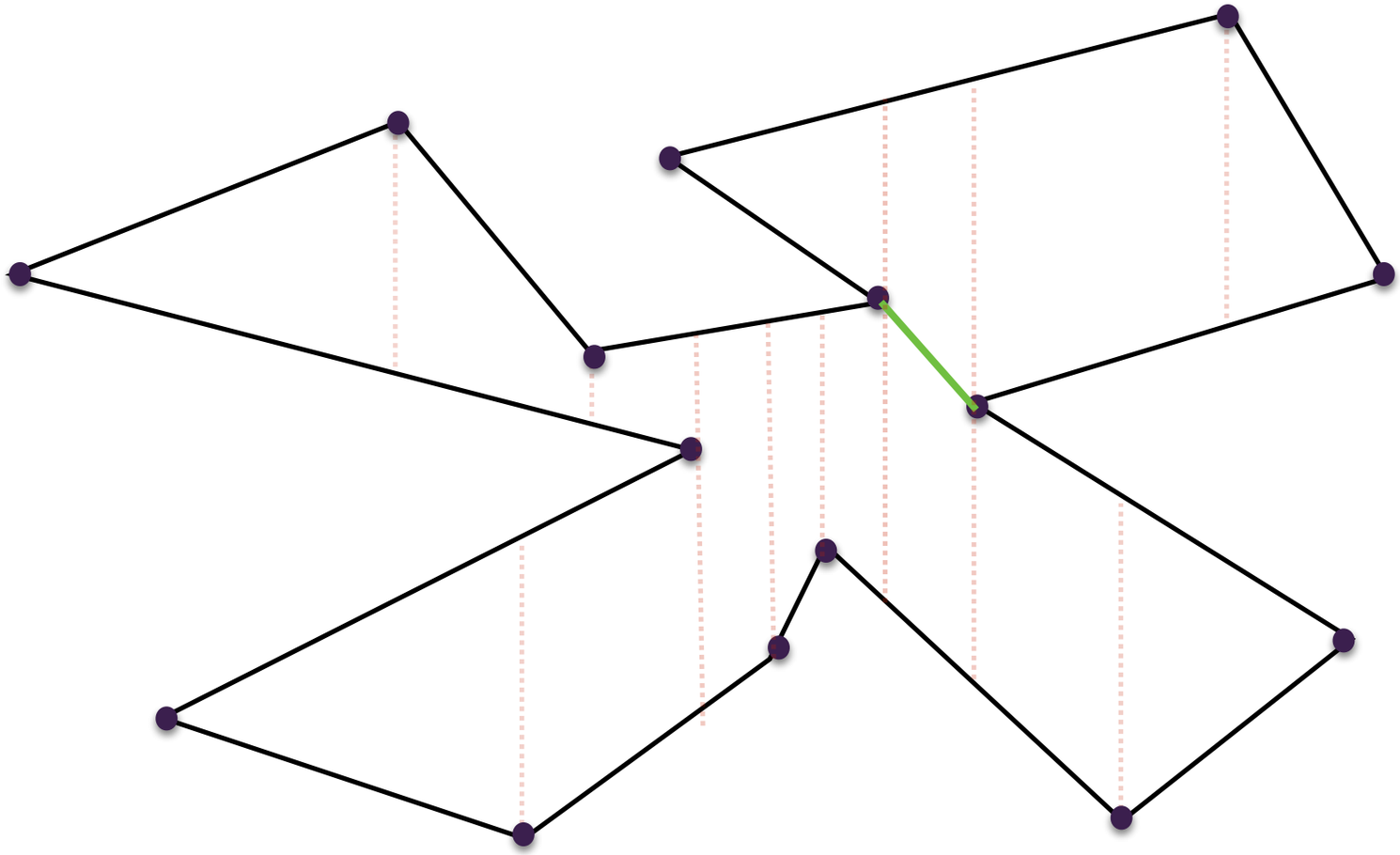
Before we discuss **how** to compute a trapezoid partition of P,

A small detour

- It turns out it is possible (and quite simple!) to use a trapezoid partition to partition P into monotone mountains

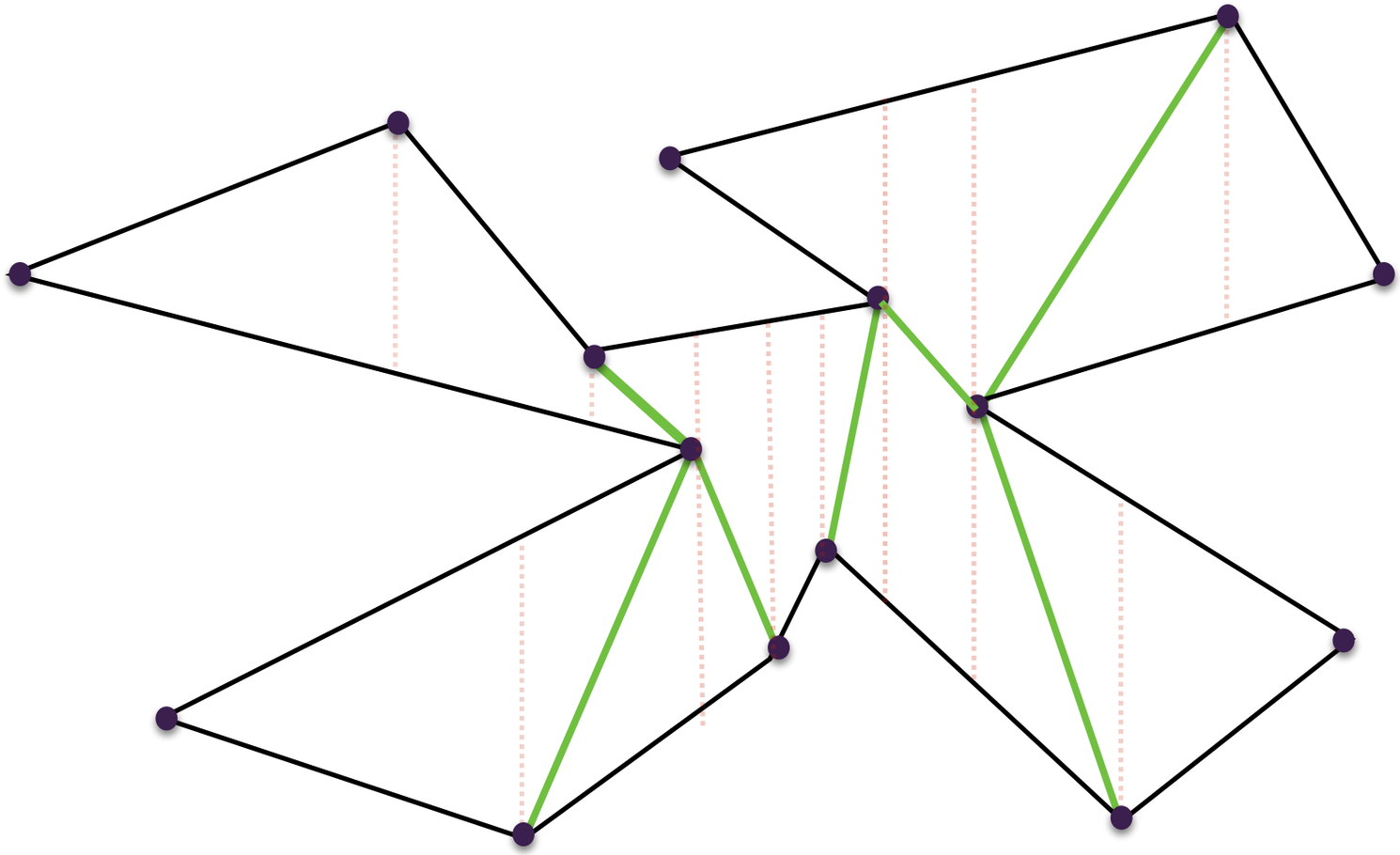
Diagonals

- Given a trapezoid partition of P,
- For each trapezoid, if its two vertices of P are not on same edge, output that as a diagonal
- This gives a new partition of P. Can we say anything about these pieces?



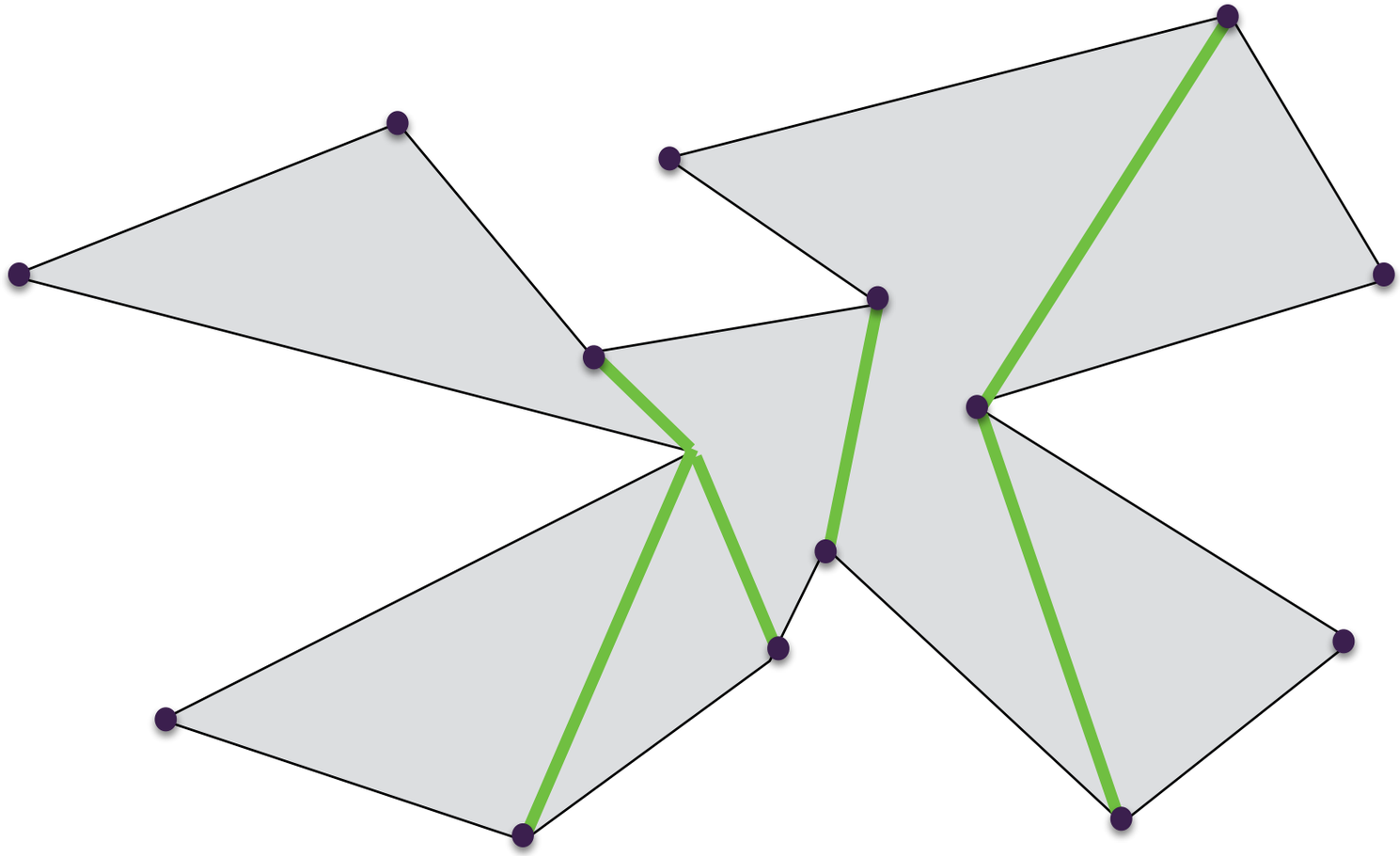
Diagonals

- Given a trapezoid partition of P,
- For each trapezoid, if its two vertices of P are not on same edge, output that as a diagonal
- This gives a new partition of P. Can we say anything about these pieces?

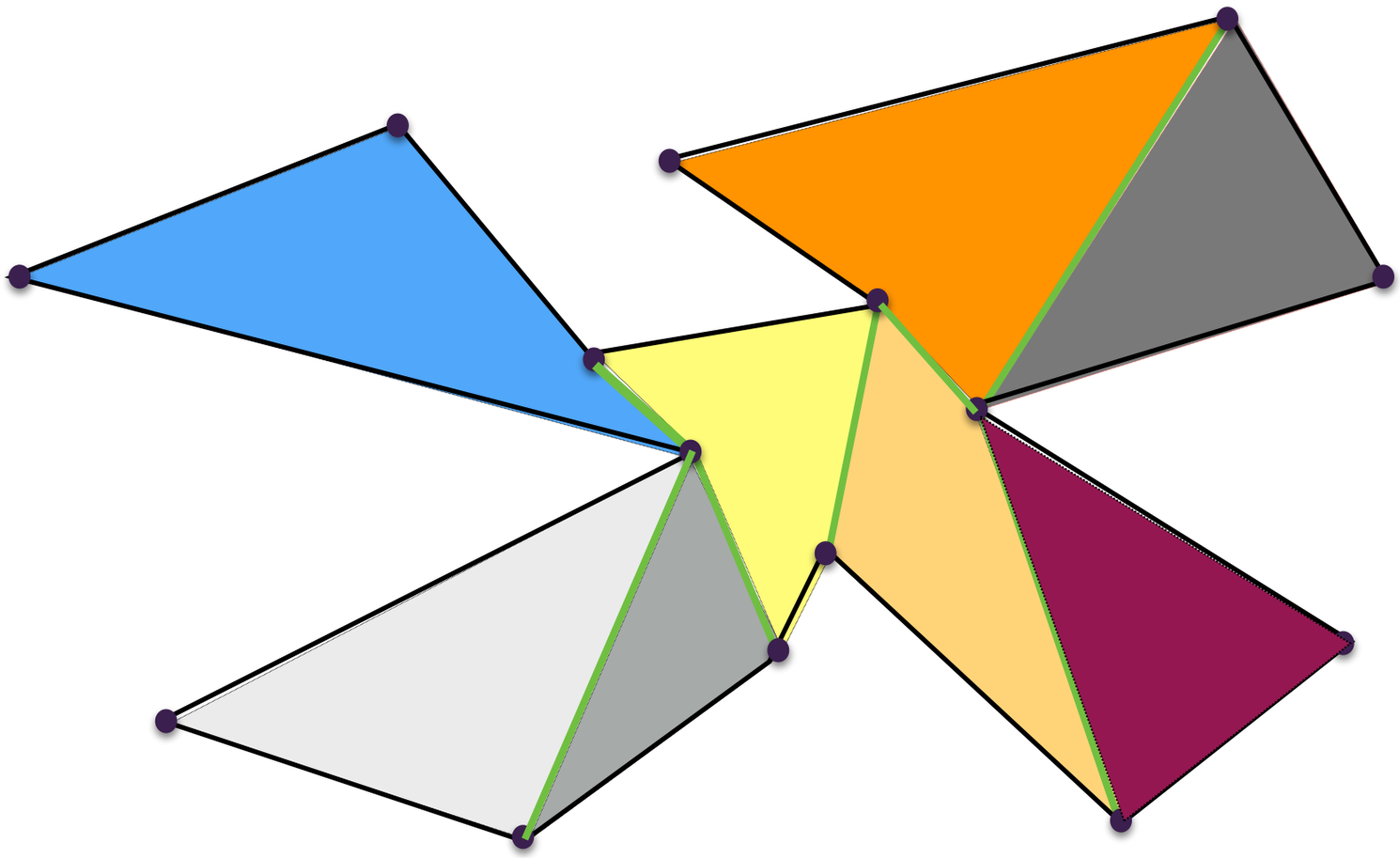


Diagonals

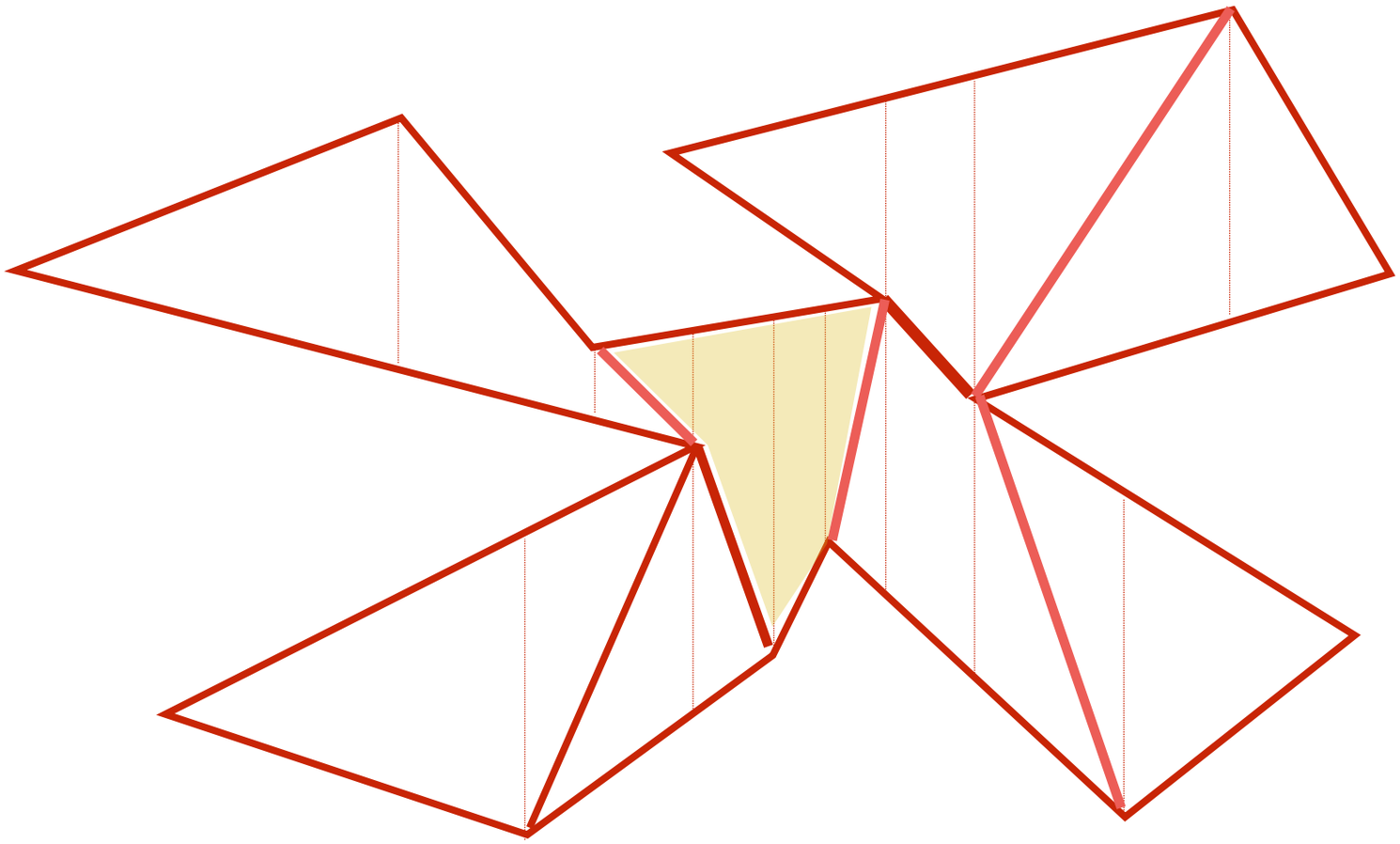
- Given a trapezoid partition of P,
- For each trapezoid, if its two vertices of P are not on same edge, output that as a diagonal
- This gives a new partition of P. Can we say anything about these pieces?



Claim: **All** diagonals partition the polygon into monotone mountains.



Claim: **All** diagonals partition the polygon into monotone mountains.

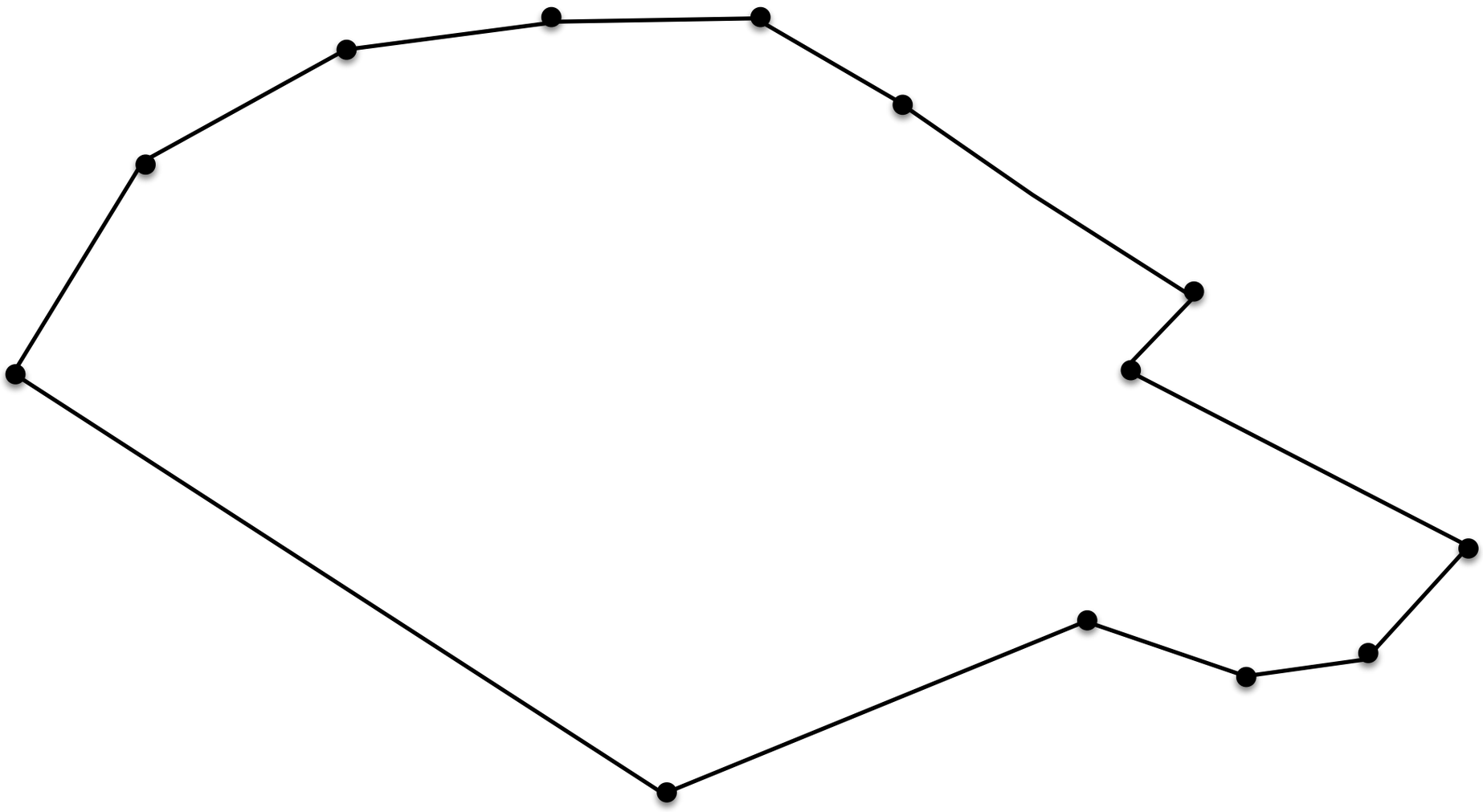


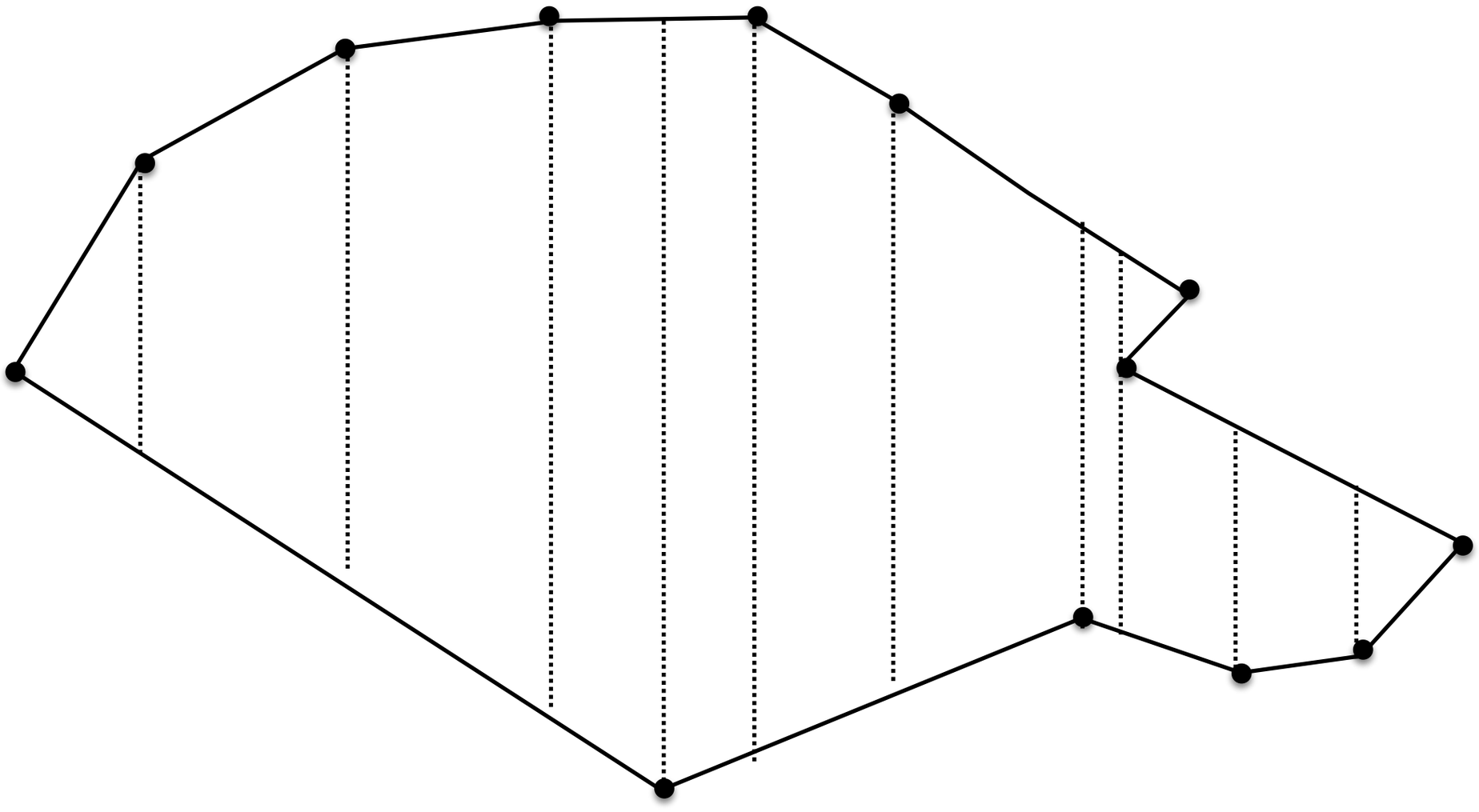
Why?

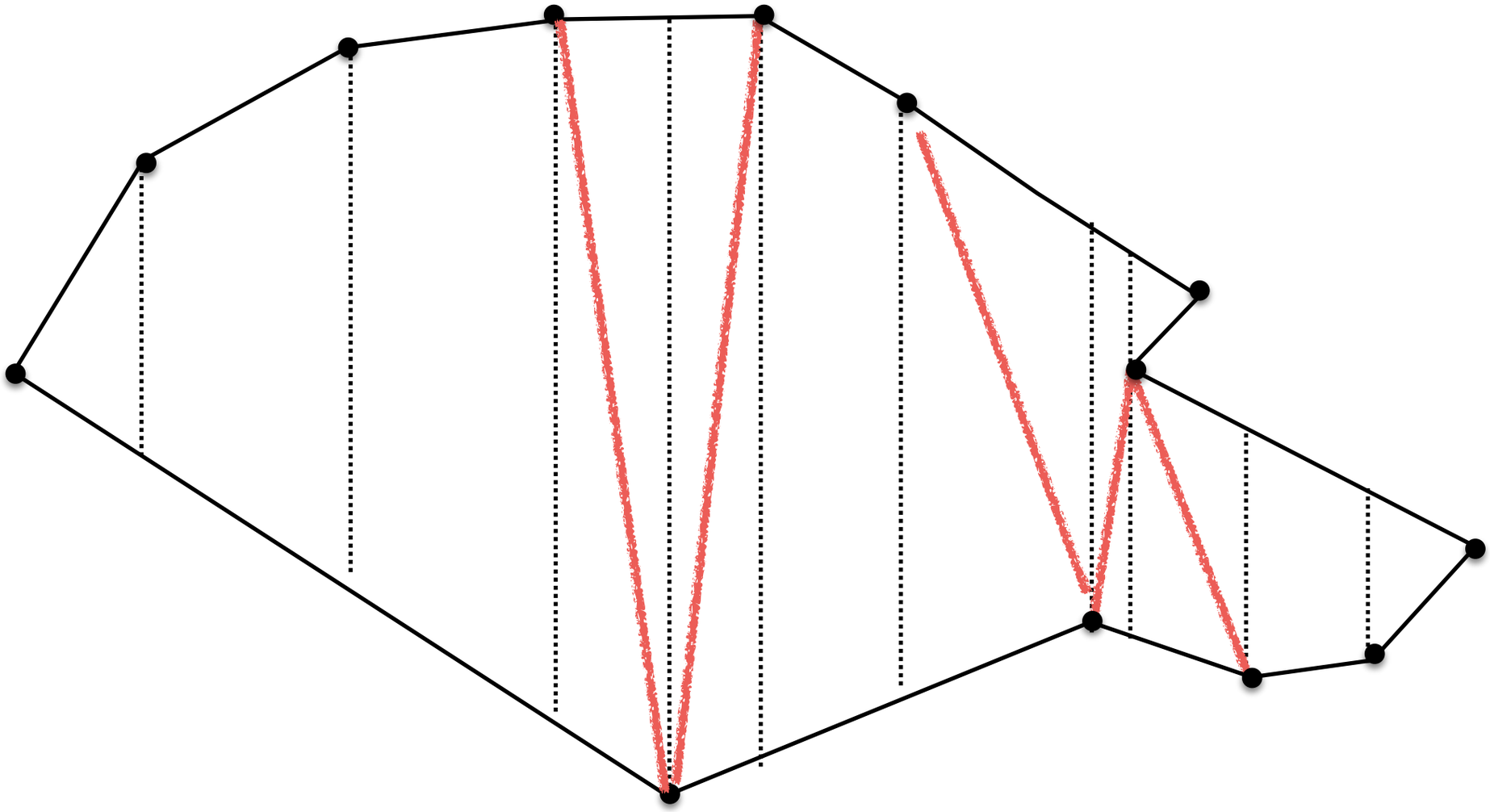
- Each “internal” trapezoid in a polygon must have the polygon vertices either both above (or both below), otherwise they would generate a diagonal => all internal trapezoids have same edge below/above => one edge

Another example

Compute a trapezoid partition and show the diagonals

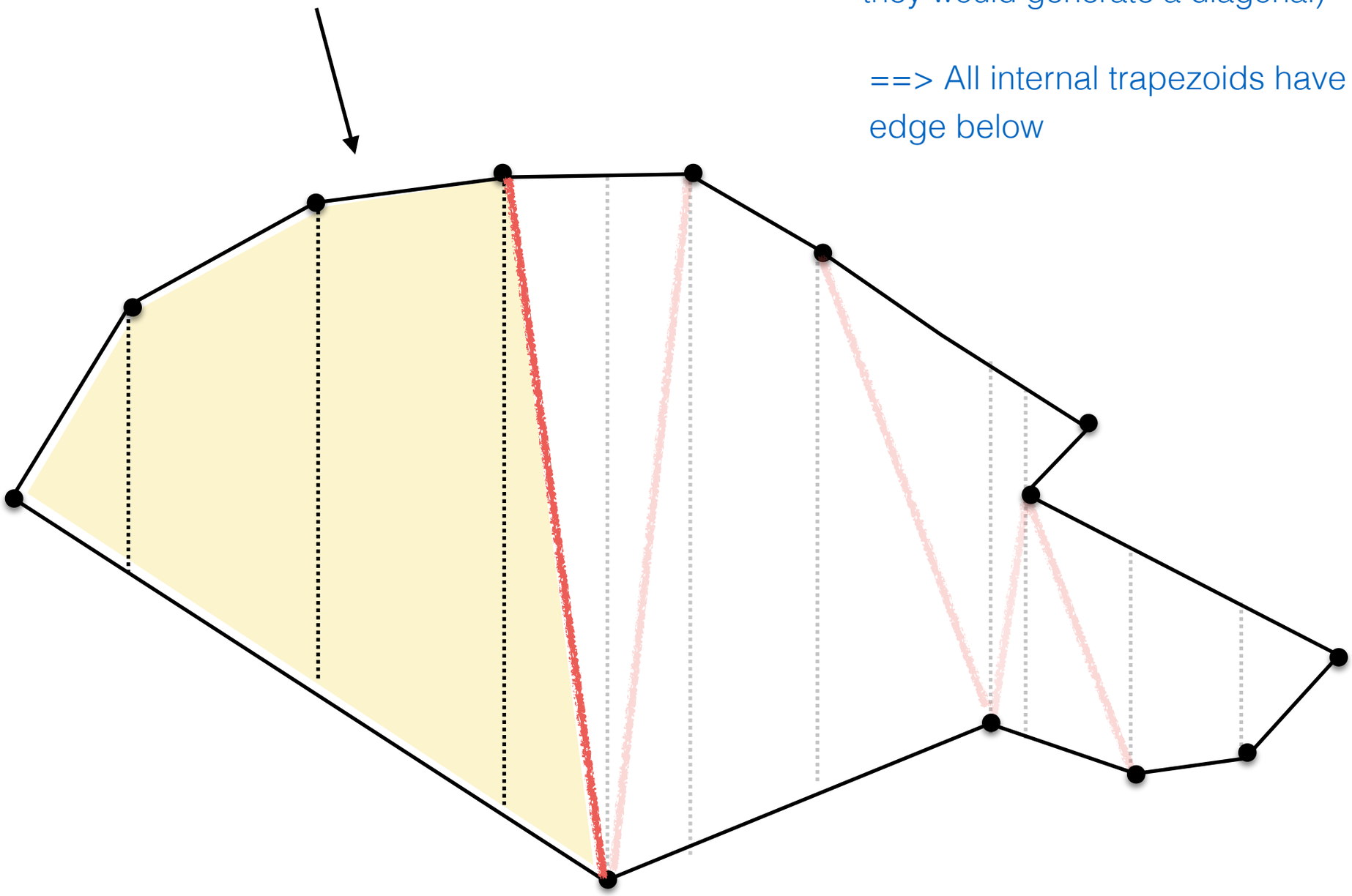






Let’s look at one of these pieces

There is no diagonal in here

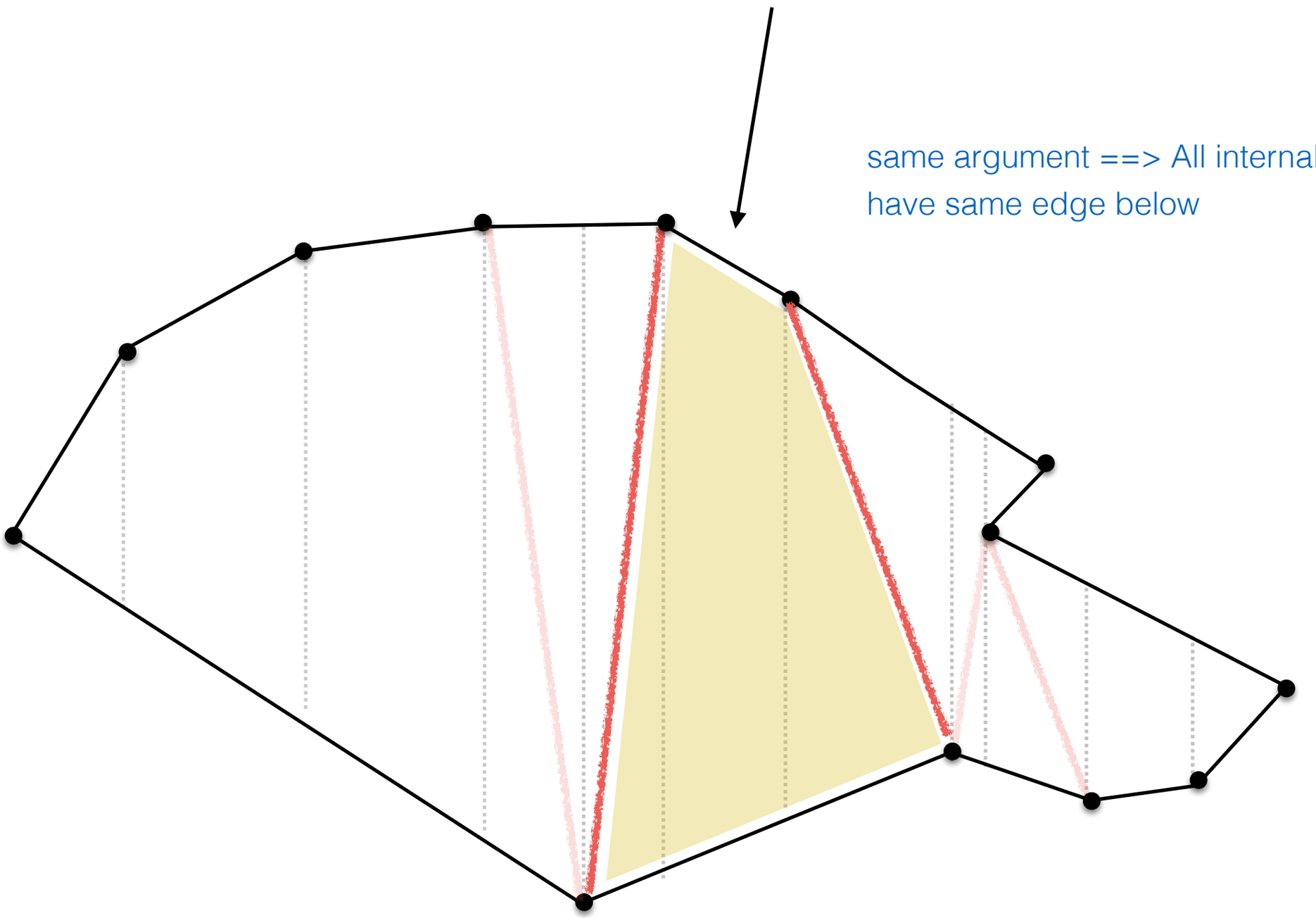


Each trapezoid must have the polygon vertices both above or both below (otherwise they would generate a diagonal)

==> All internal trapezoids have same edge below

There is no diagonal in here

same argument ==> All internal trapezoids have same edge below

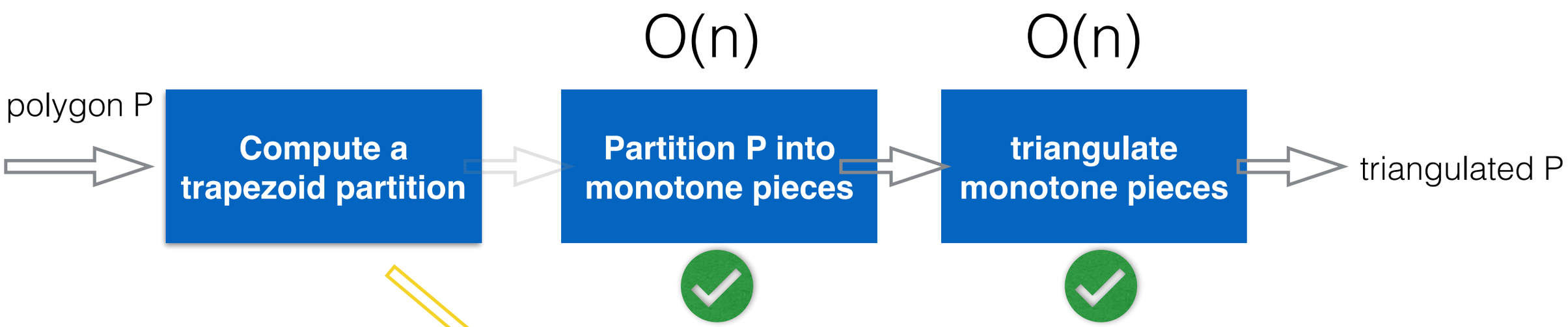


Every piece is a monotone mountain!

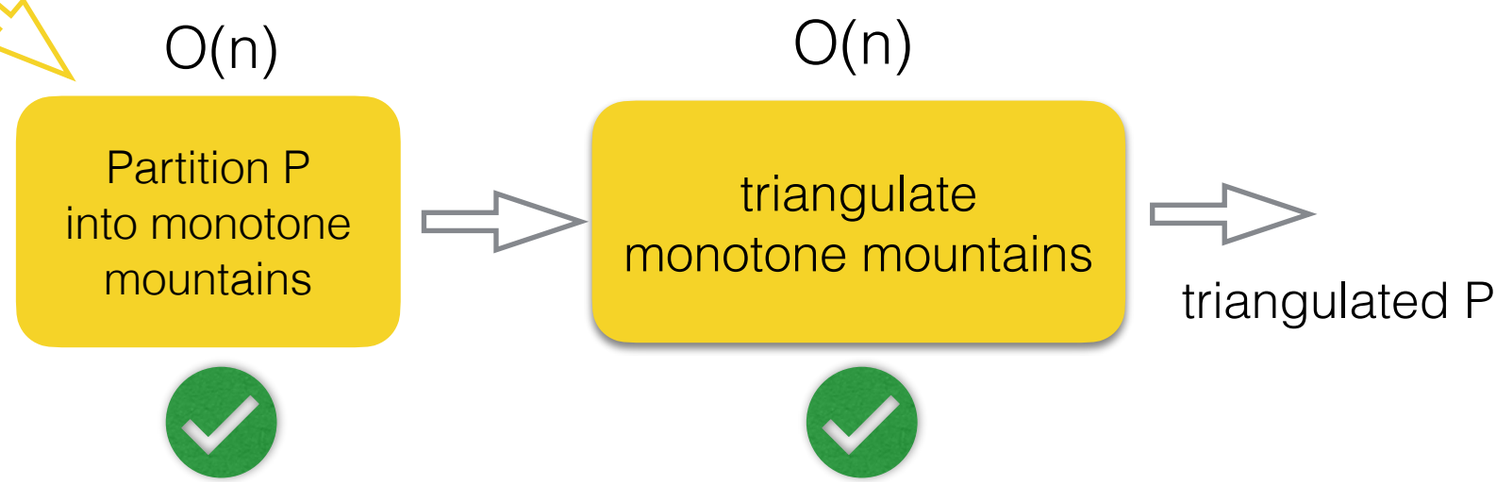
End detour

Where we are:

Given a trapezoid partition of P, we can partition into monotone polygons and triangulate it in O(n) time.



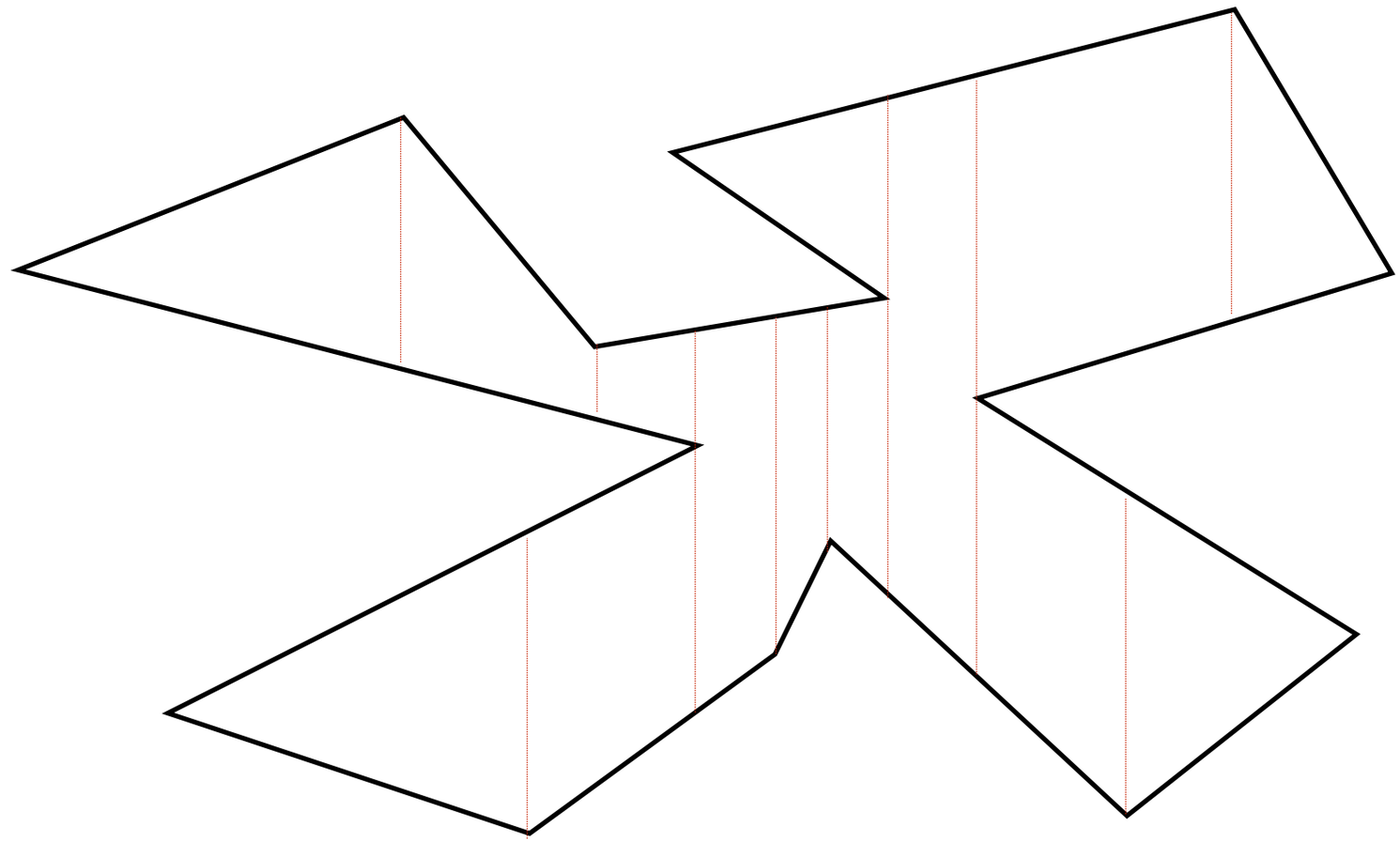
In fact, a trapezoid partition can give a partition into monotone mountains



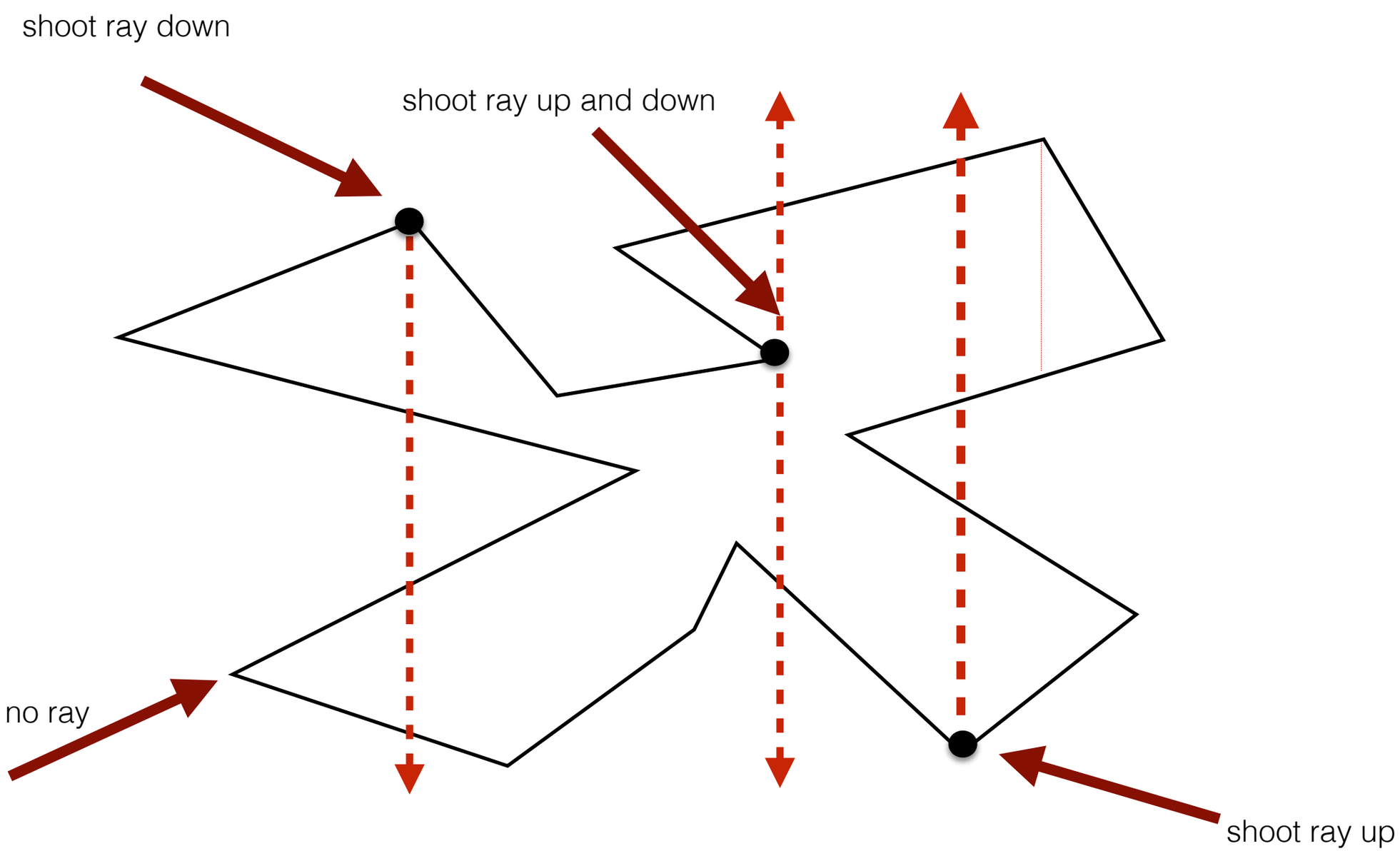
The missing piece:

How to compute a trapezoid partition?

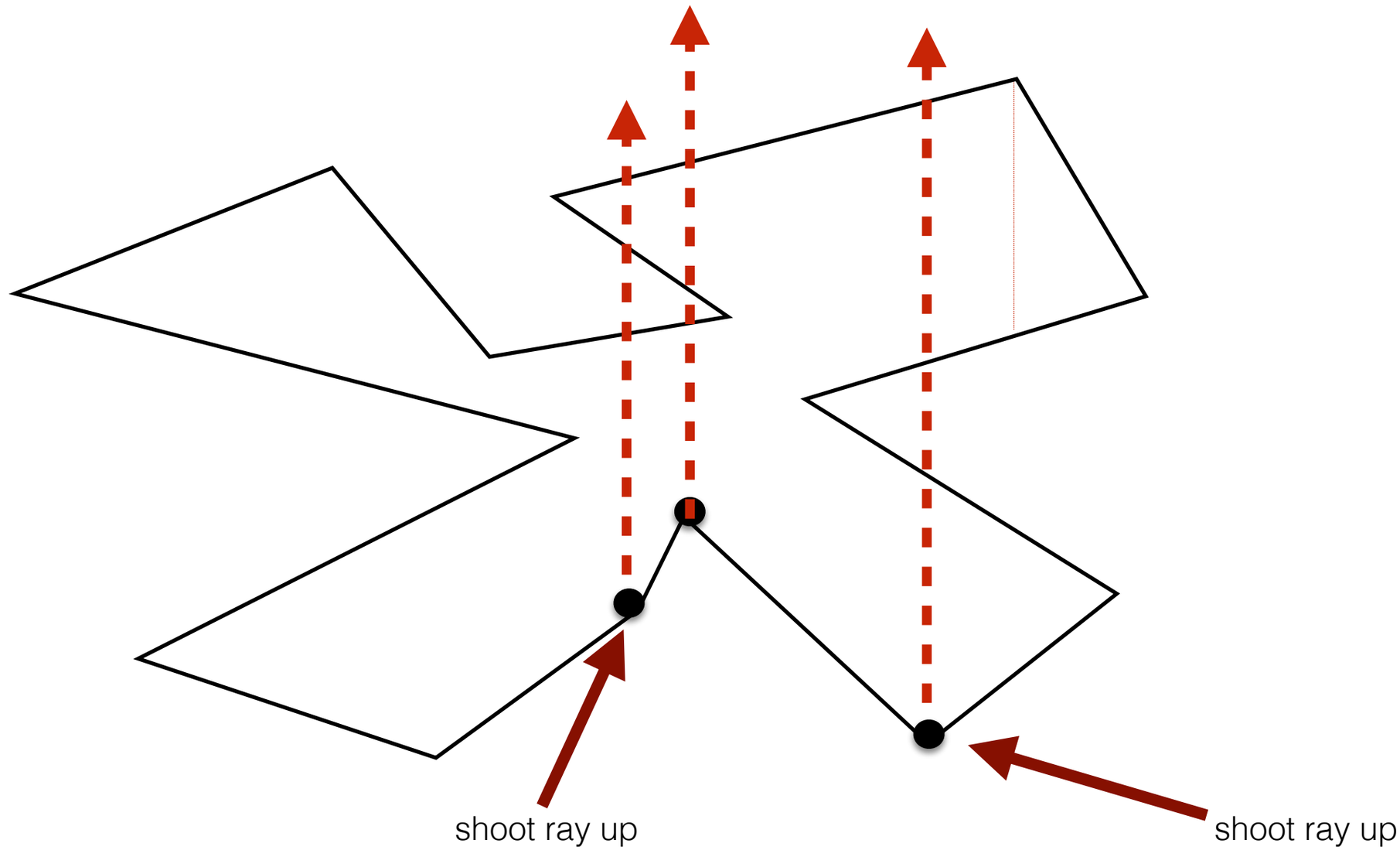
Computing the trapezoid partition



Naive algorithm?

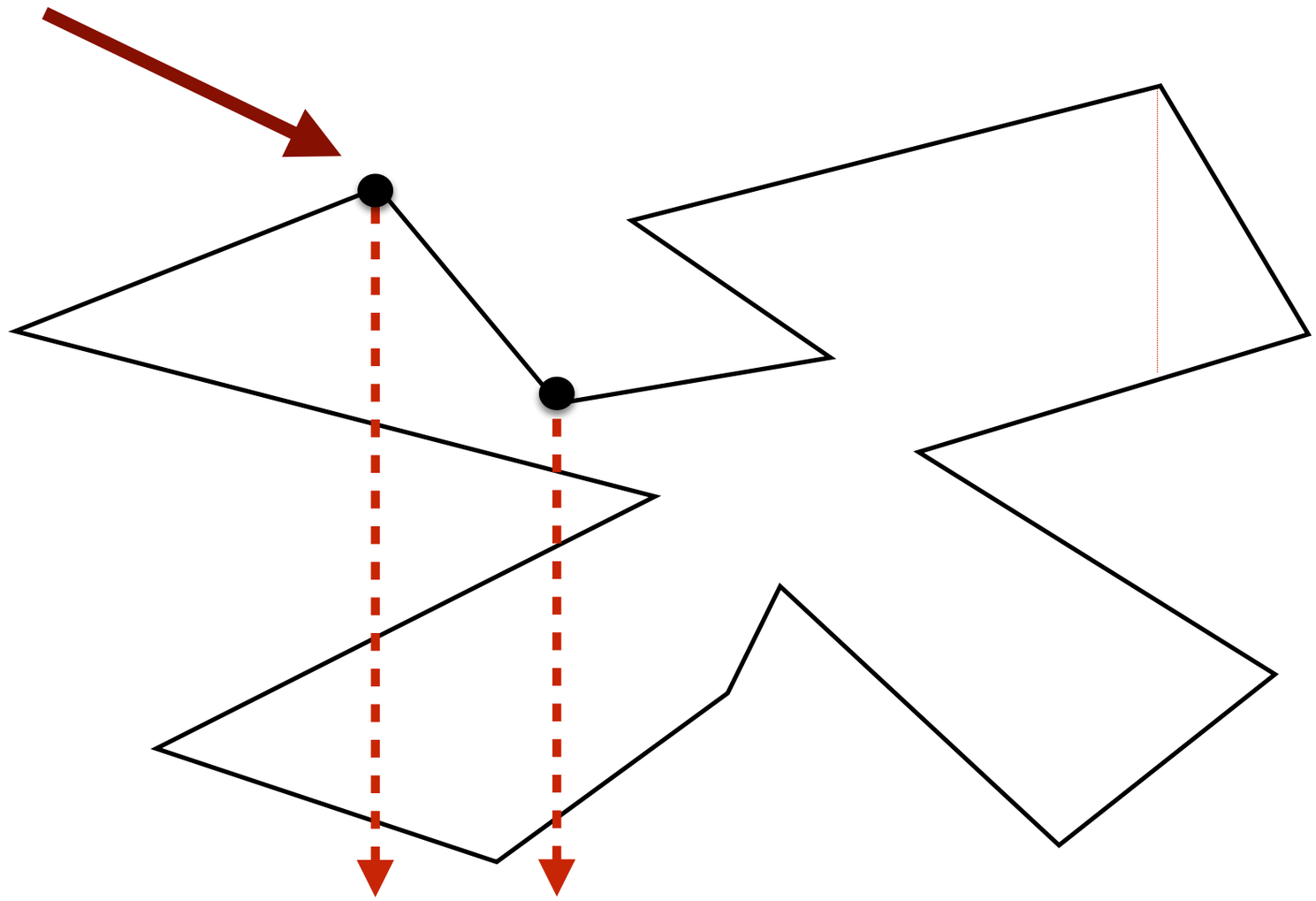


Naive algorithm: for each vertex, shoot a vertical ray through it and compute the “closest” edge(s) that intersects it… [Note: This part only takes $O(n^2)$]

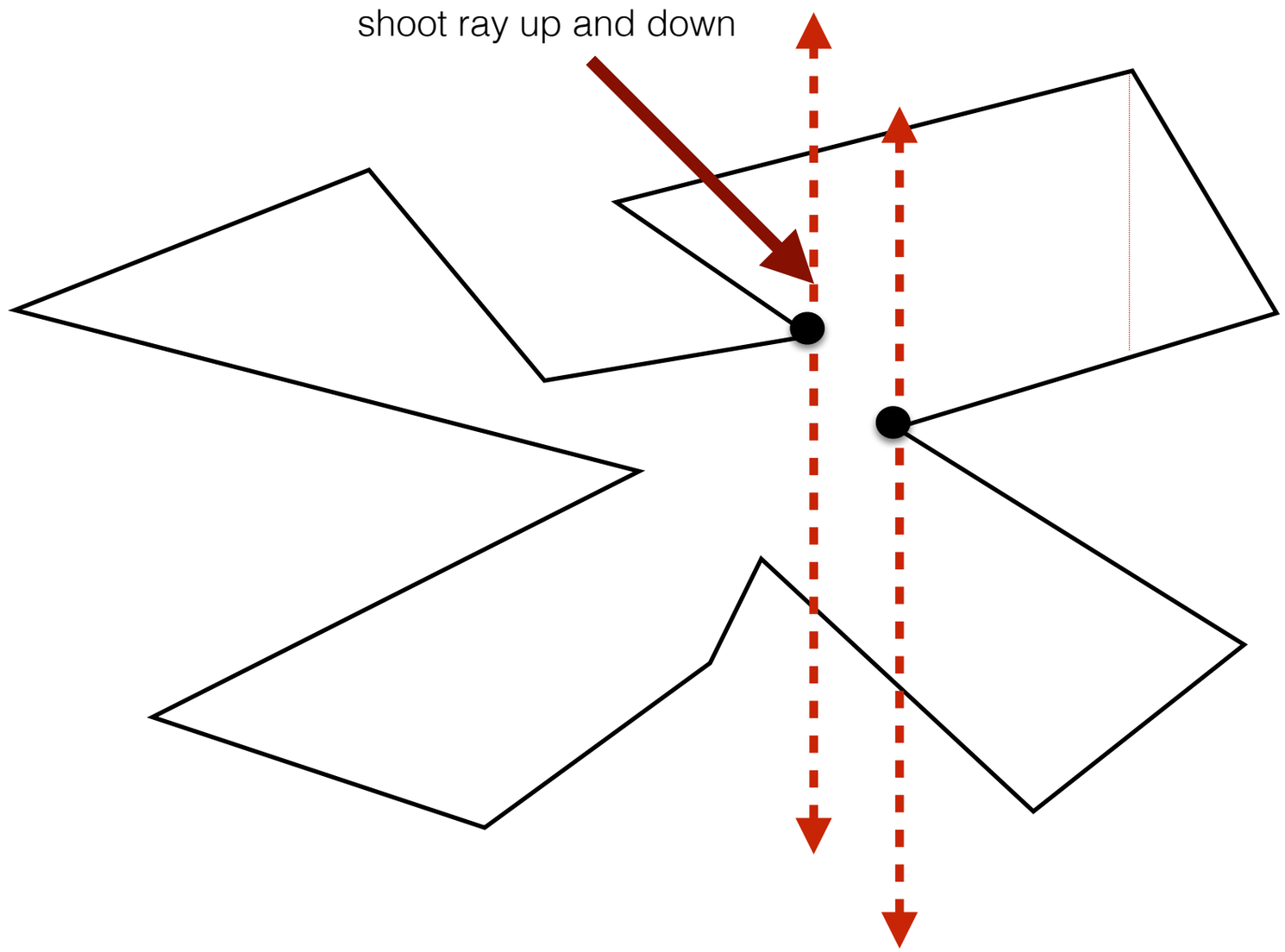


Some cases (not all). How to distinguish between these cases?

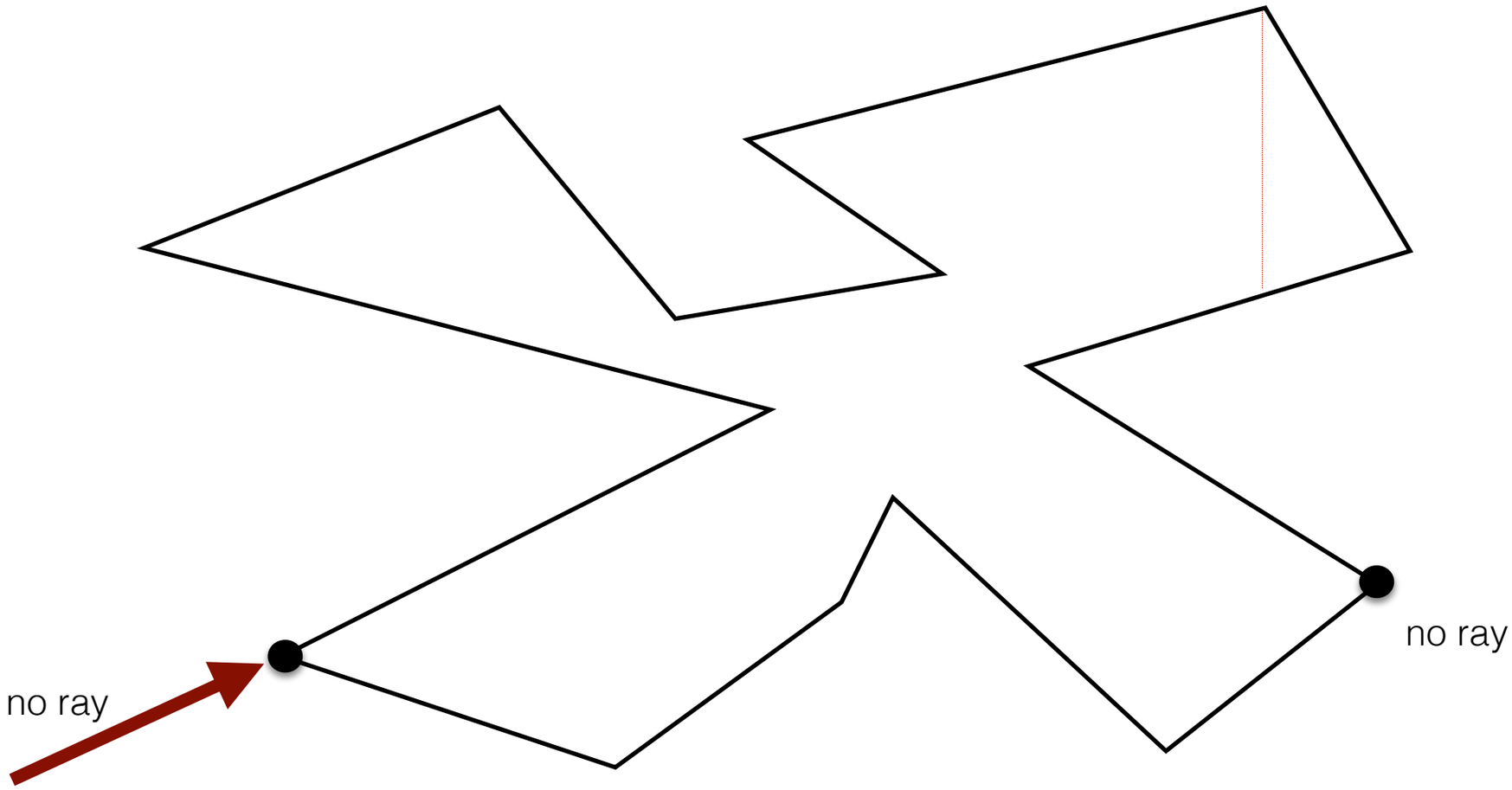
shoot ray down



Some cases (not all). How to distinguish between these cases?



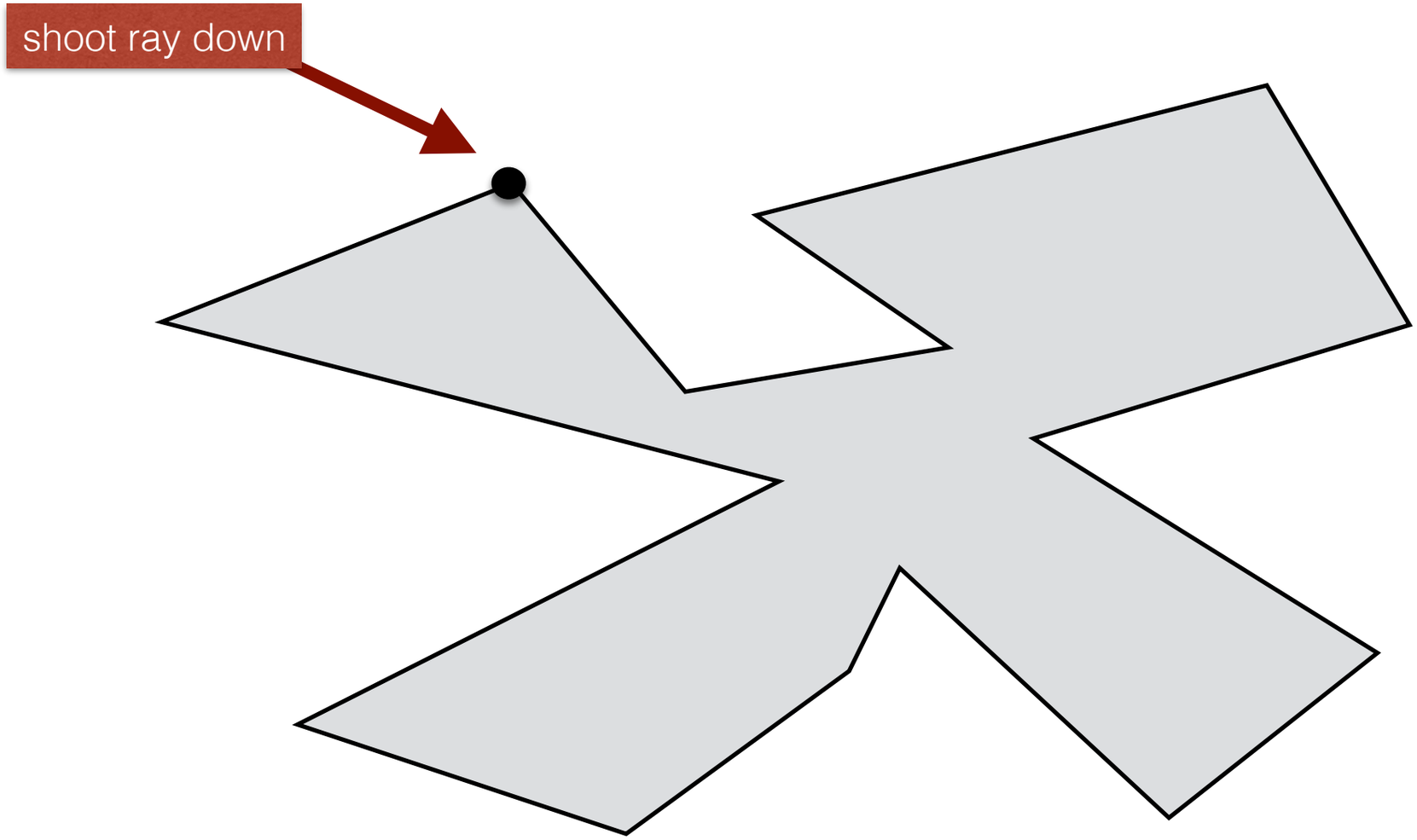
Some cases (not all). How to distinguish between these cases?



Some cases (not all). How to distinguish between these cases?

v convex if LeftOf v^-, v, v^+

v is convex
and
v^-, v^+ are both below v_y

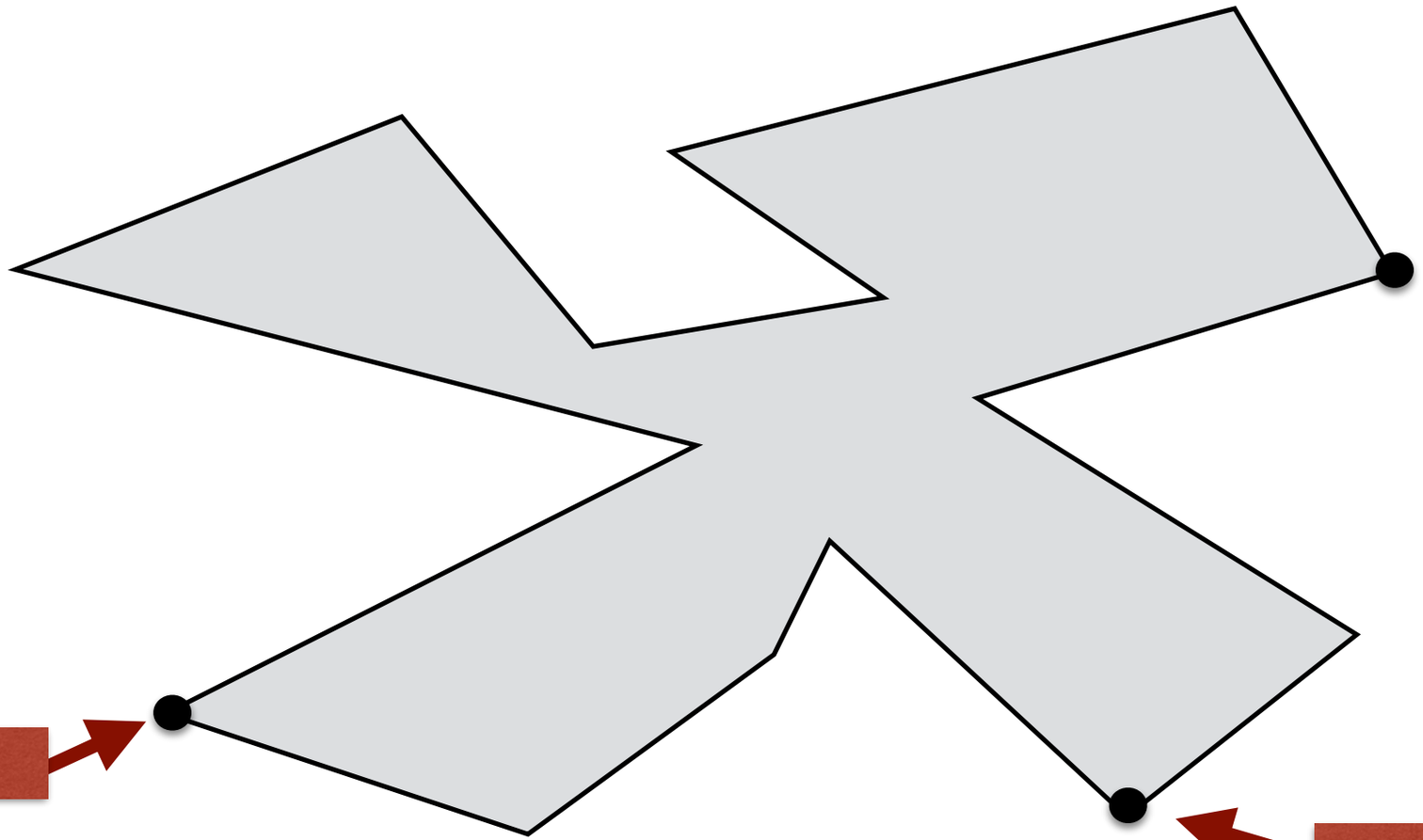


Assume P is given so that ∂P is ccw

Fact: The polygon is to the left of every edge.

We can figure out which case we are in …

v convex if LeftOf v^-, v, v^+



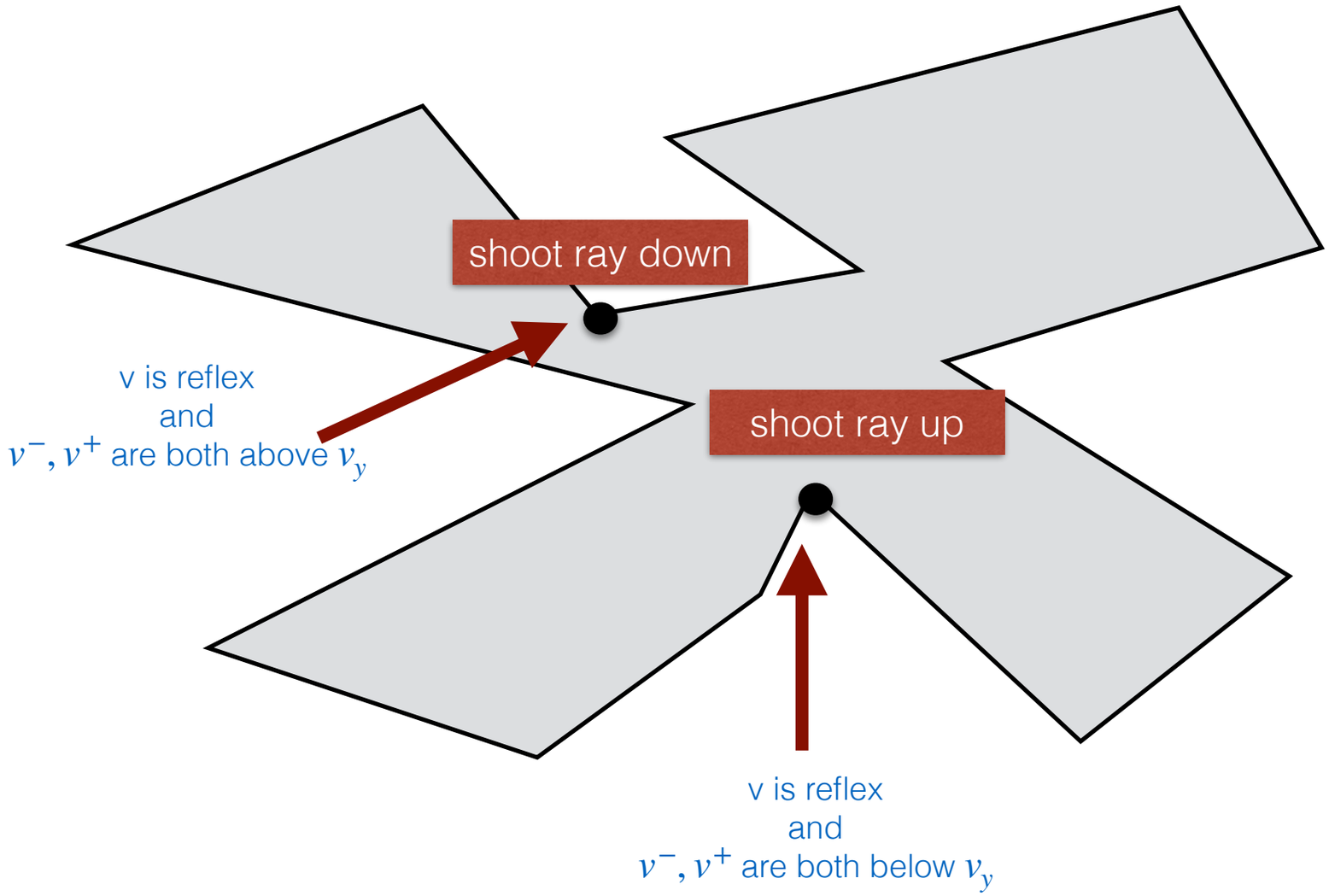
no ray

v is convex
and
v^- and v^+ are to the right of v_x

no ray

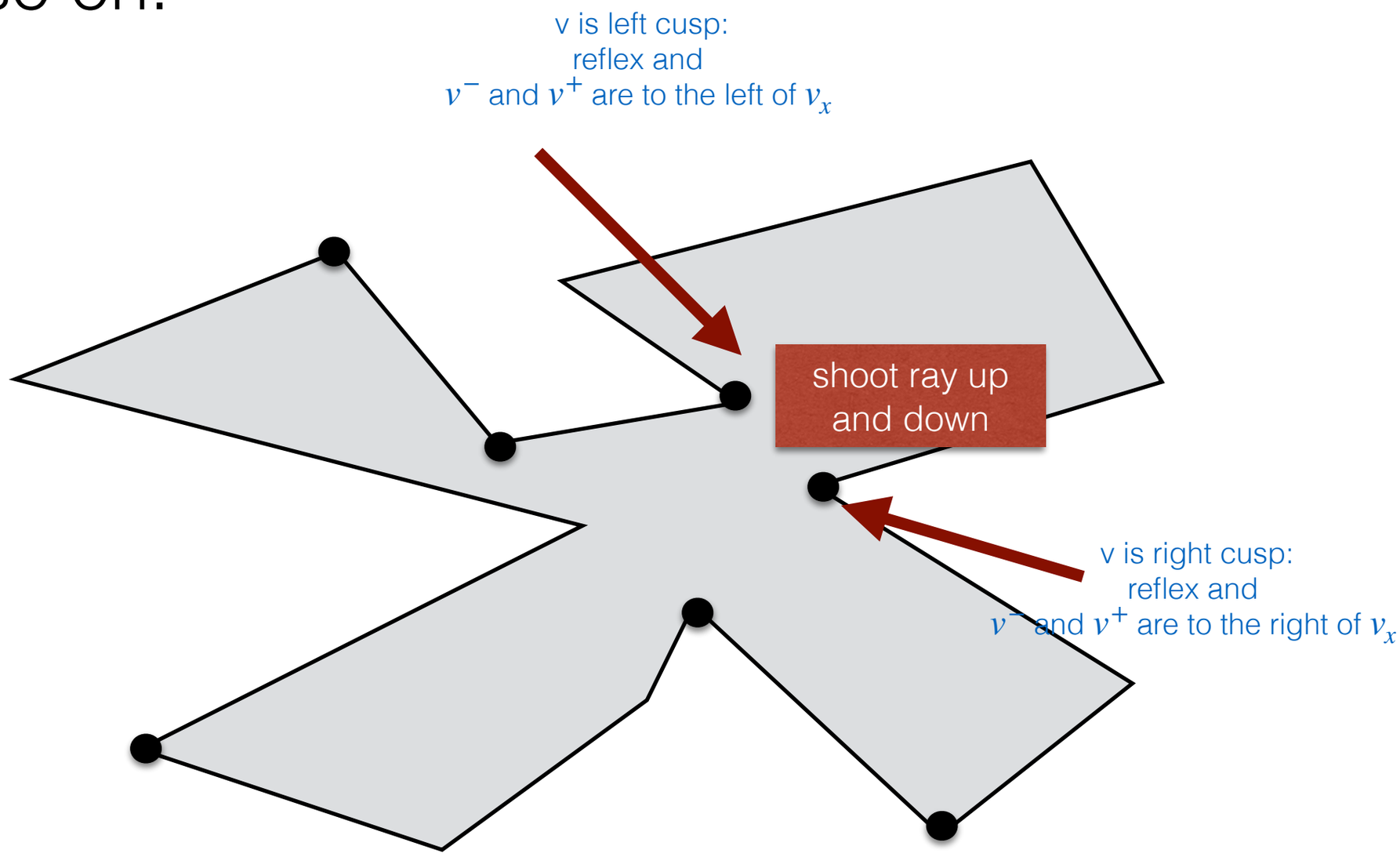
v is convex
and
v^-, v^+ are both above v_y

v convex if LeftOf v^-, v, v^+



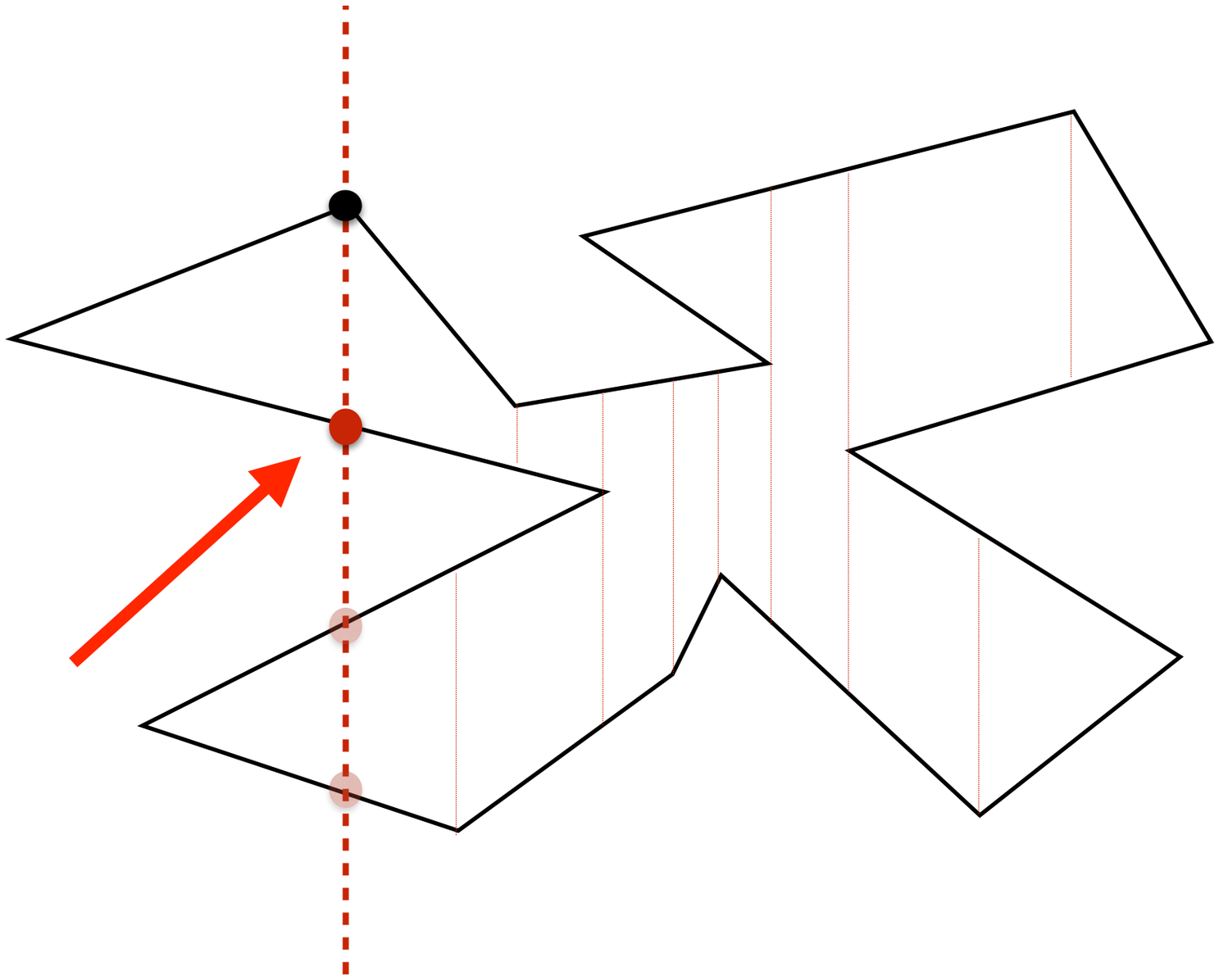
v convex if LeftOf v^-, v, v^+

And so on.



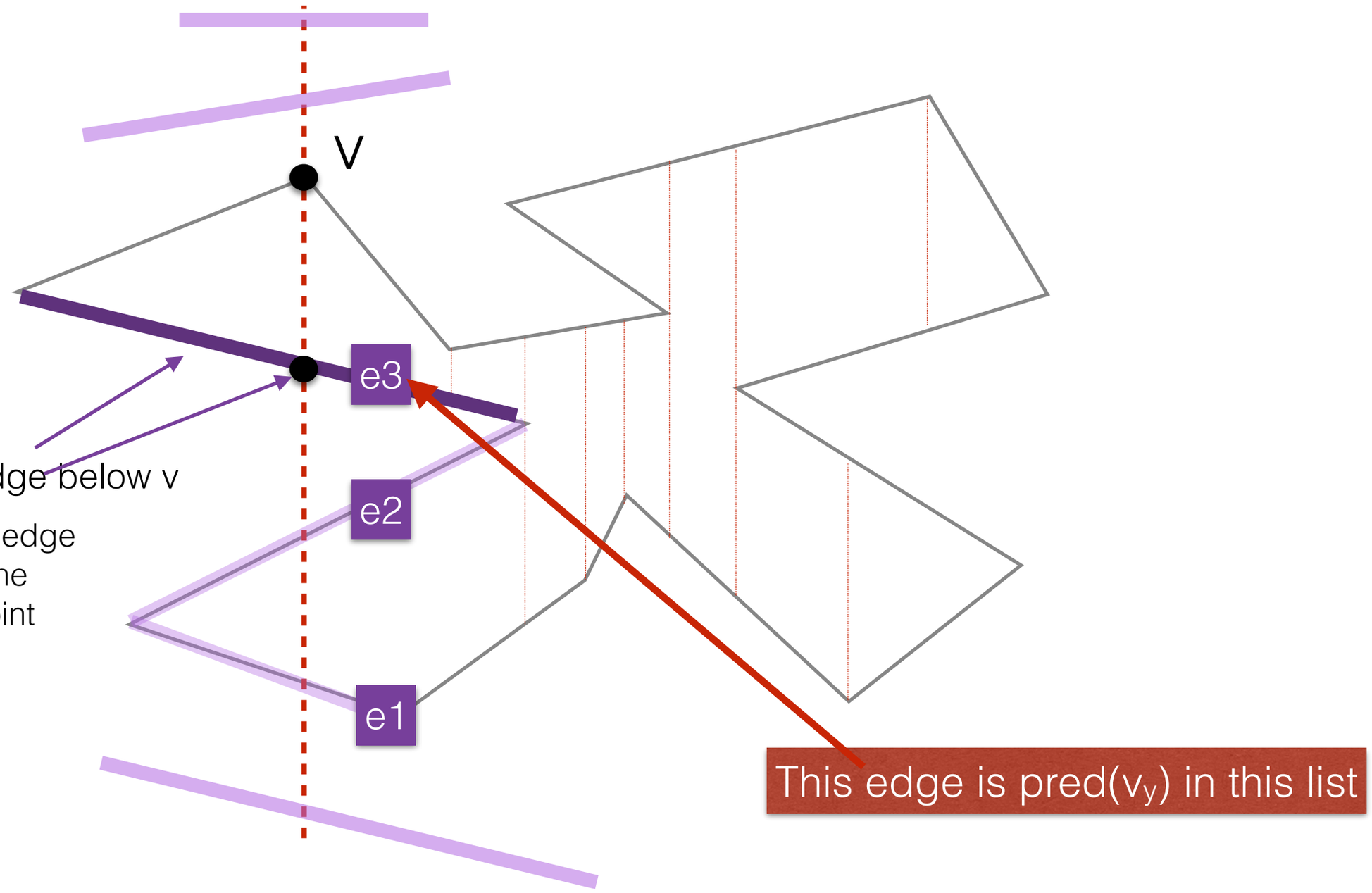
Summary: these cases can be figured out.

Back to computing the trapezoid partition



Naive algorithm: for each vertex, shoot a vertical ray through it and compute the “closest” edge that intersects it… [Note: This part only takes $O(n^2)$]

What if all the edges that intersect the ray from v sit in a nice structure in y-order?



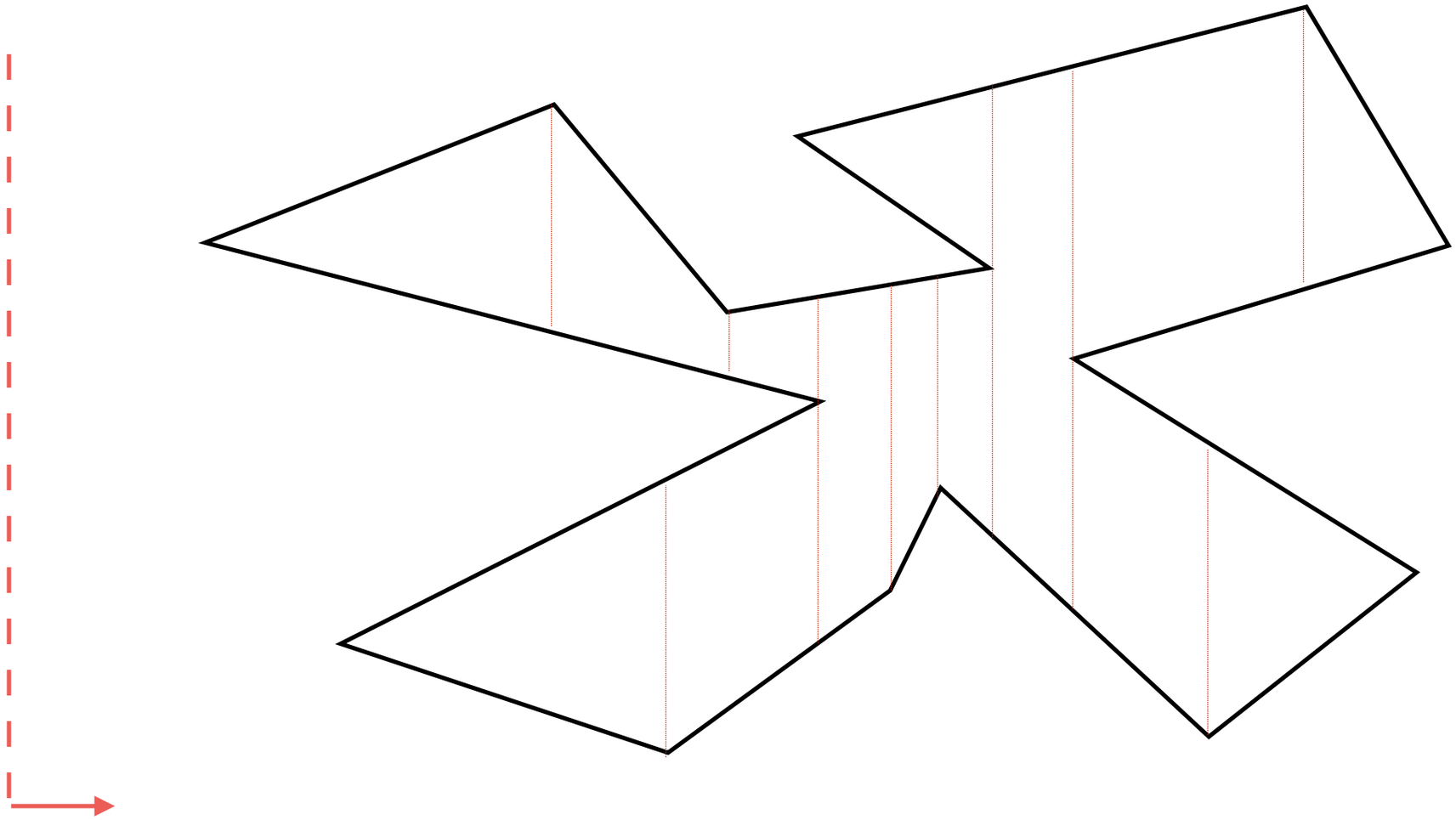
we want the first edge below v

If we can get this edge
=> compute the
intersection point

….< e1 < e2 < e3 < …

Computing the trapezoid partition in O(n lg n)

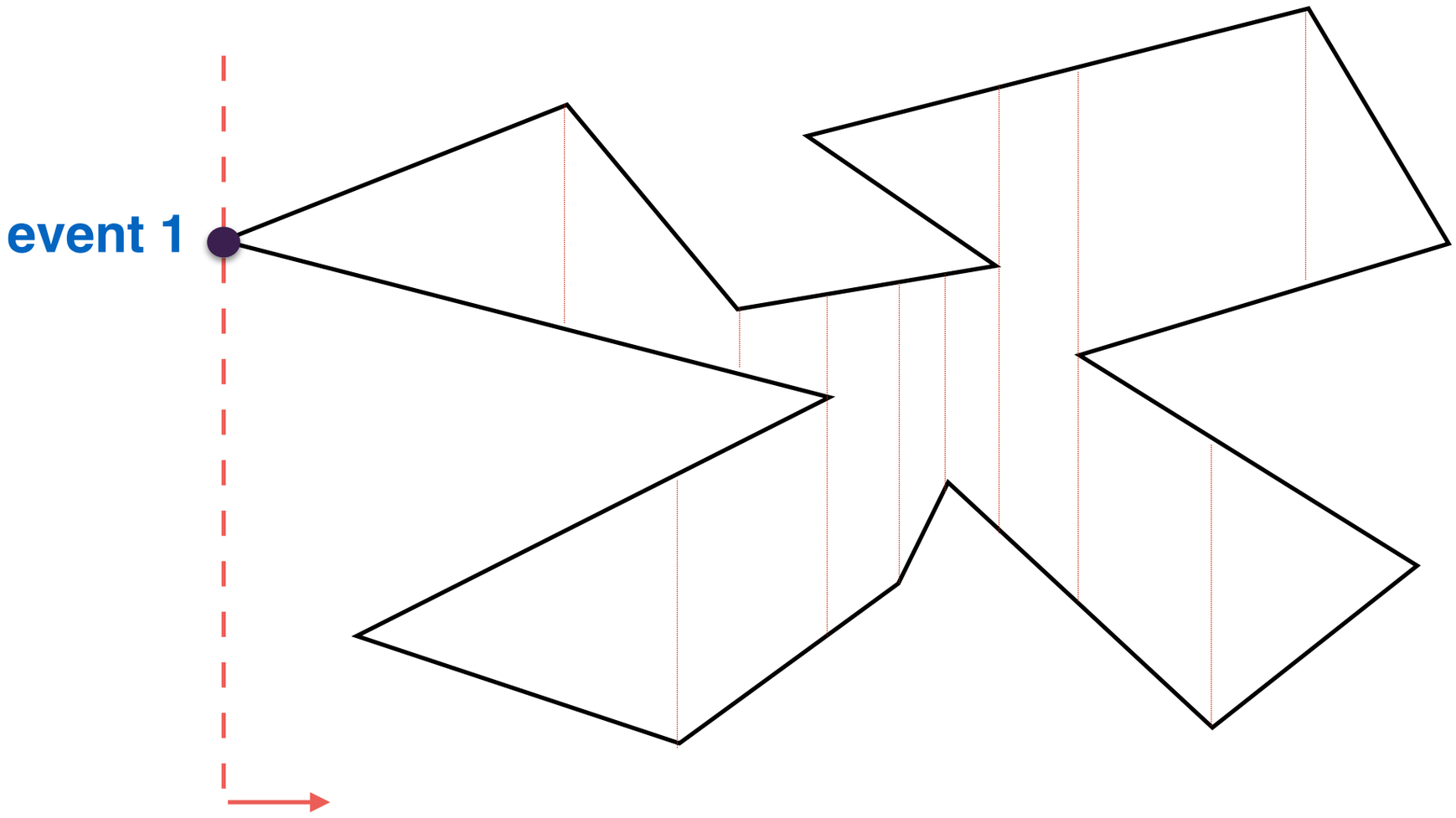
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

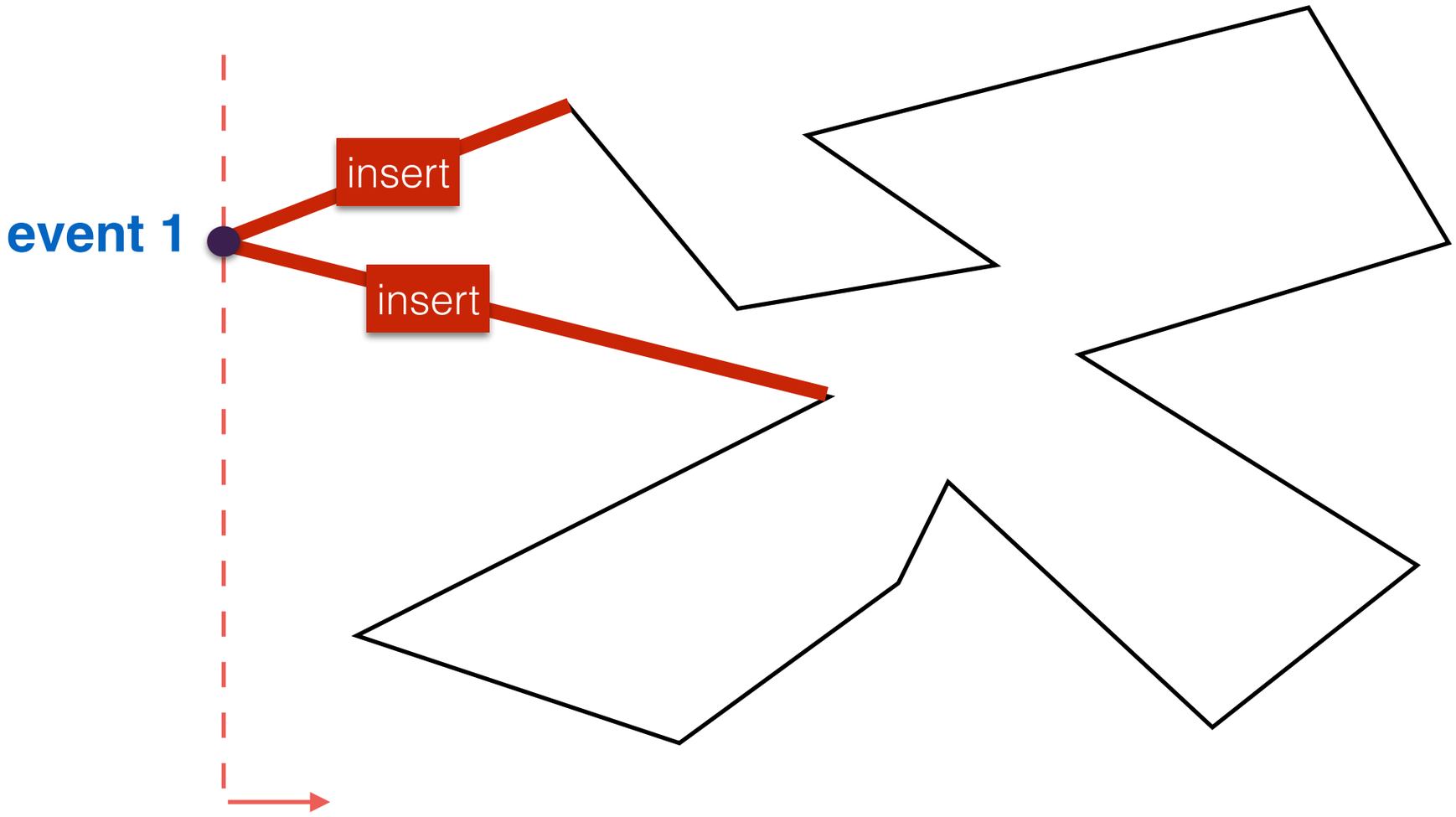
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

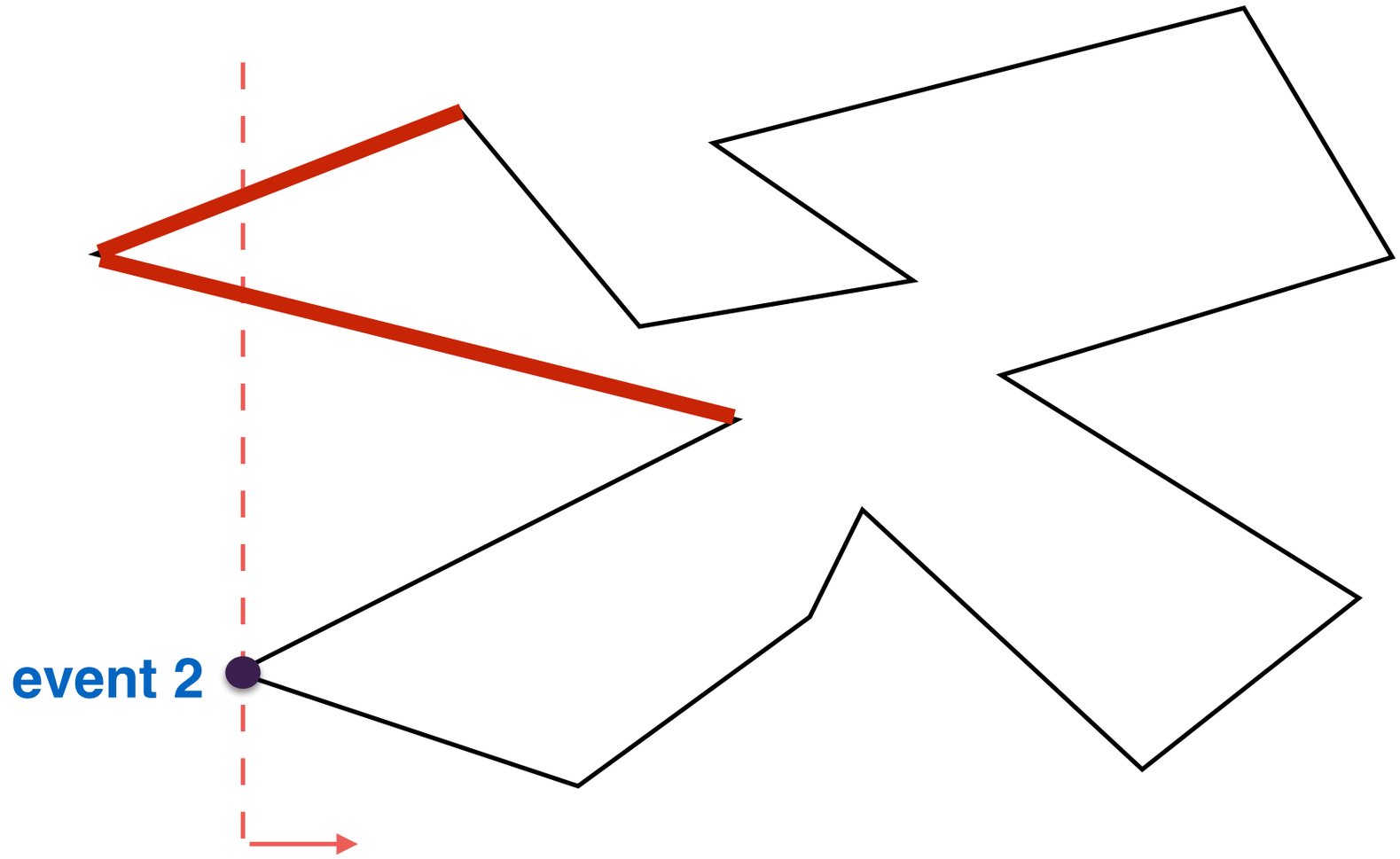
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

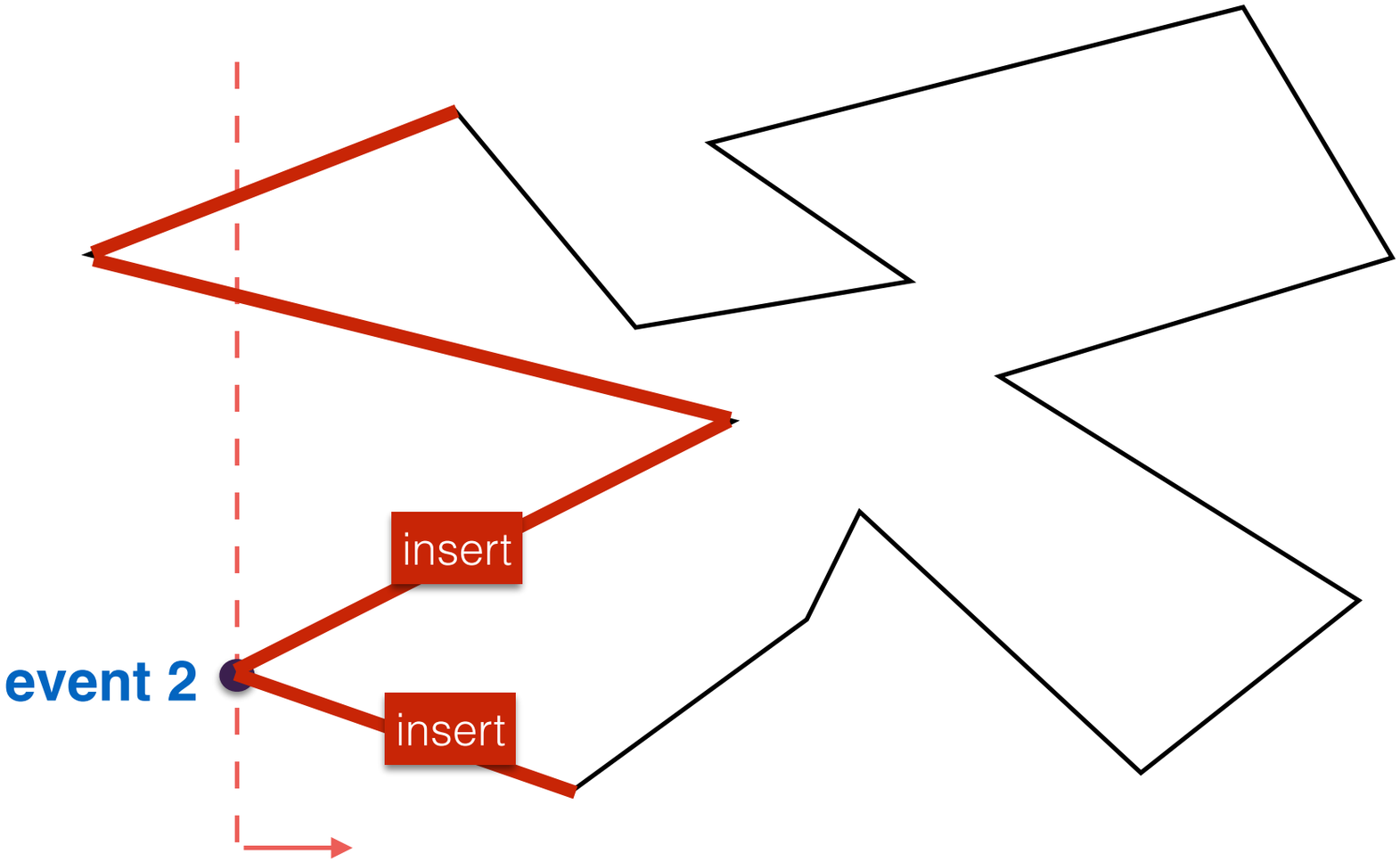
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

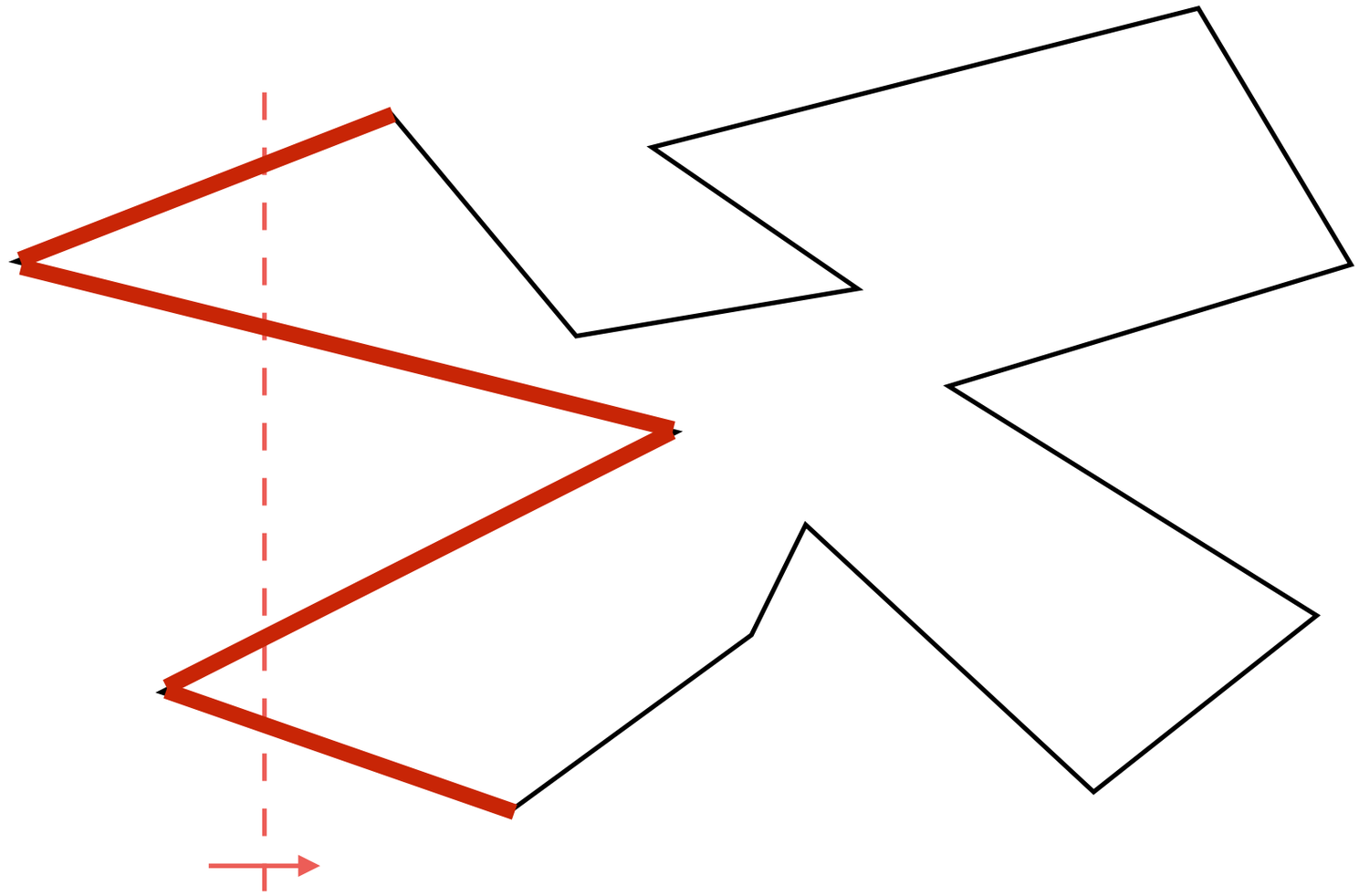
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

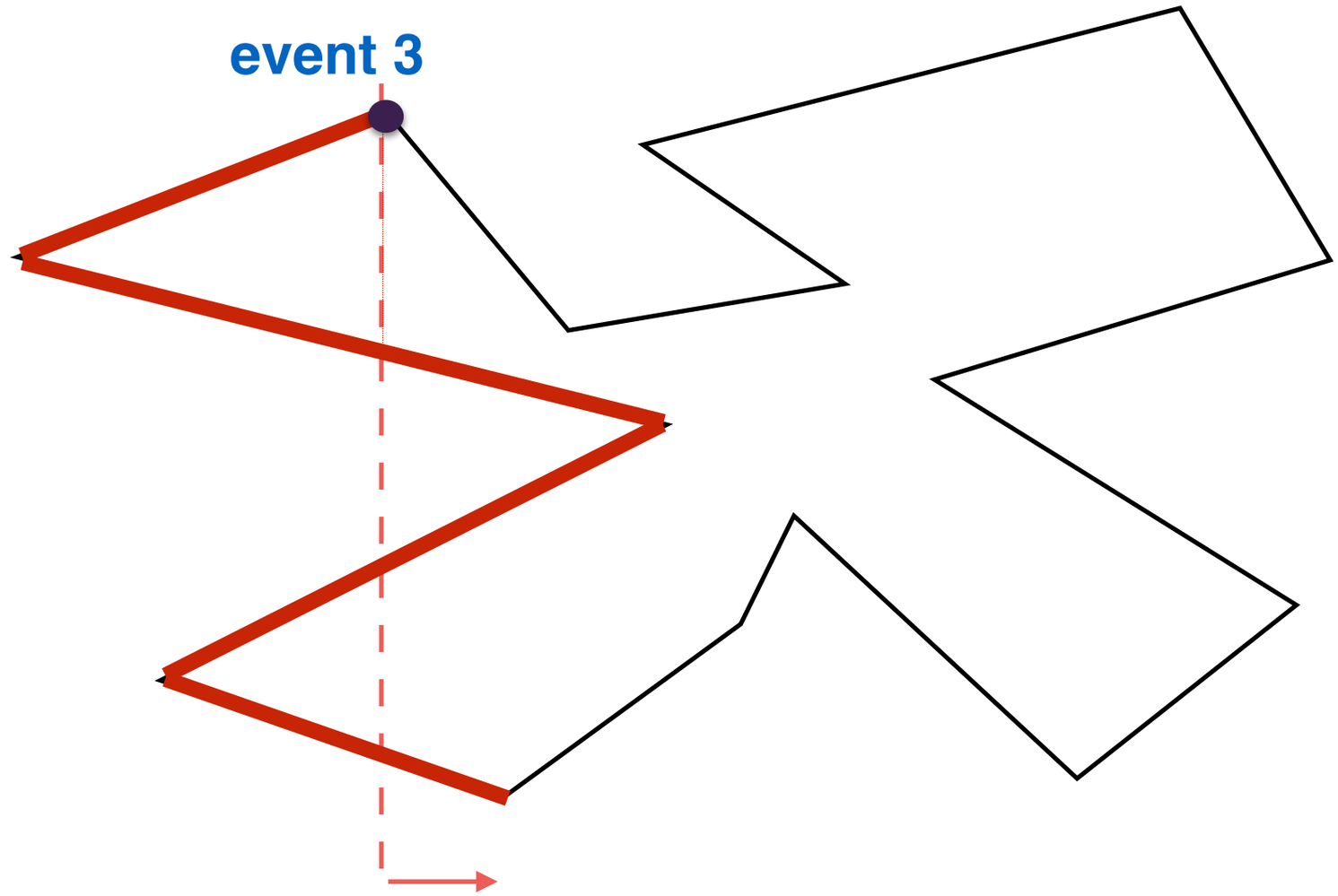
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

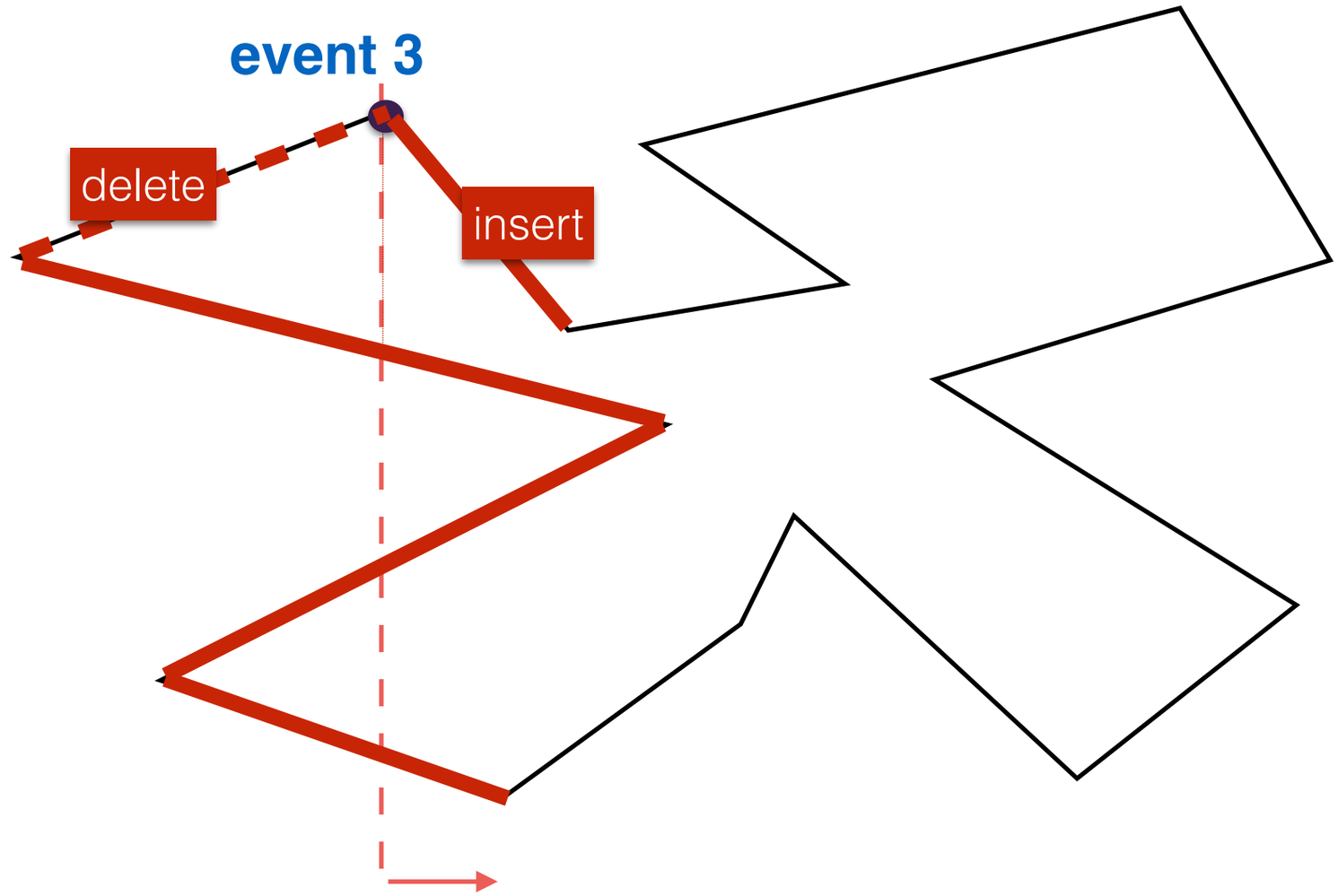
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

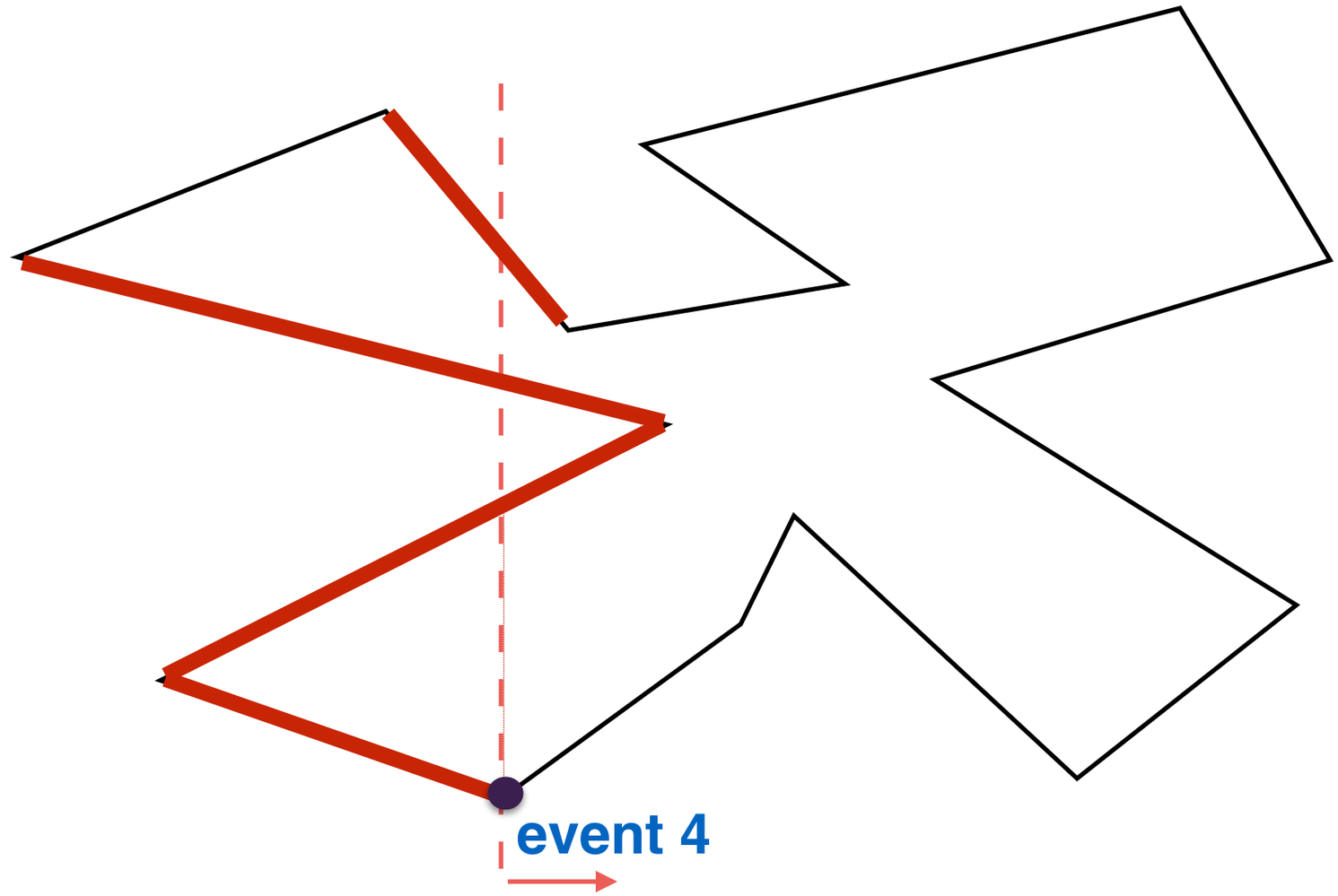
- Line sweep



events = polygon vertices

Computing the trapezoid partition in O(n lg n)

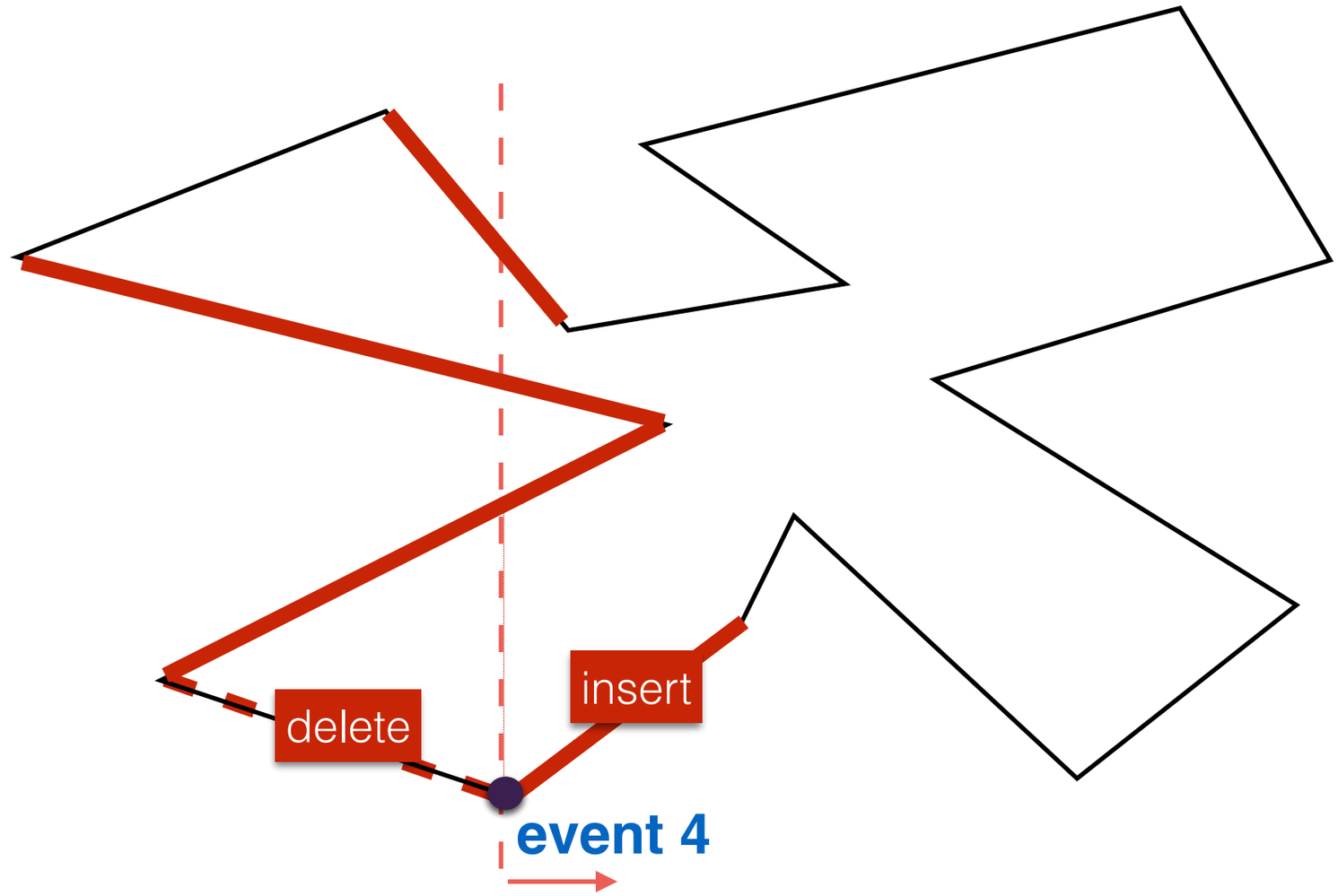
- Line sweep



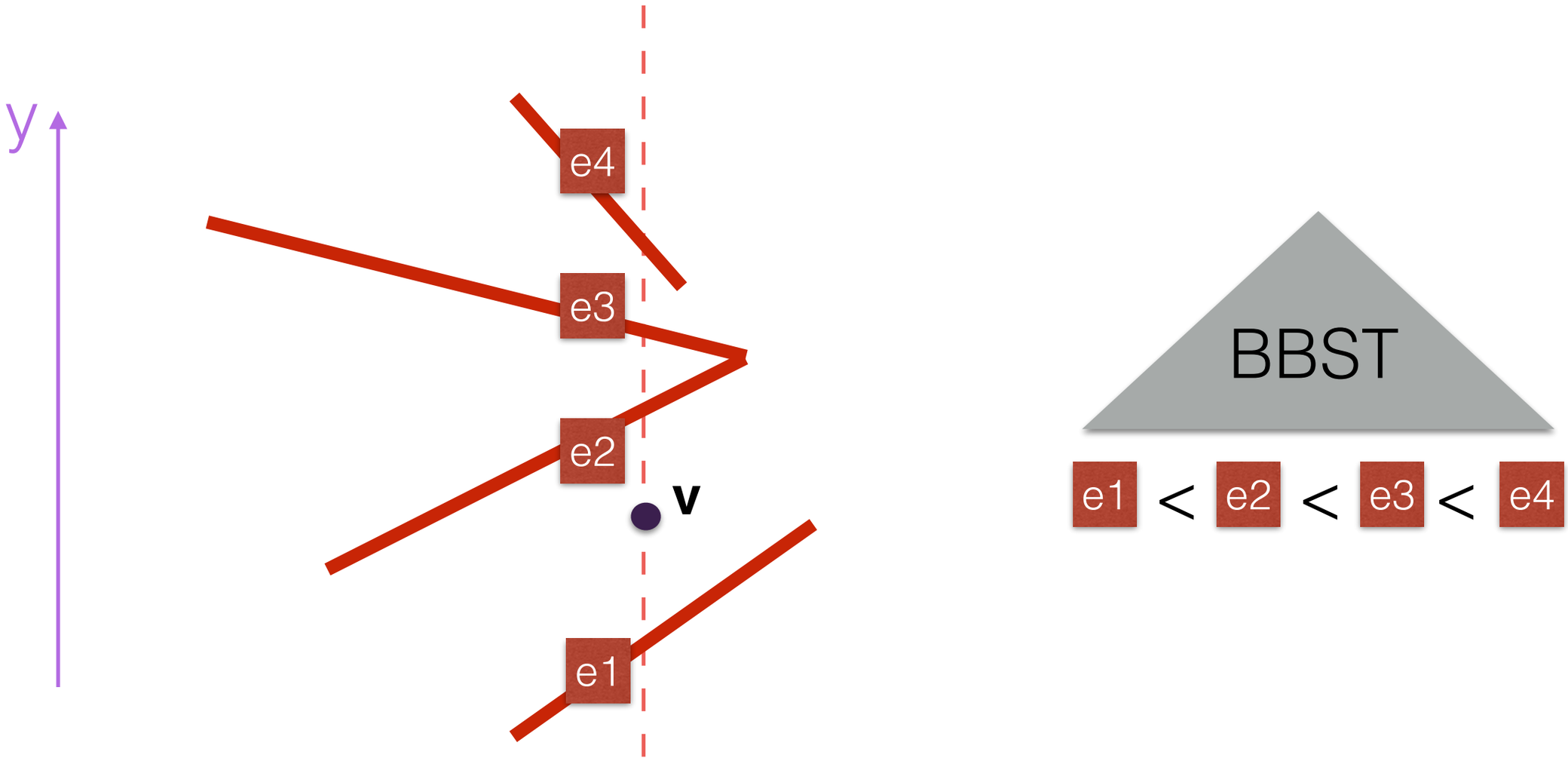
events = polygon vertices

Computing the trapezoid partition in O(n lg n)

- Line sweep

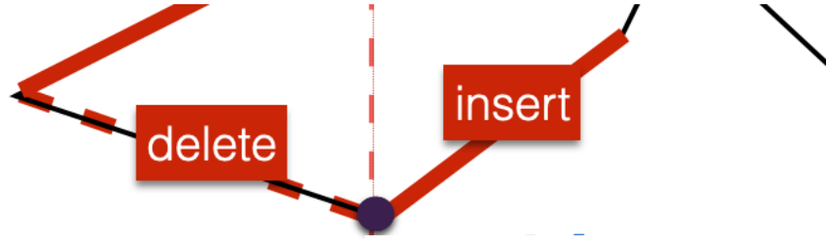


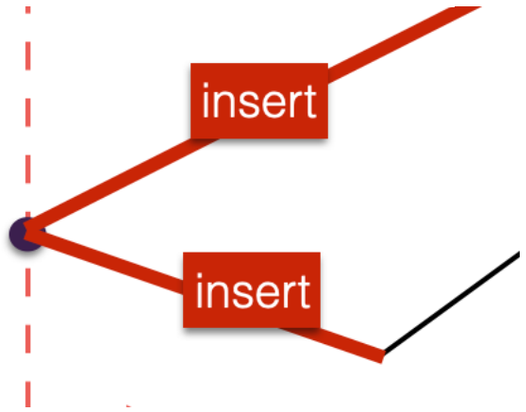
- Active structure: edges that intersect current sweep line, in y-order, kept in a BBST



- We use the AS to find the first edge intersected by a ray from v
    - for upward ray: first edge intersected by ray is AS.succ(v)
    - for a downward ray: first edge intersected by ray is AS.pred(v)

Algorithm





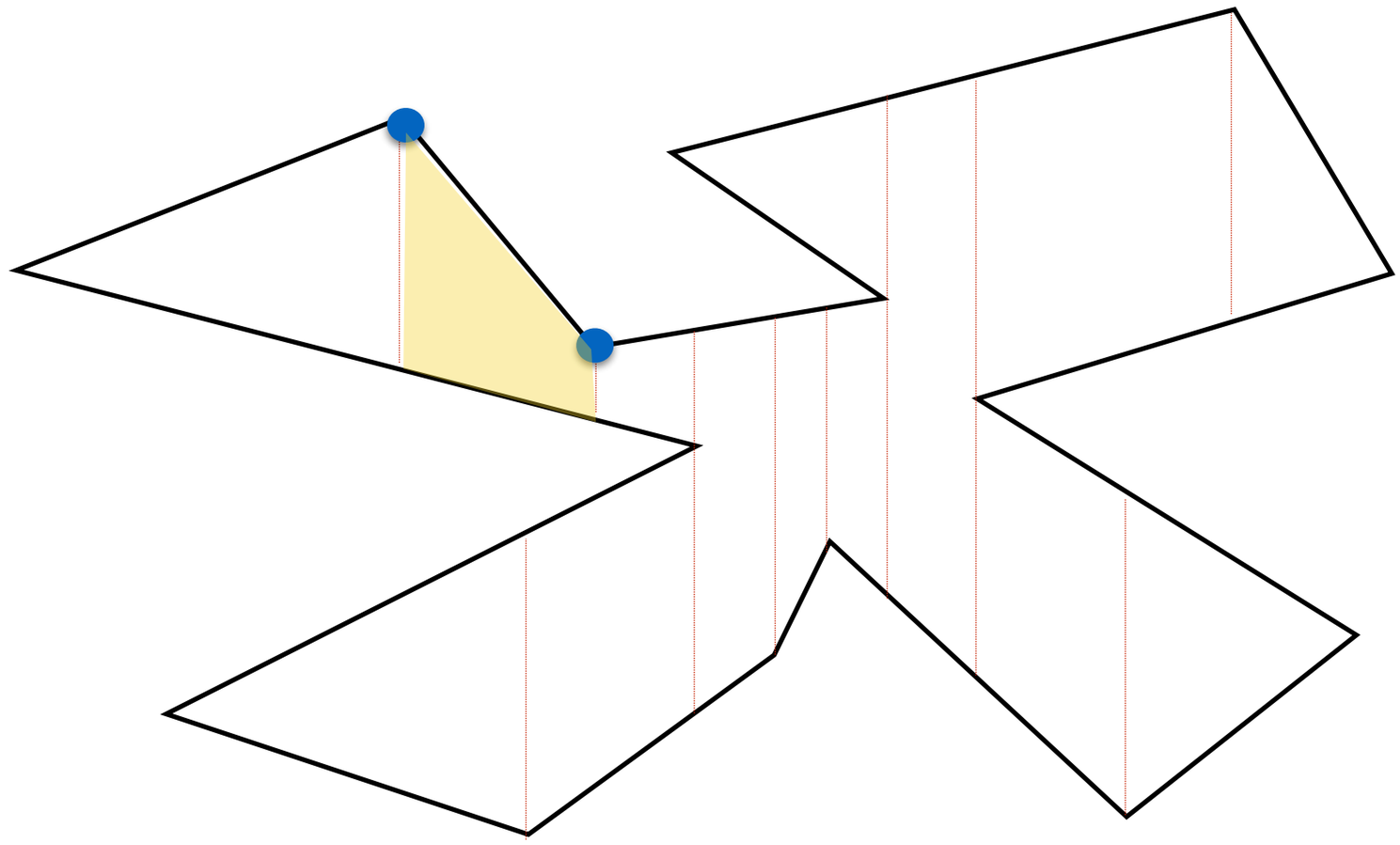
- LineSweep(V)
  - sort_by_x(V)
  - AS = { }
  - For each vertex v in x order:
    - let e1, e2 be the two edges with v as endpoint
    - if e1 is left of v: AS.delete(e1); else AS.insert(e1)
    - if e2 is left of v: AS.delete(e2); else AS.insert(e2)
    - Figure out which case v is and. how to shoot the ray (up, down, both, none). Use the AS to find the edge that intersects the ray (AS.succ(v) or AS.pred(v) or both). Compute the intersection point
    - create current trapezoid (…)

Analysis: O(n lg n) with a BBST

We’ve seen how to use the AS to find the edges intersected by the rays efficiently

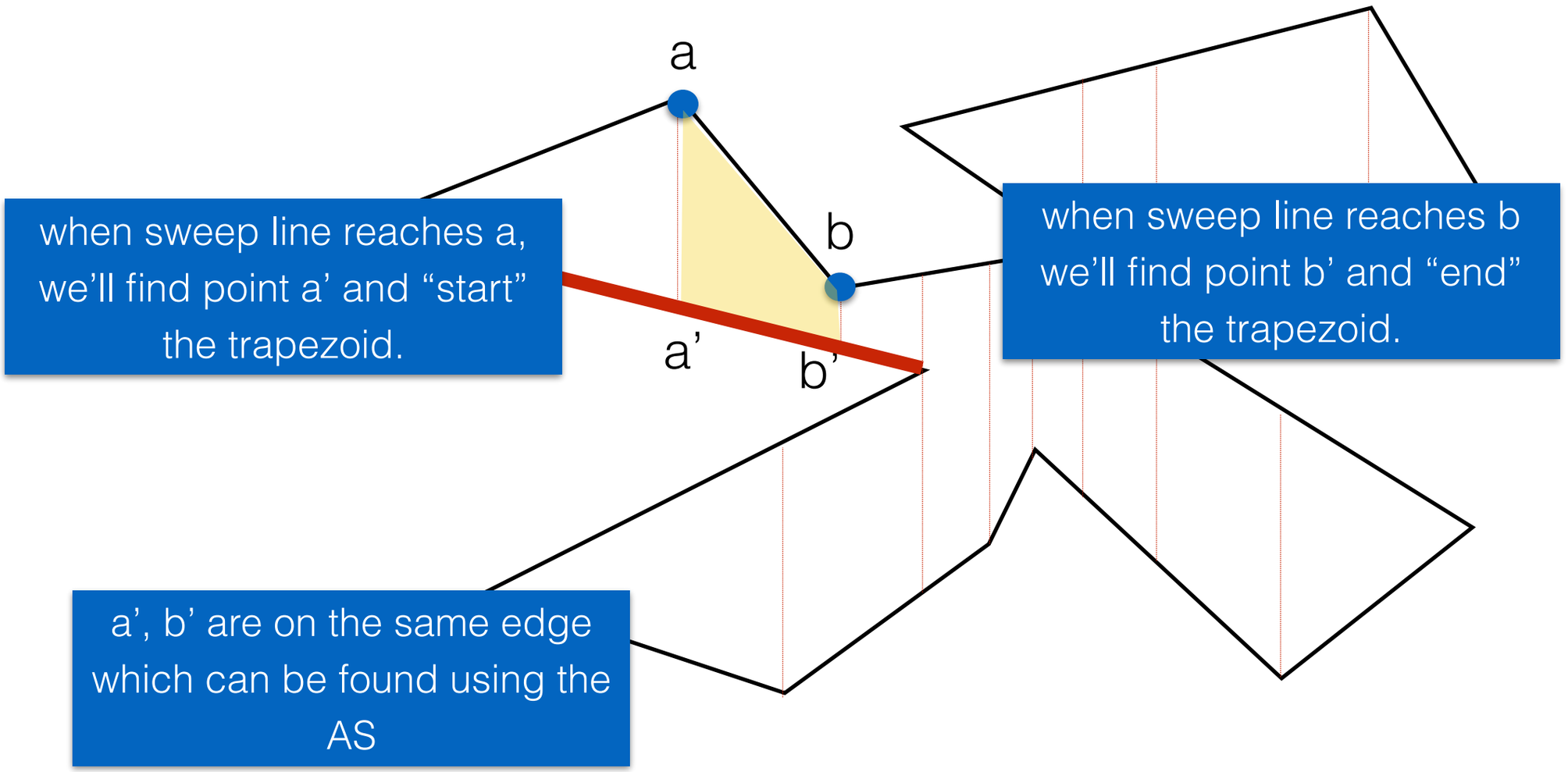
Some detail left: How to set the trapezoids?

How can we determine the trapezoids?



Each trapezoid has 2 vertices of P, one on the left and one on the right

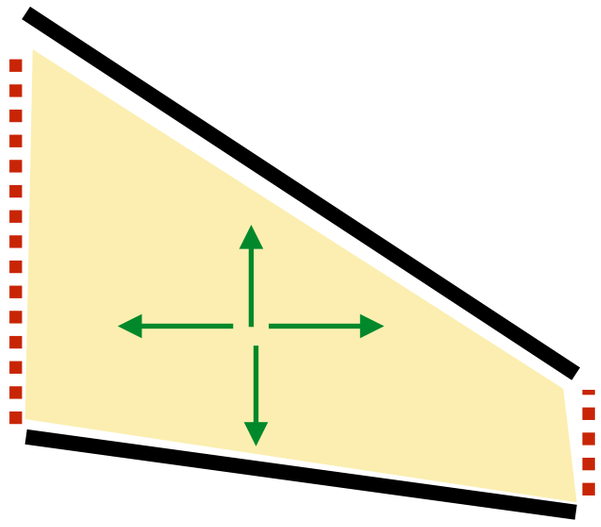
How can we determine the trapezoids?



Each trapezoid has 2 vertices of P, one on the left and one on the right

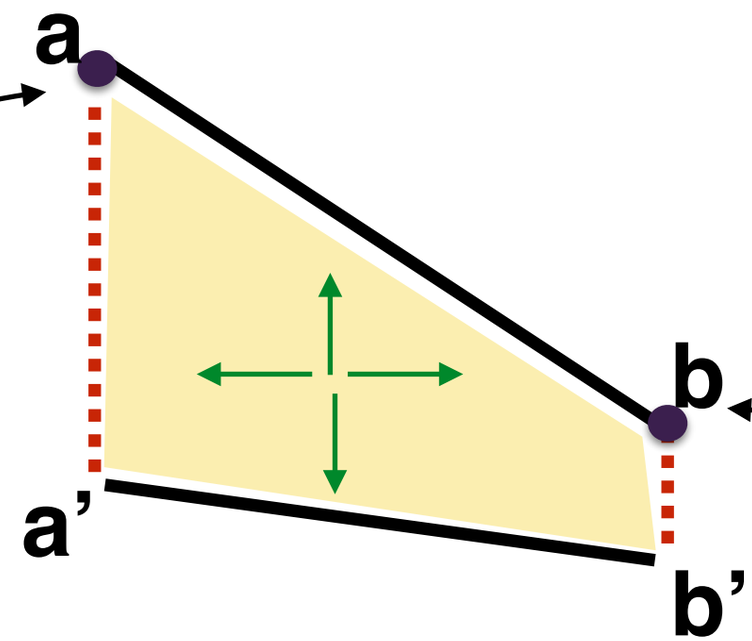
How can we determine the trapezoids?

- Let's say we want to set up pointers from a trapezoid to its edges.



How can we determine the trapezoids?

- For e.g. in this case:

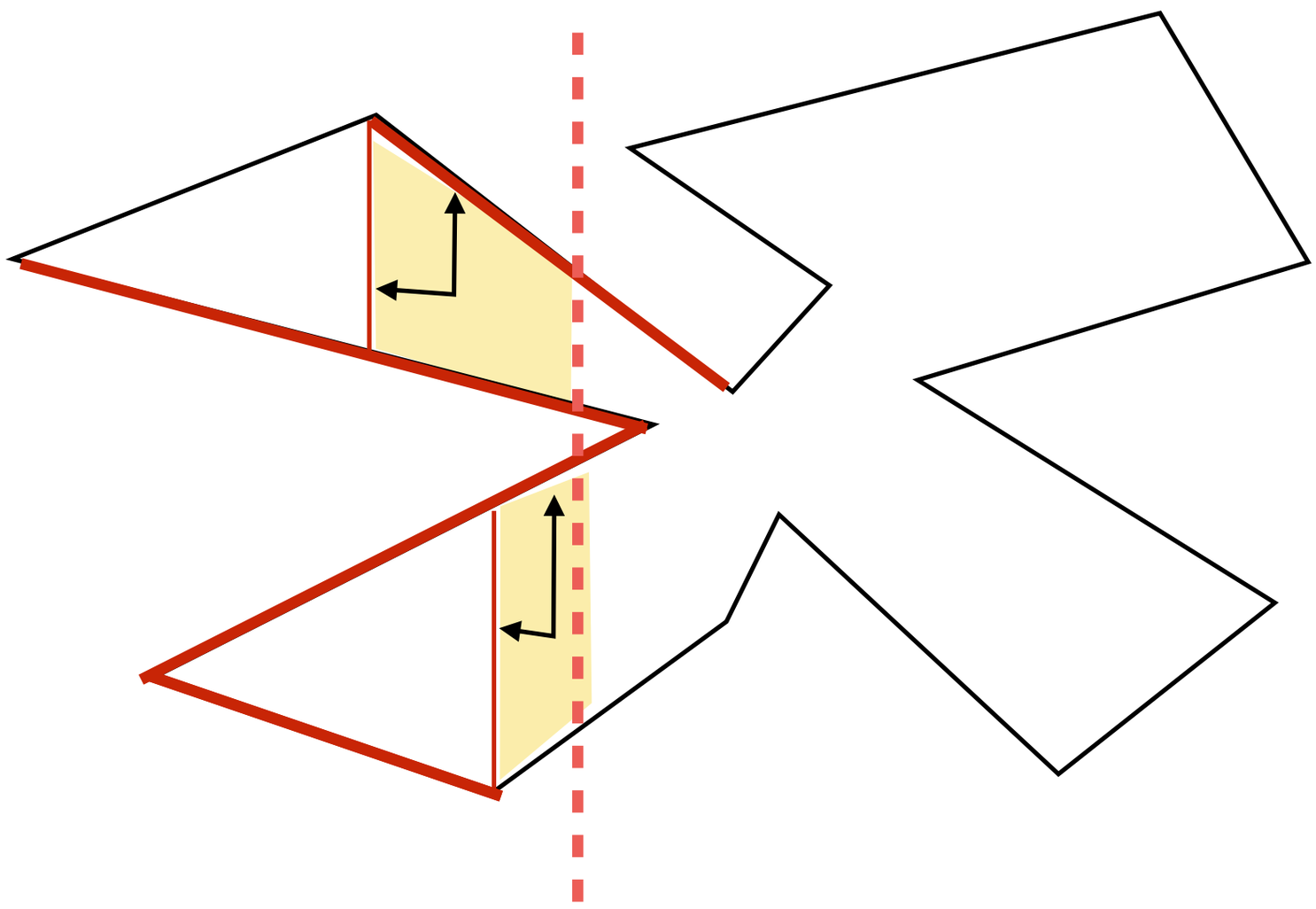


vertex a finds a’ and “opens” the trapezoid: it sets a pointer to aa’, and top and bottom edges.

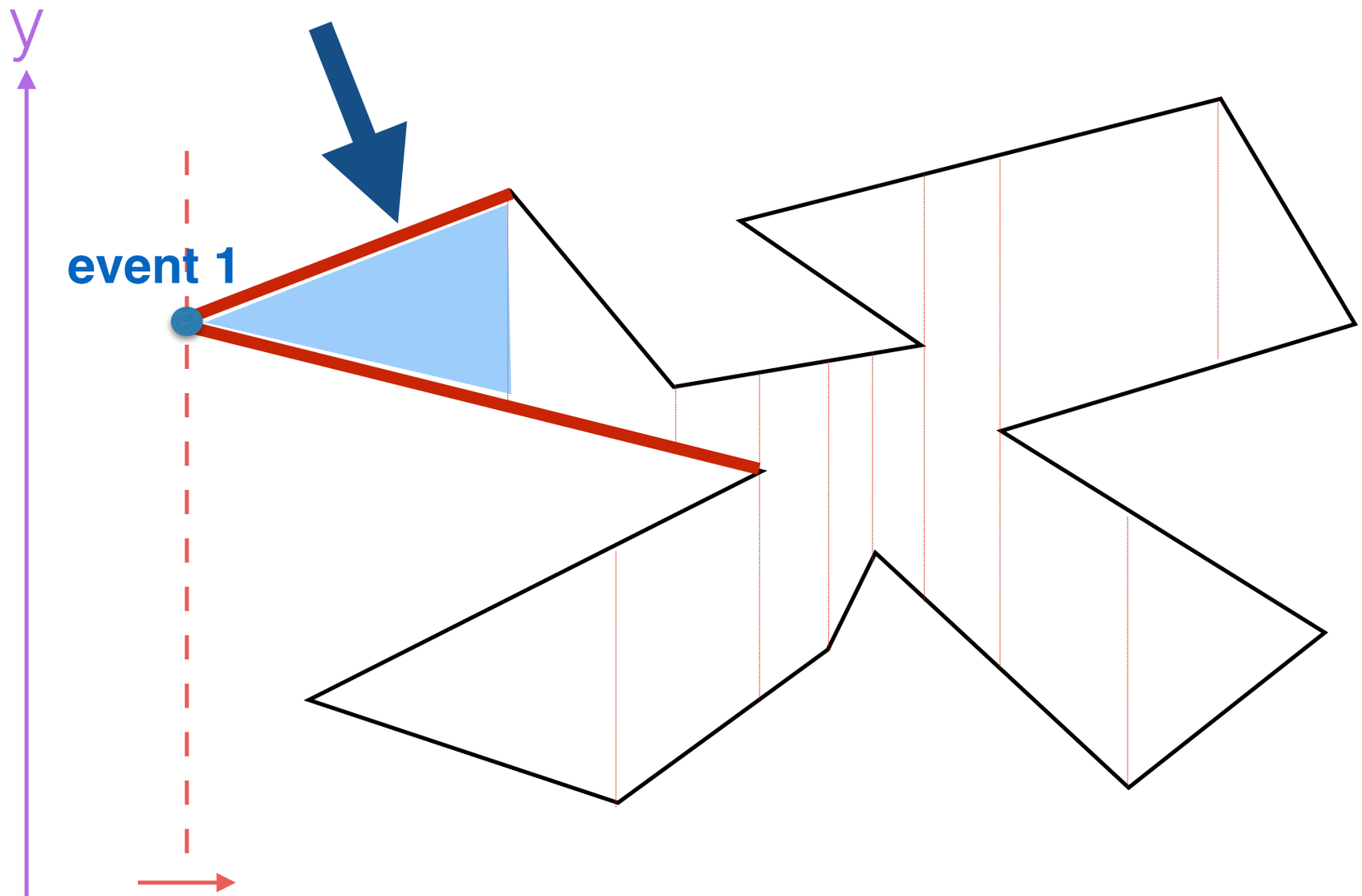
vertex b finds b’ and “closes” the trapezoid: it sets a pointer to bb’.

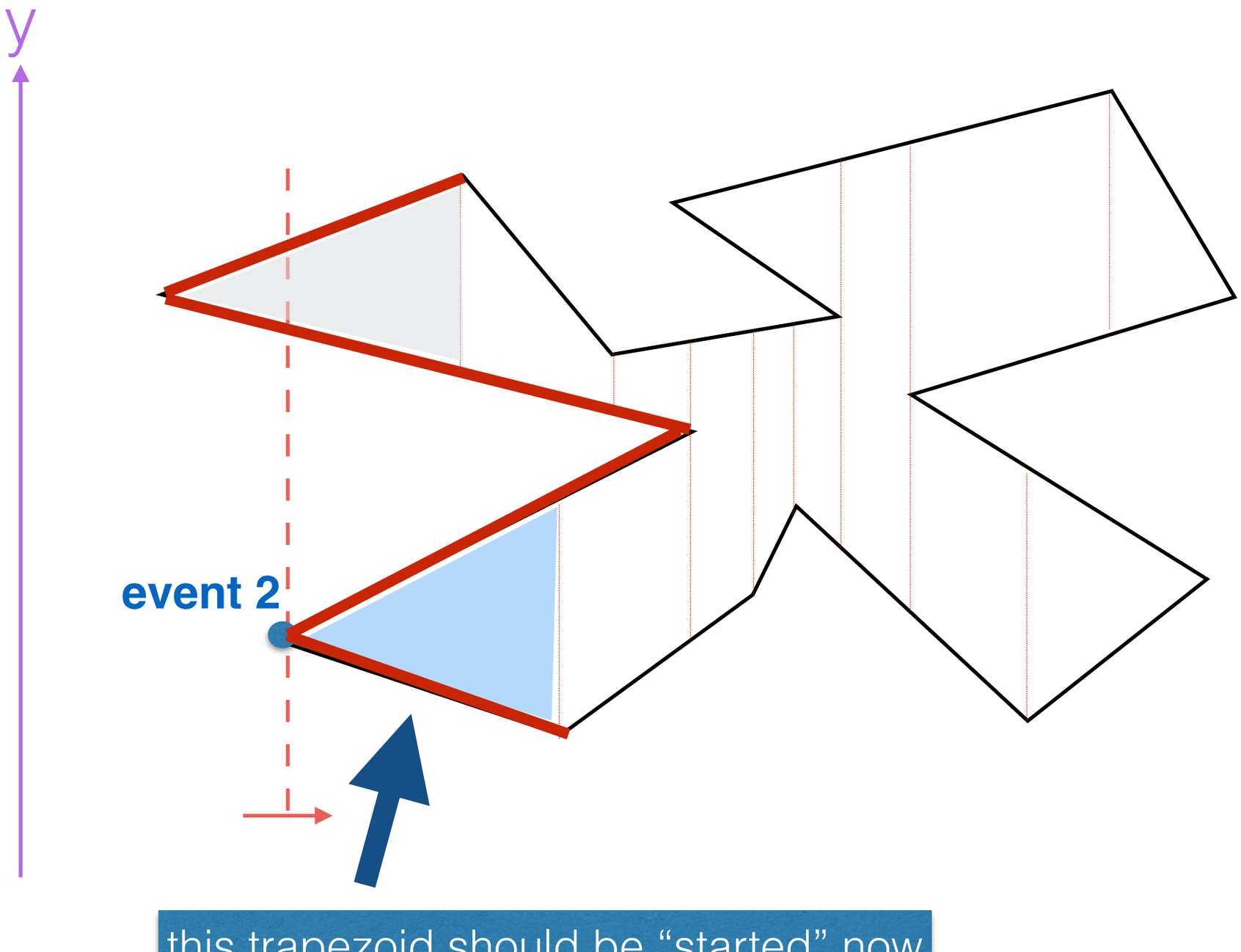
How can we determine the trapezoids?

- With each segment in the AS: we store a pointer to the vertical trapezoid directly below it (if the interior of the polygon is directly below it)
- This pointer stores the left side of the currently open trapezoid , with that segment as the top side

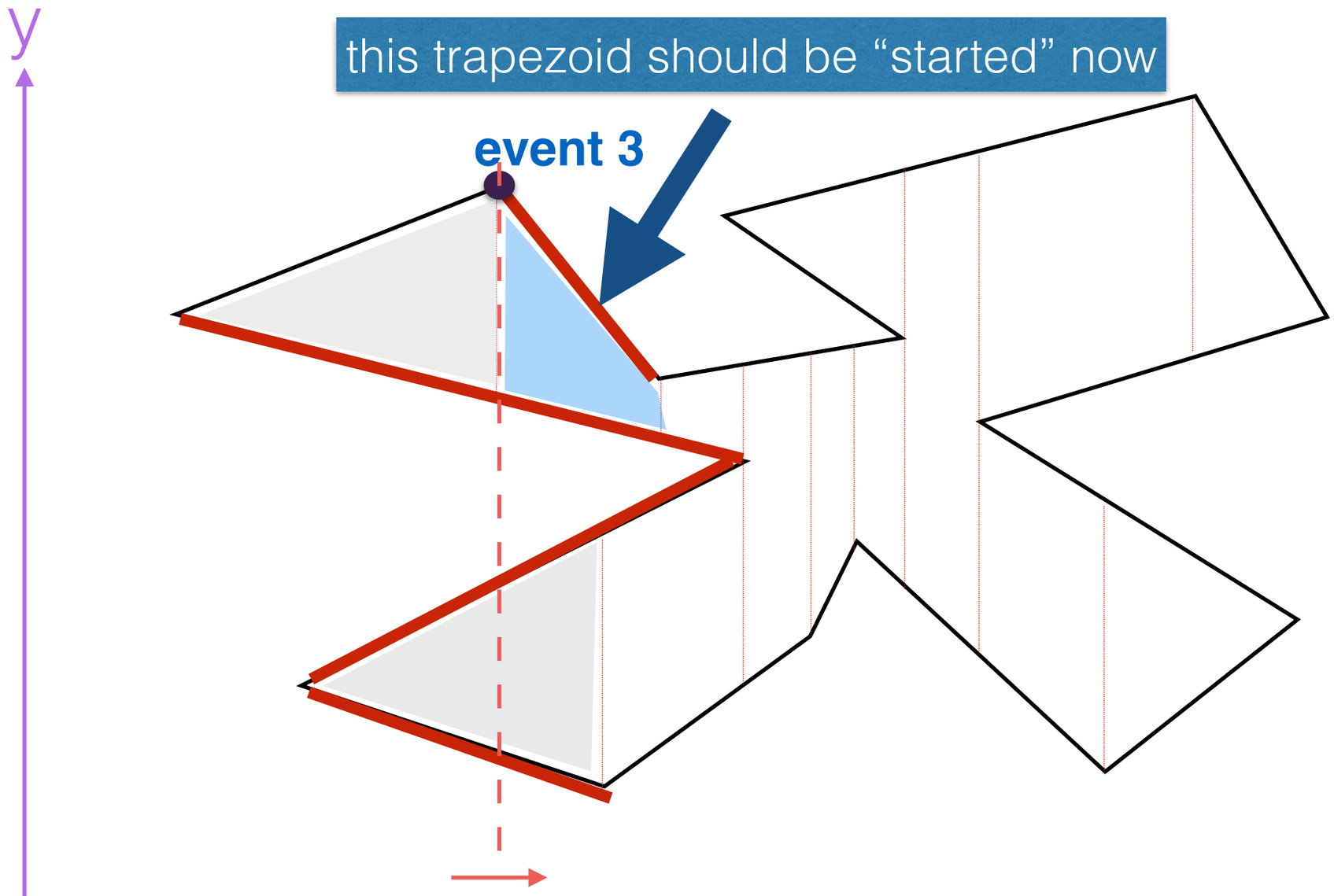


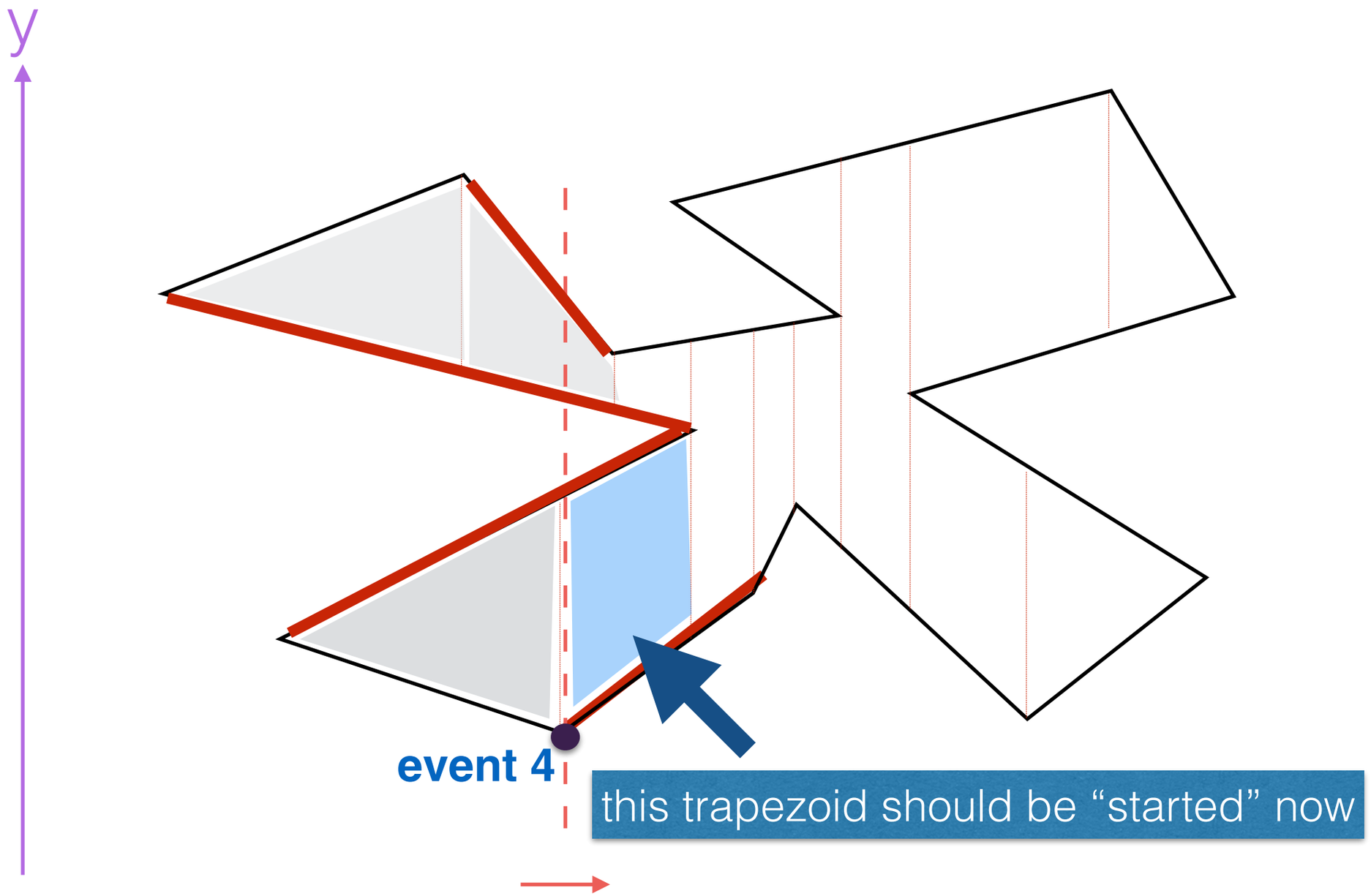
this trapezoid should be “started” now

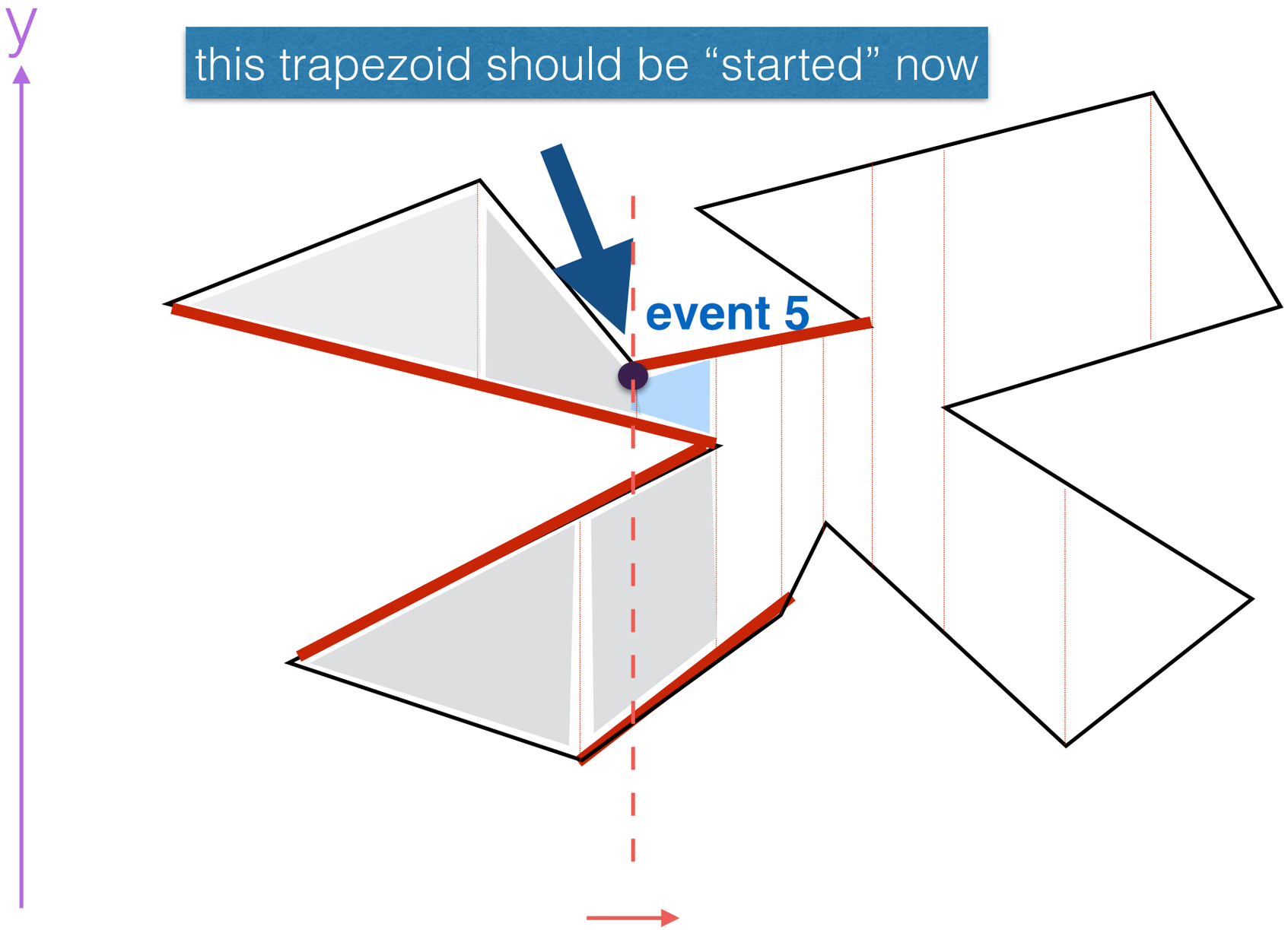


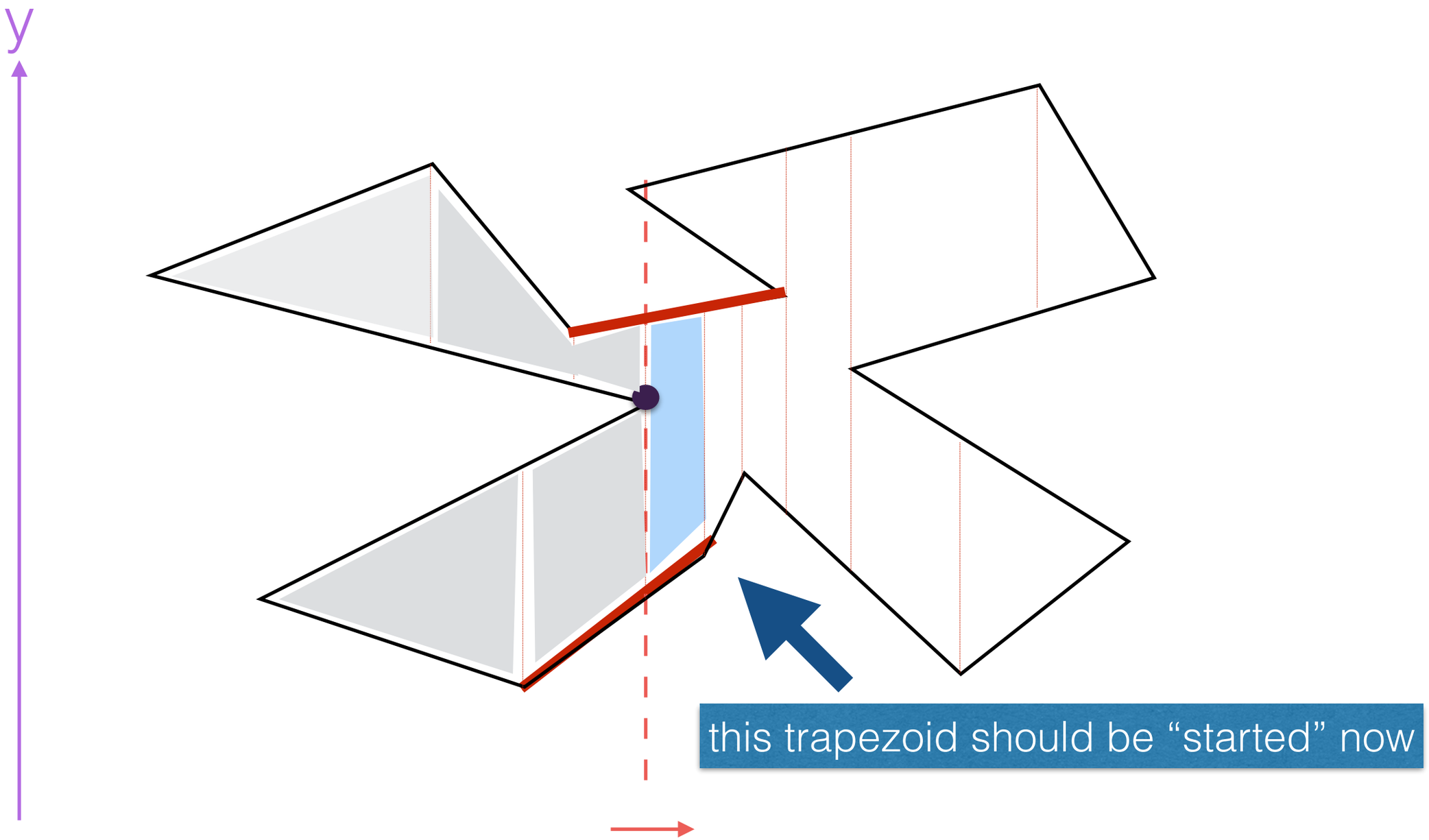


this trapezoid should be “started” now

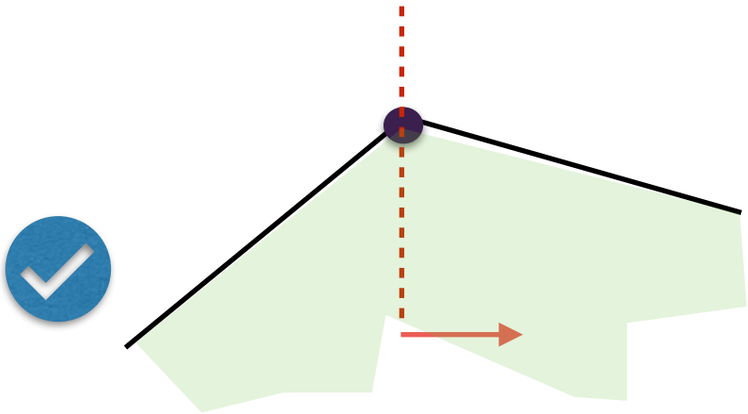

y
this trapezoid should be “started” now
event 3

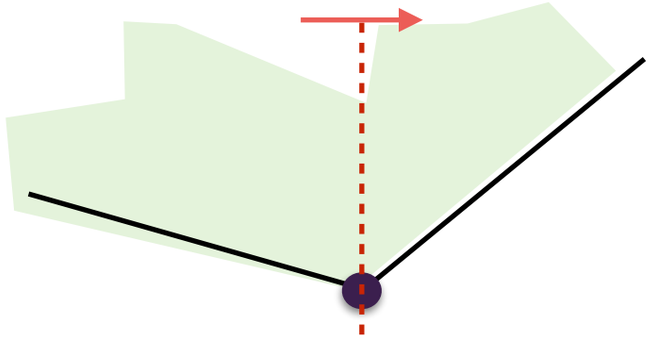

y
event 4
this trapezoid should be “started” now

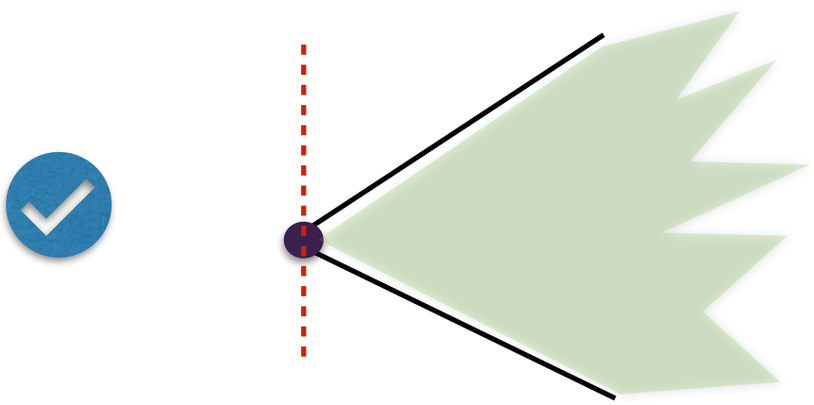

y
this trapezoid should be “started” now
event 5

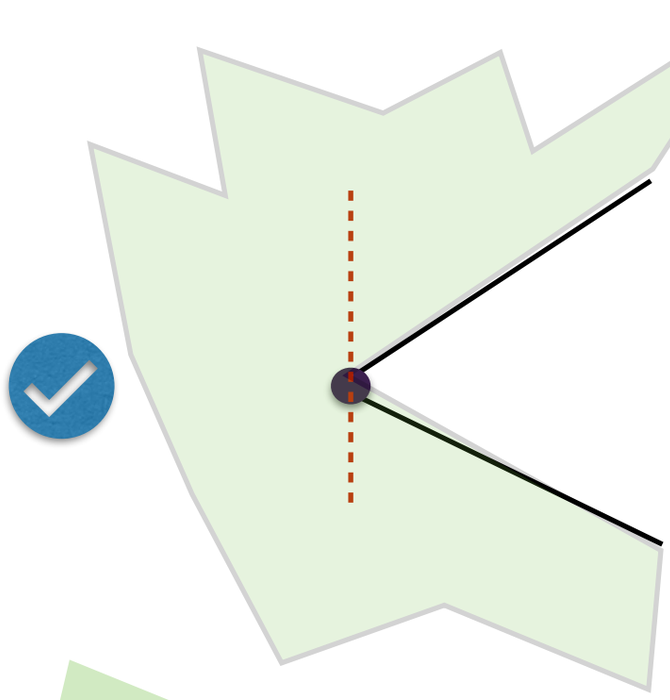


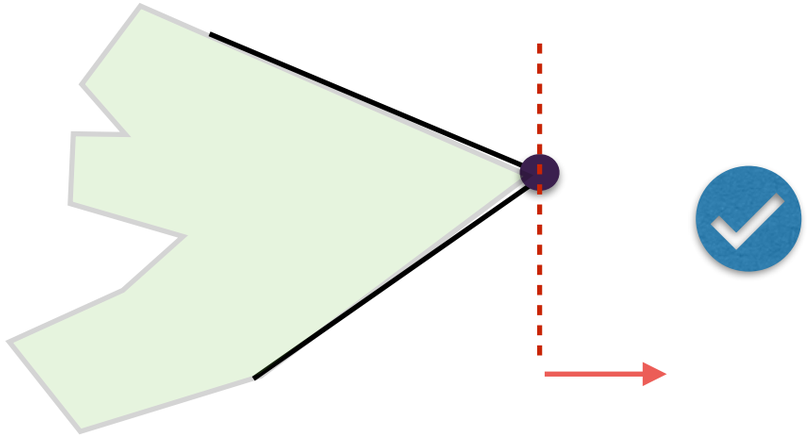
and so on

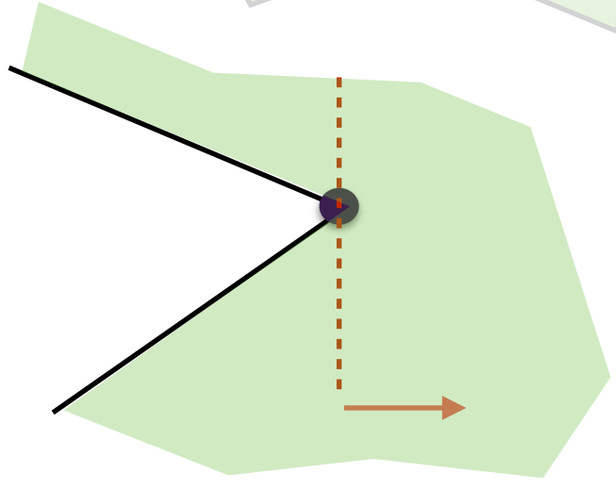




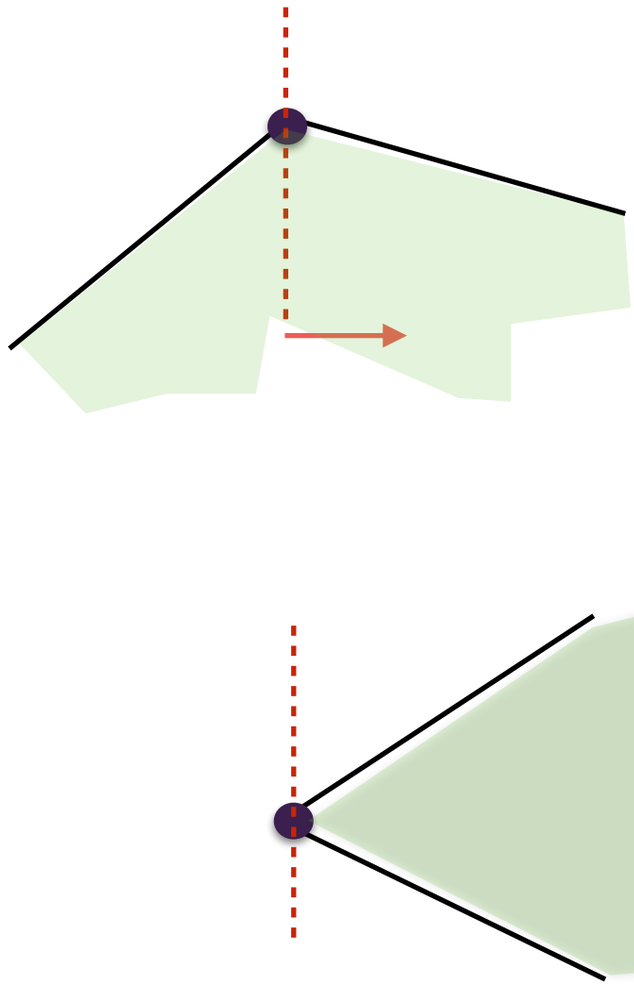


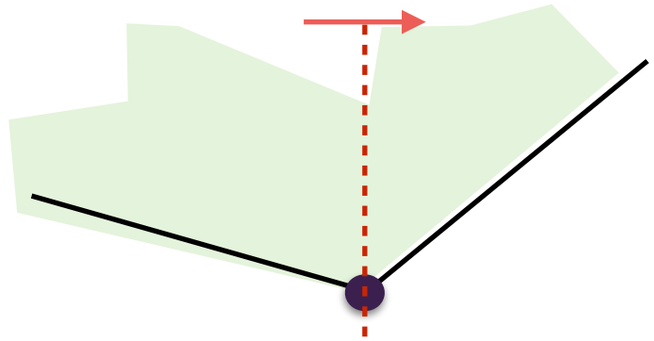


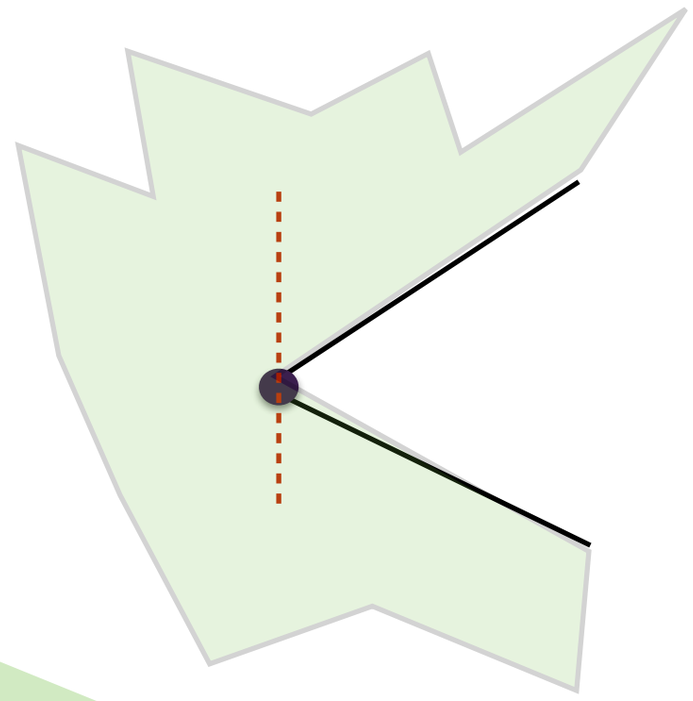


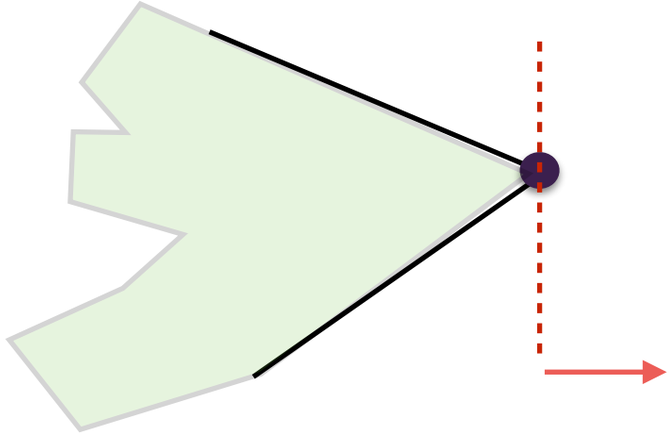


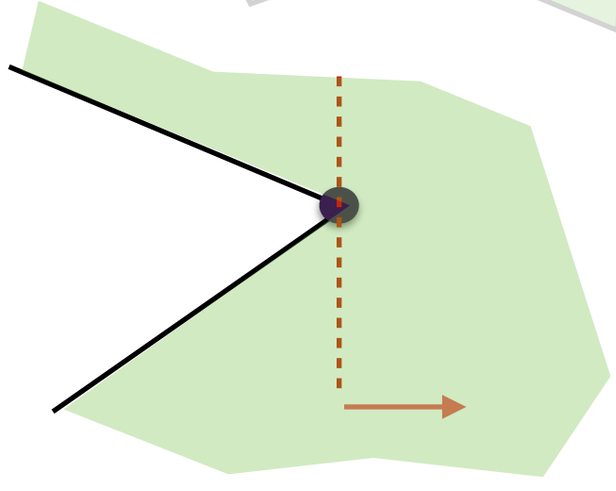
Type of events



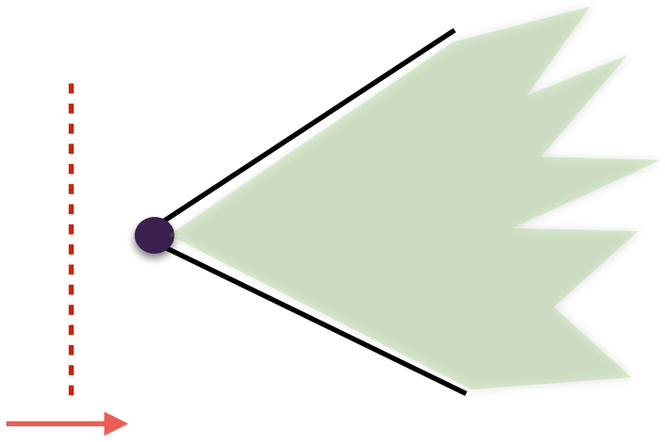




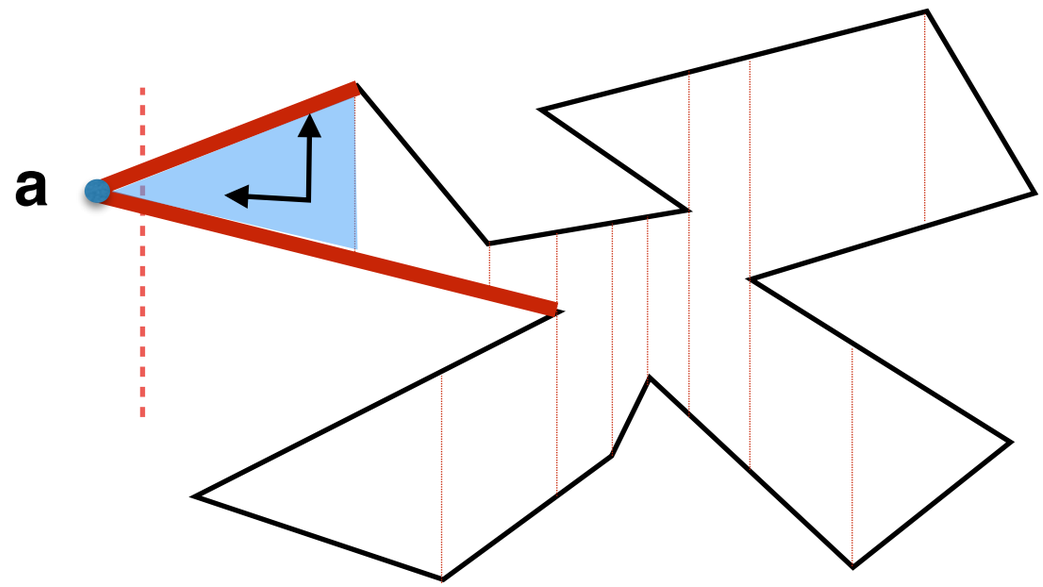




Handling events, details:



Example

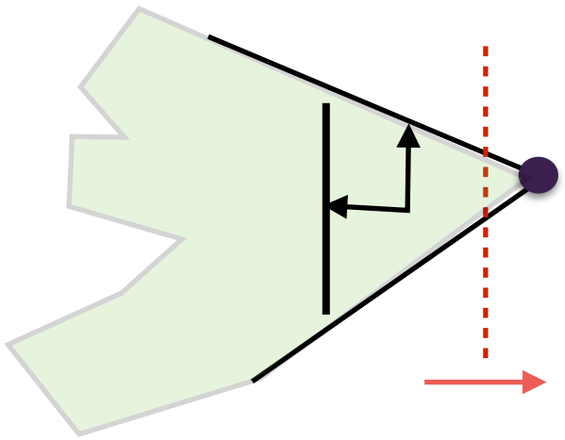


Do:

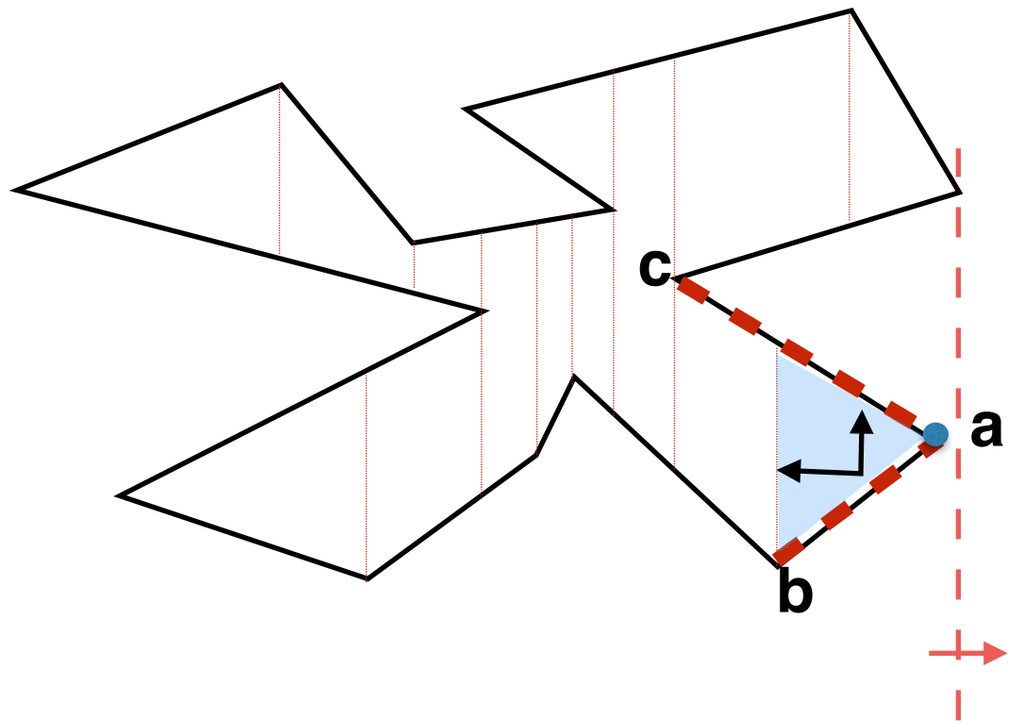
insert edges in AS

“start” trapezoid to the right of a

Handling events, details:



Example

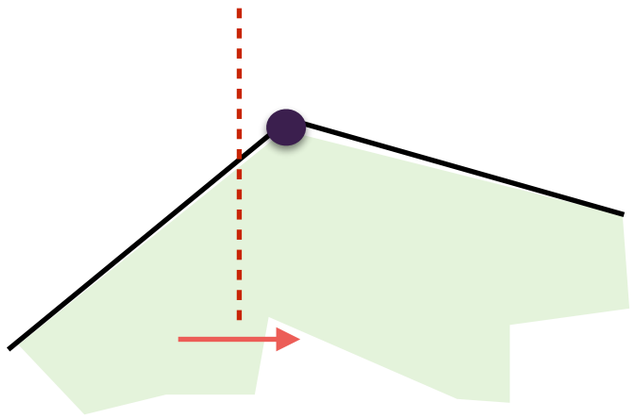


Do:

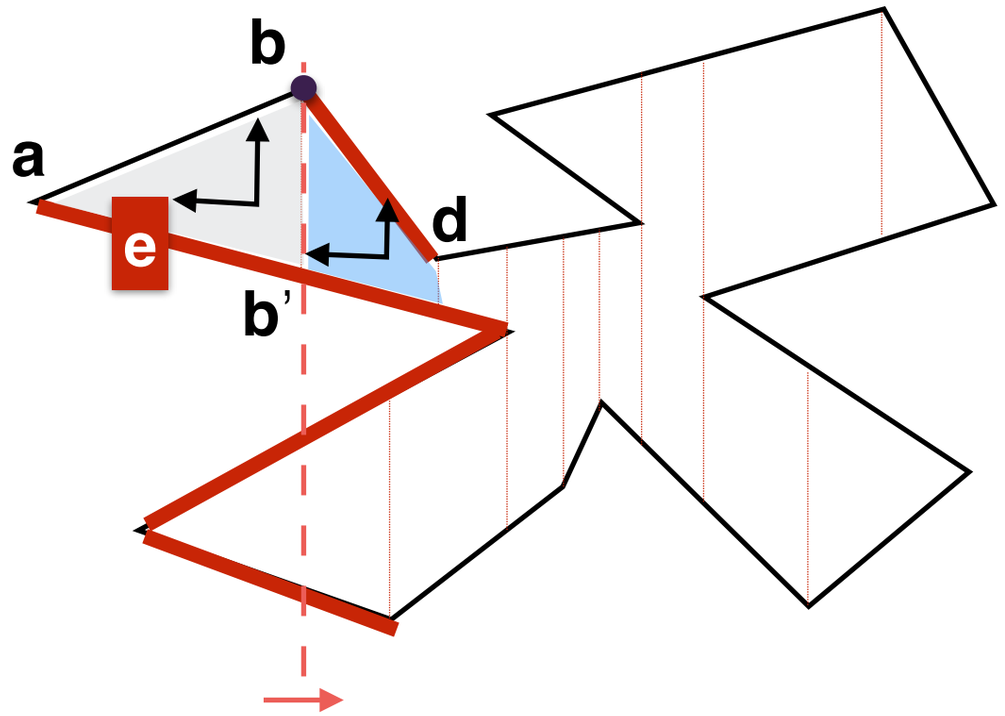
delete edges from AS

“end” trapezoid to the left of a

Handling events, details:



Example



b' must be on the edge that is right below b in the active structure

Do:

find e = pred(b) in AS and find b'=intersection (ray, e)

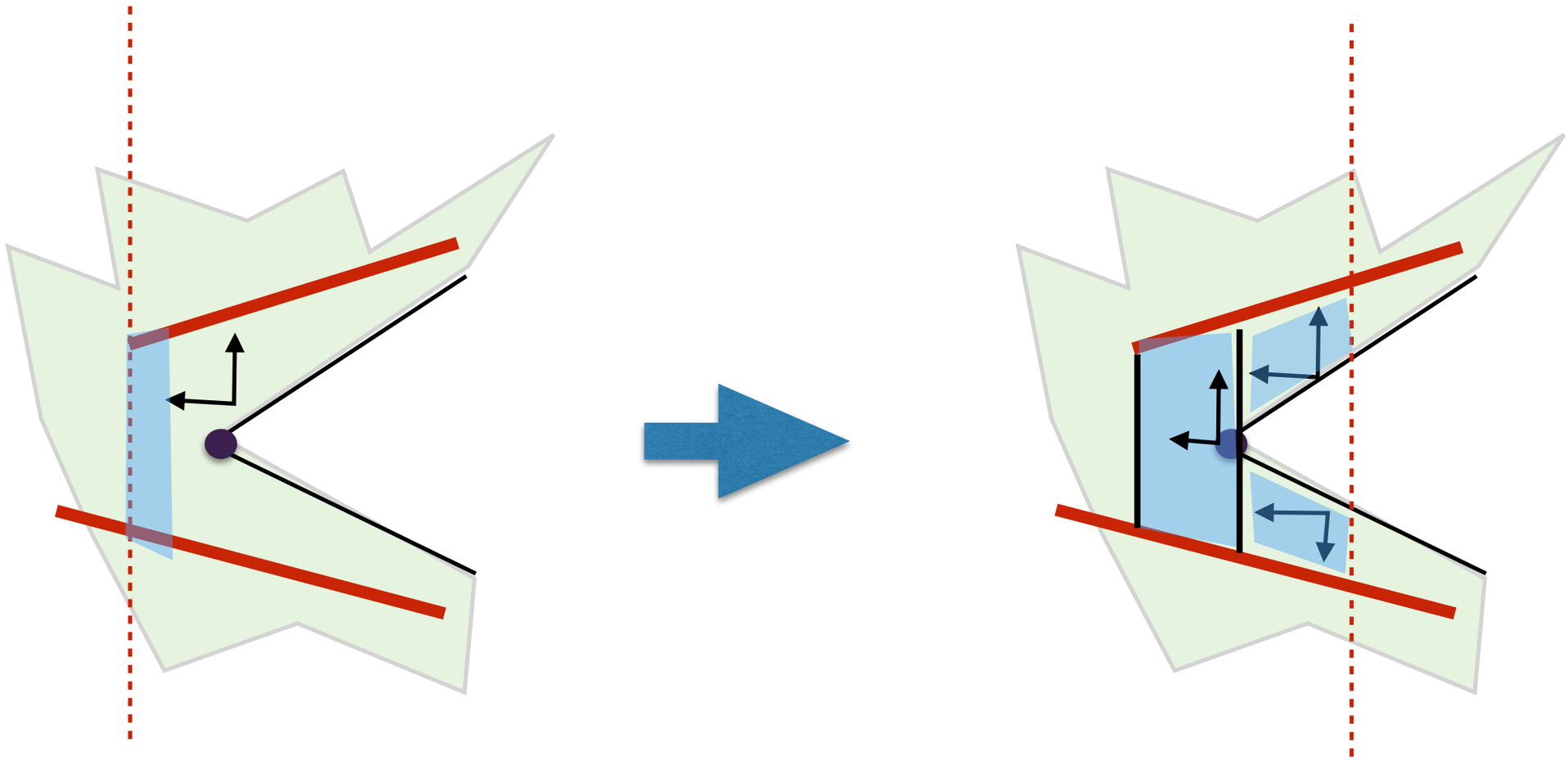
edge bd becomes active (insert bd in AS)

edge ab become inactive (delete ab from AS)

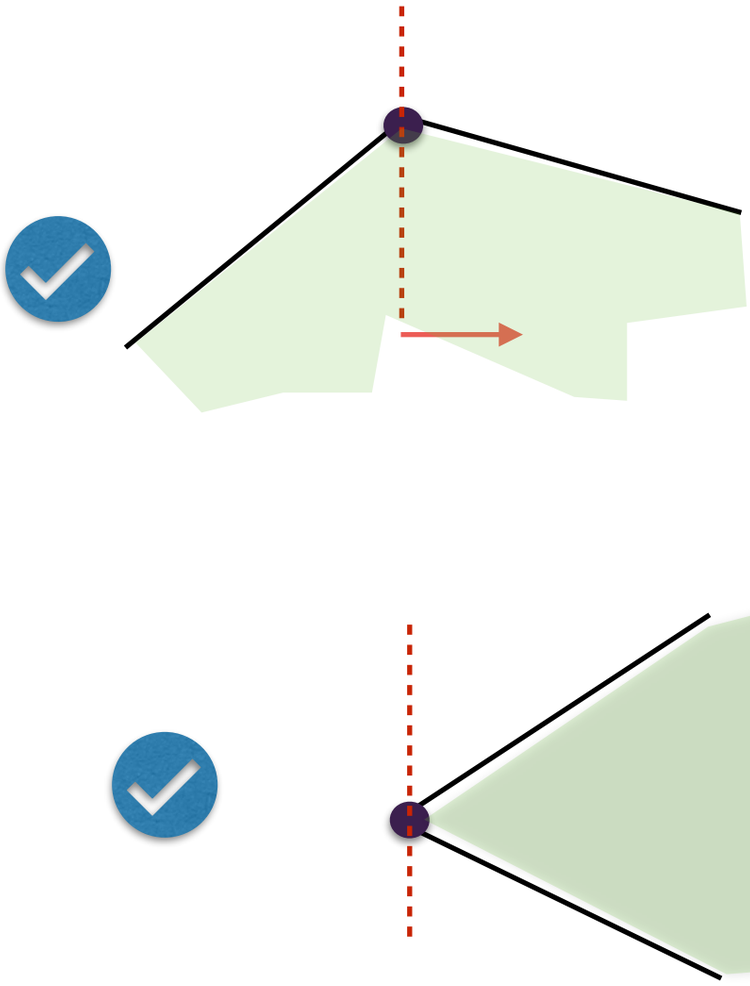
"end" prev. trapezoid to the left of bb'

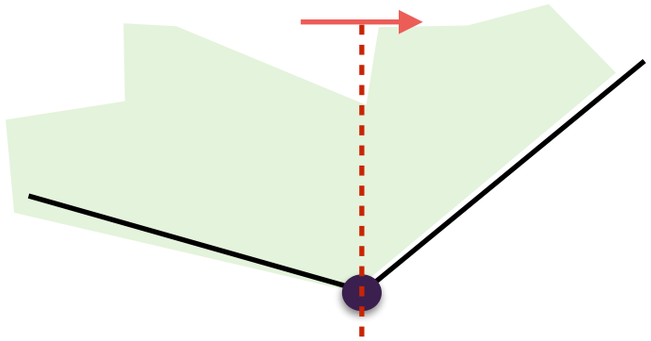
"start" the trapezoid right of bb'

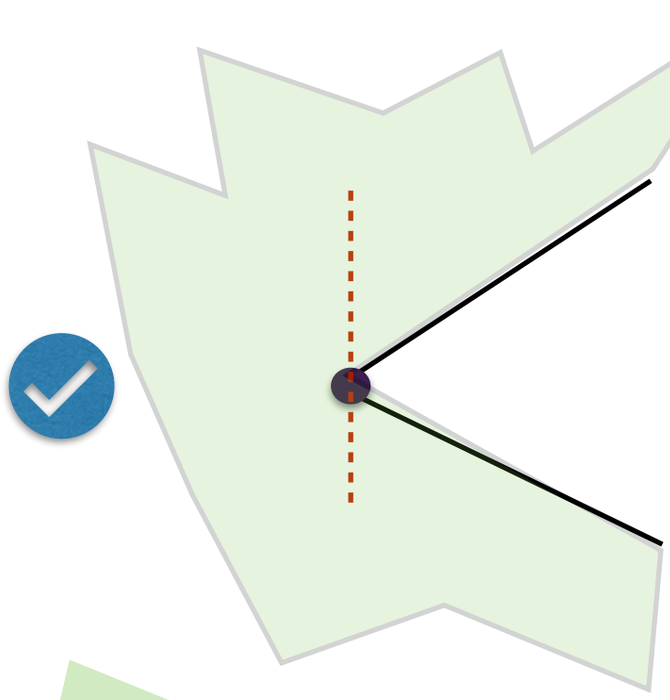
Handling events, details:

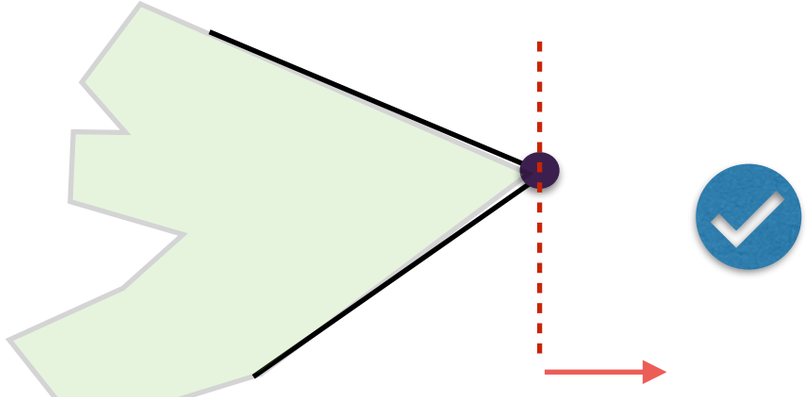


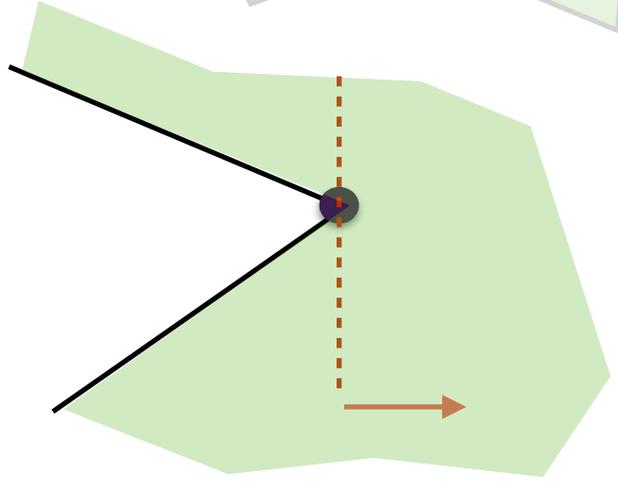
And so on…..











What have we learnt?

- A simple polygon can be triangulated in O(n lg n) time.

- Also, a trapezoidal decomposition of a polygon P can be computed in O(n lg n) time.

Tools