# Polygon Triangulation

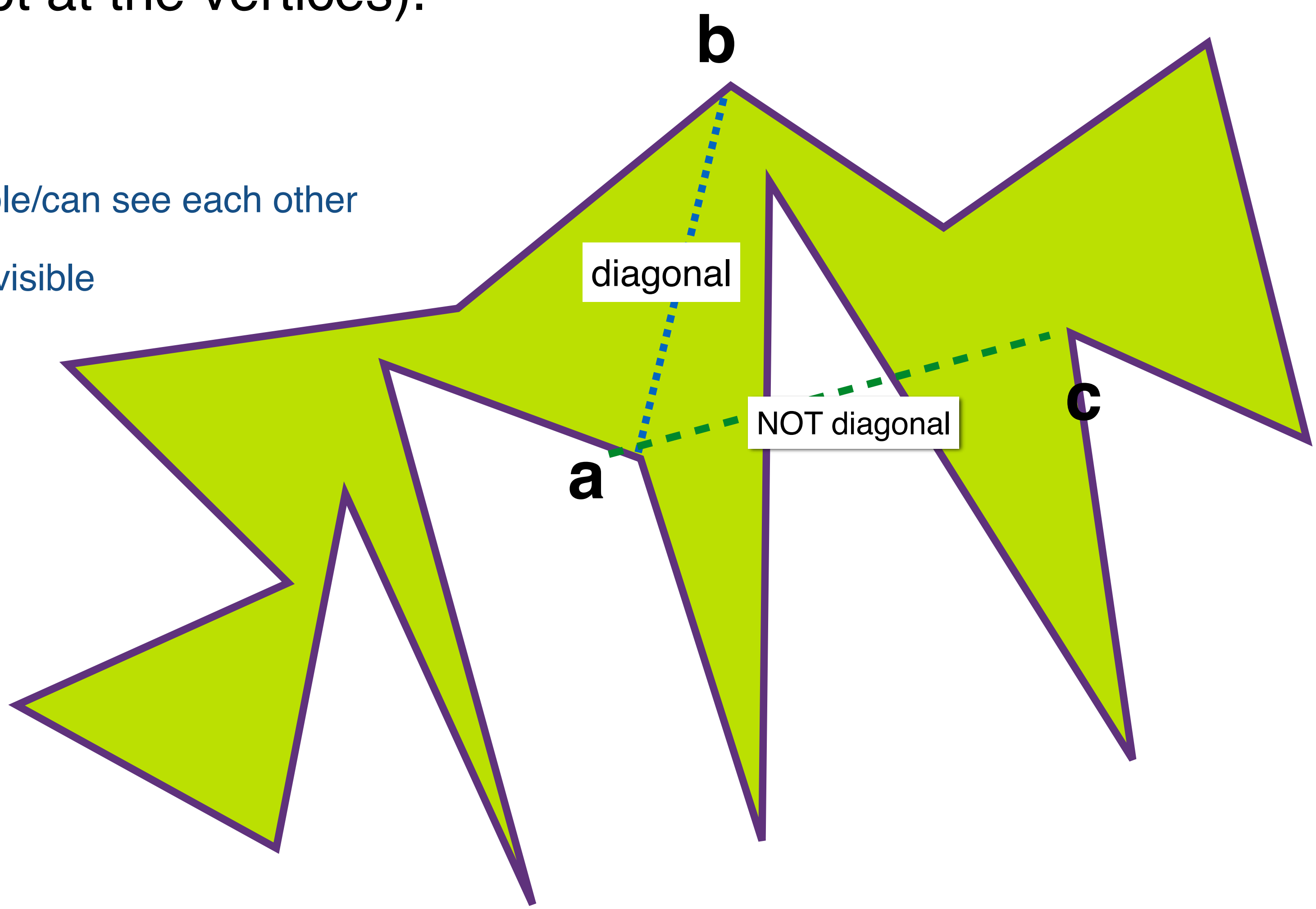
Computational Geometry [csci 3250]
Laura Toma
Bowdoin College

# Definition

Given a polygon, a diagonal is a line segment between two non-adjacent vertices of the polygon which does not intersect the polygon (except at the vertices).

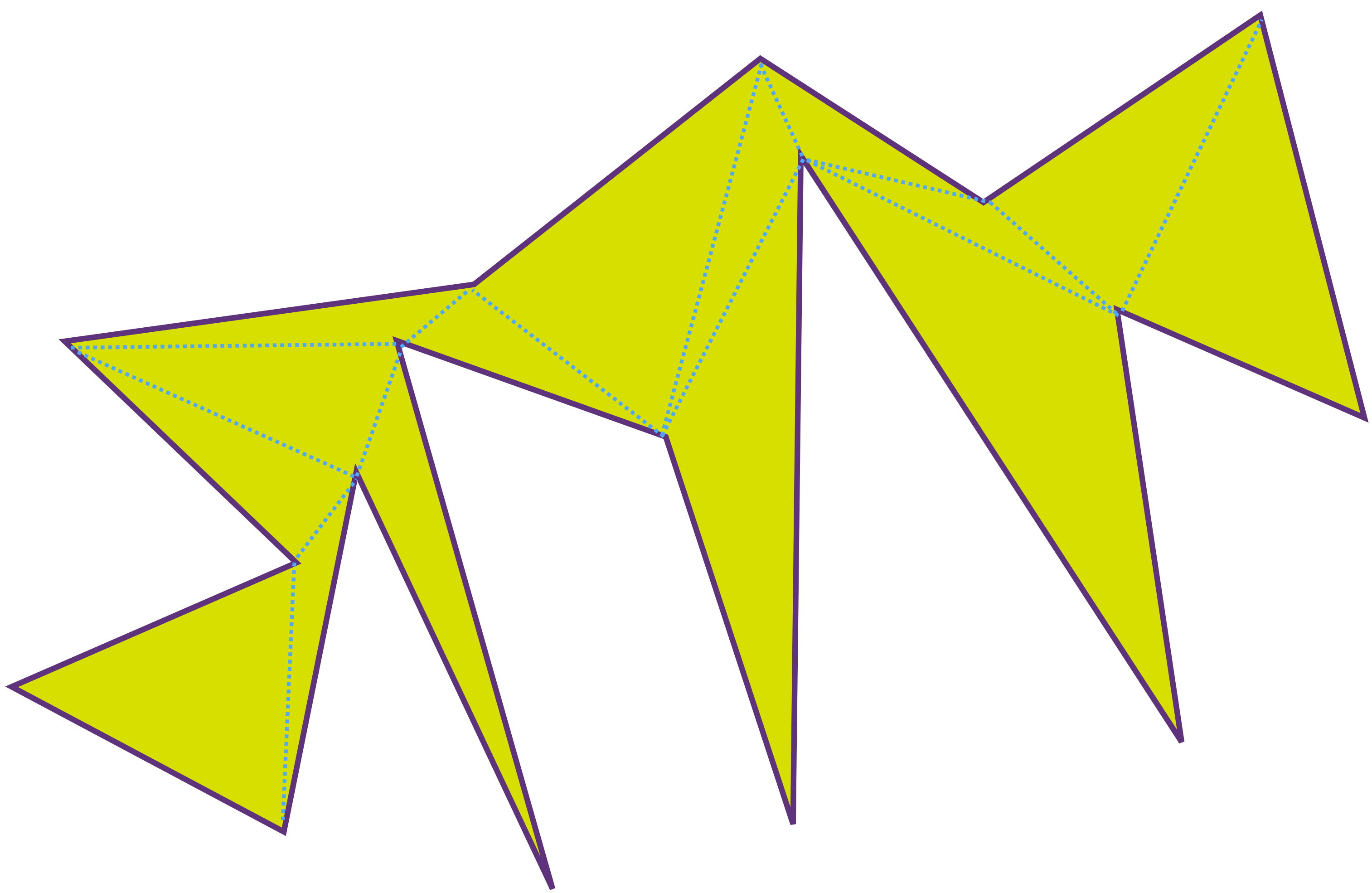
a, b visible/can see each other

a, c **not** visible



## Definition

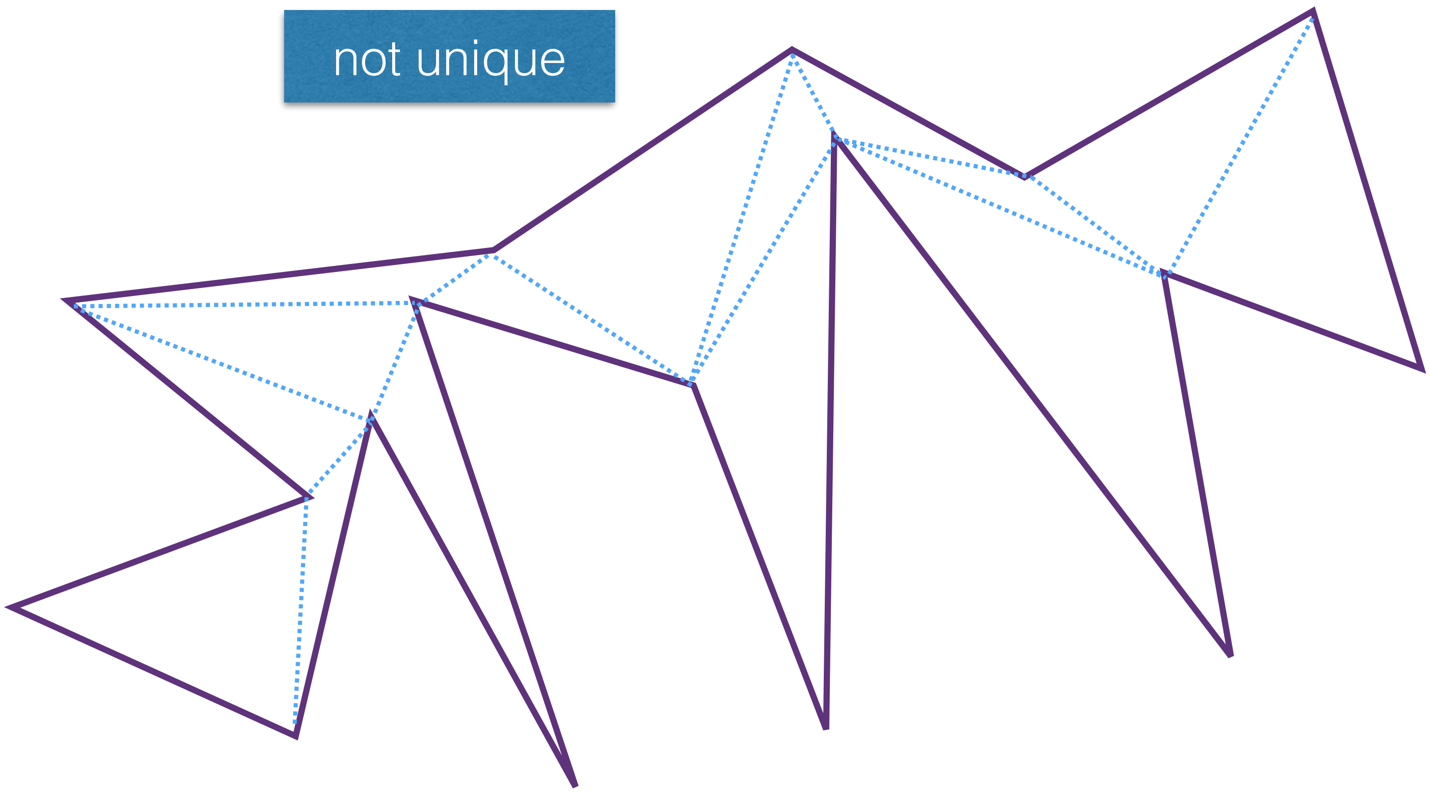
A **triangulation** of a polygon is a partition of the interior of the polygon into triangles using a set of non-intersecting diagonals.



## Goal

Given a polygon P: triangulate it, i.e. output a set of diagonals that partition the polygon into triangles.

not unique

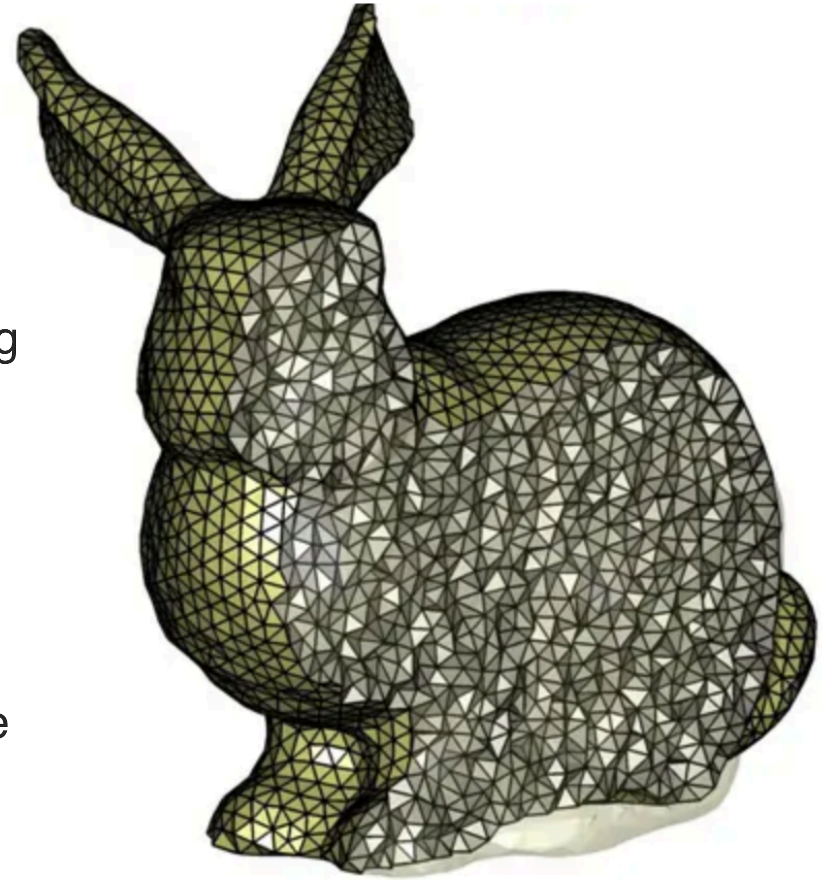


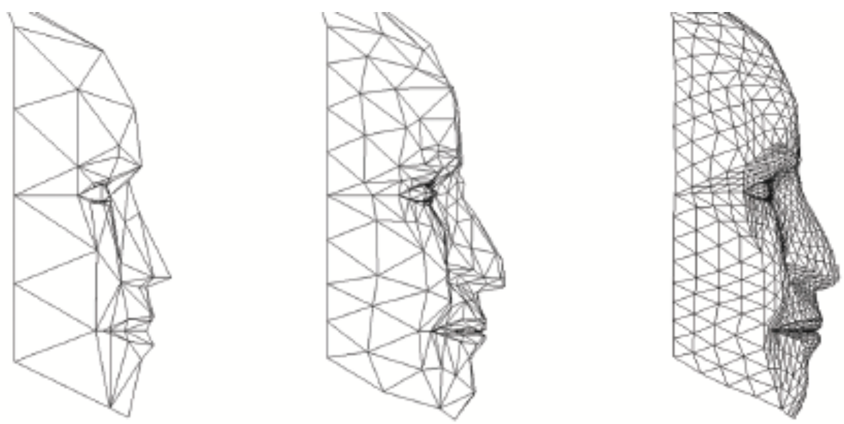
# Why triangulation?

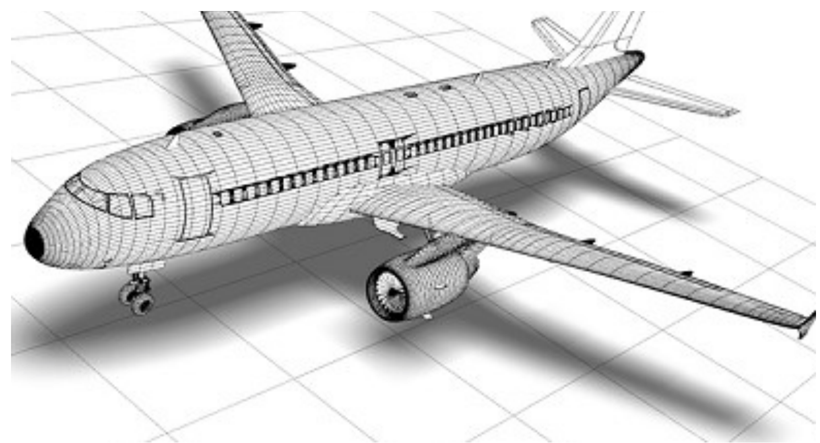
Partitioning into simpler shapes: technique for dealing with complexity

In 3D this is known as “meshing”

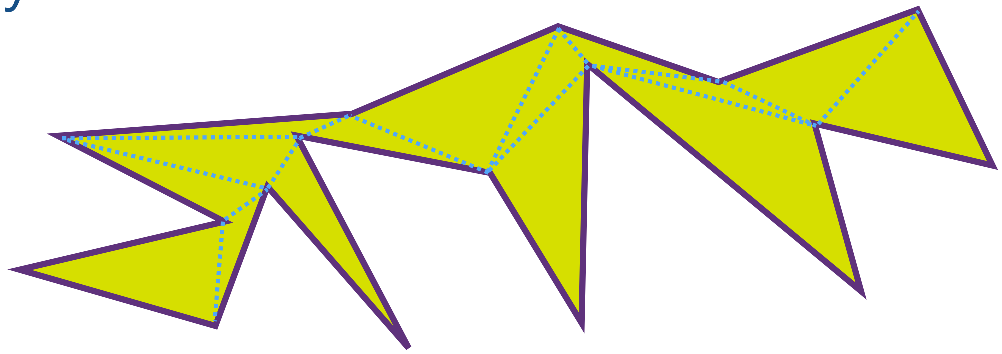
Triangulating a polygon is a simpler 2D version of the more general meshing problem.







# Does a triangulation always exist?



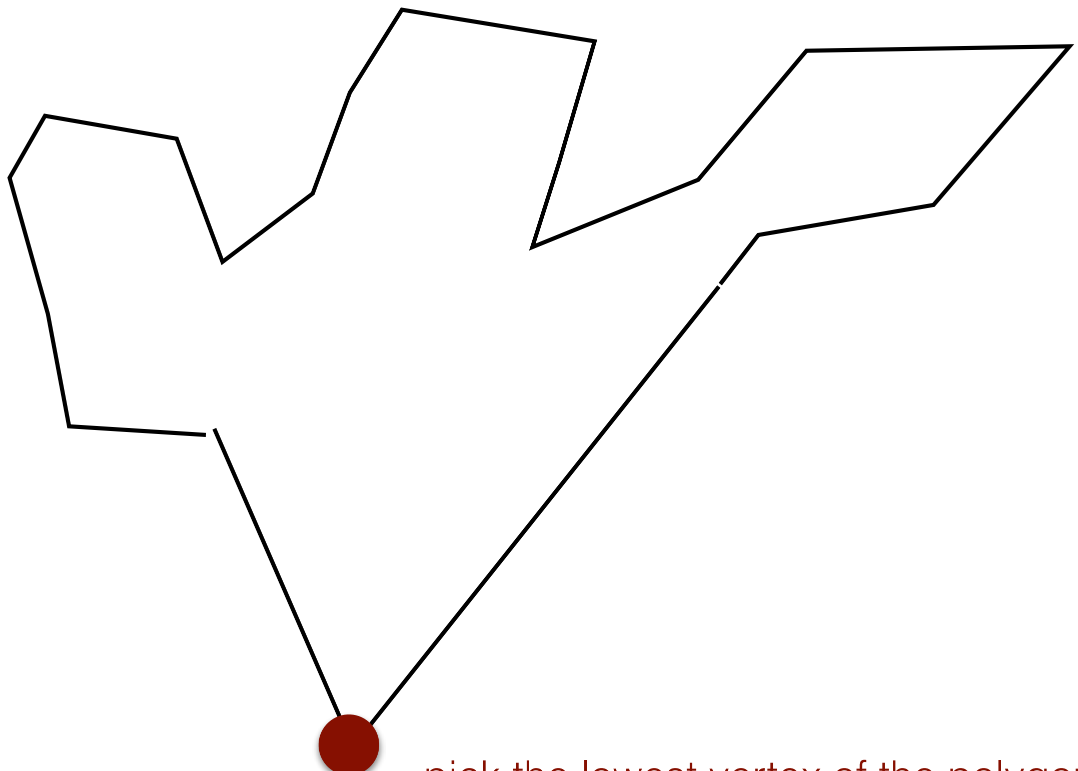
YES.

We can show the following:

- 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.

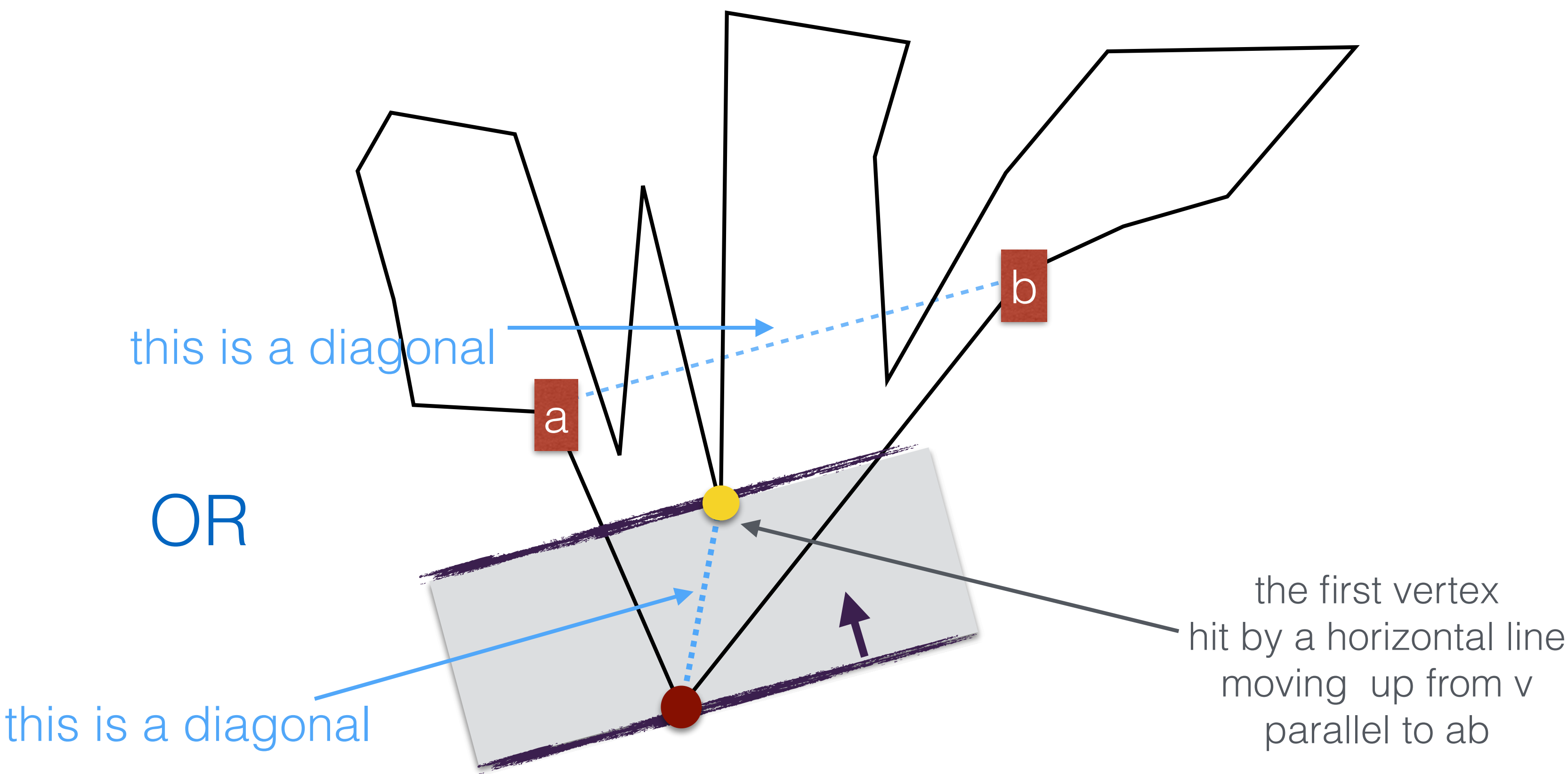
Theorem 1: Any simple polygon contains at least one convex vertex

the angle is <180



pick the lowest vertex of the polygon

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



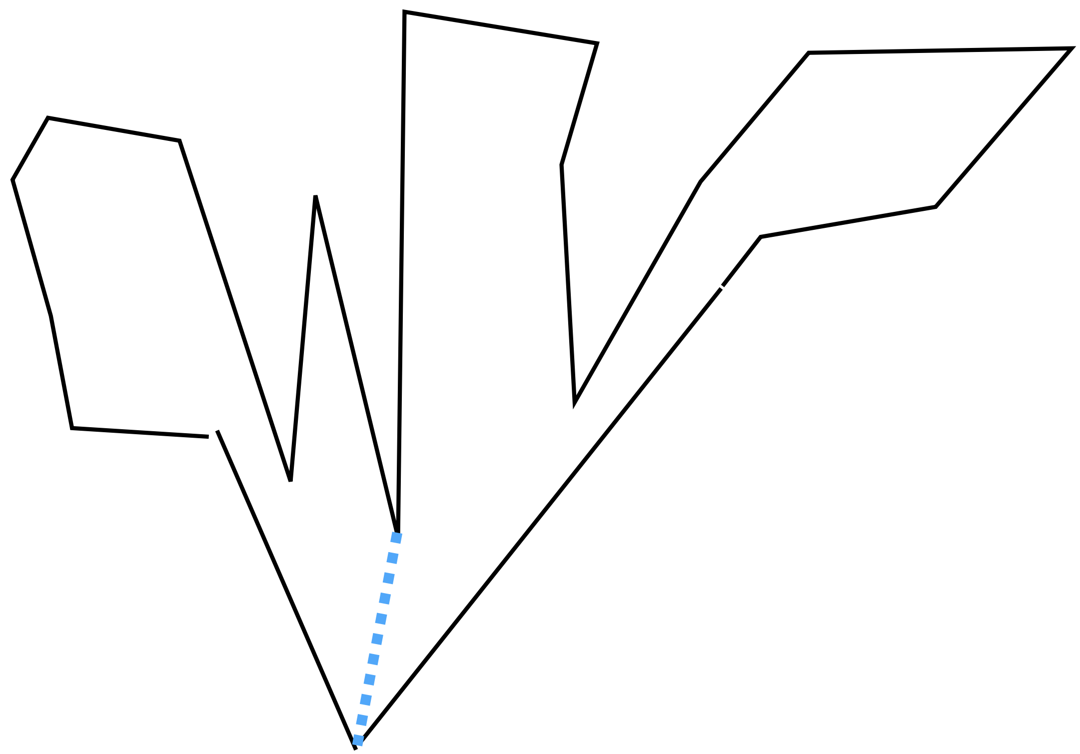
Theorem 3: Any polygon can be triangulated by adding diagonals

Proof: By induction on the size of the polygon

if n=3, holds trivially

Assume it holds for any k < n.

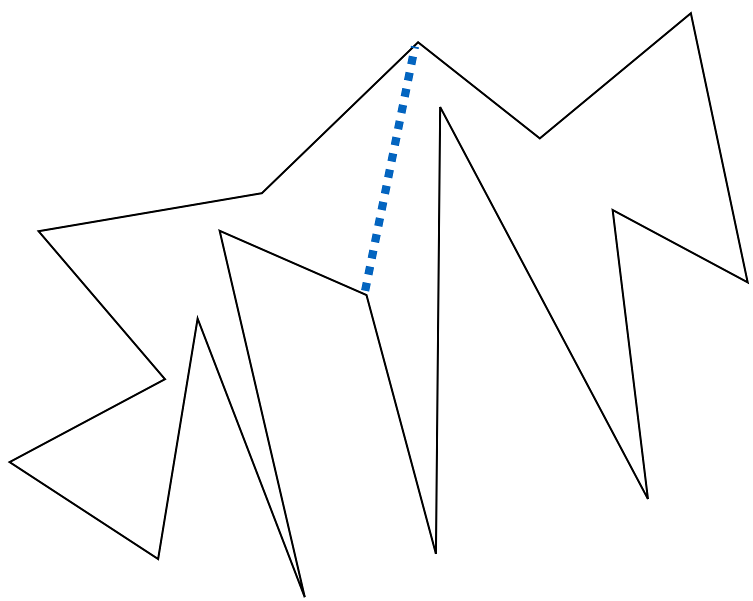
A diagonal must exist. It partition P into two polygons, each one has < n vertices, and can be triangulated by ind. hyp.



# Polygon triangulation Algorithm 1: Naive

// P is a polygon given as a vector of points (in ccw order along boundary)

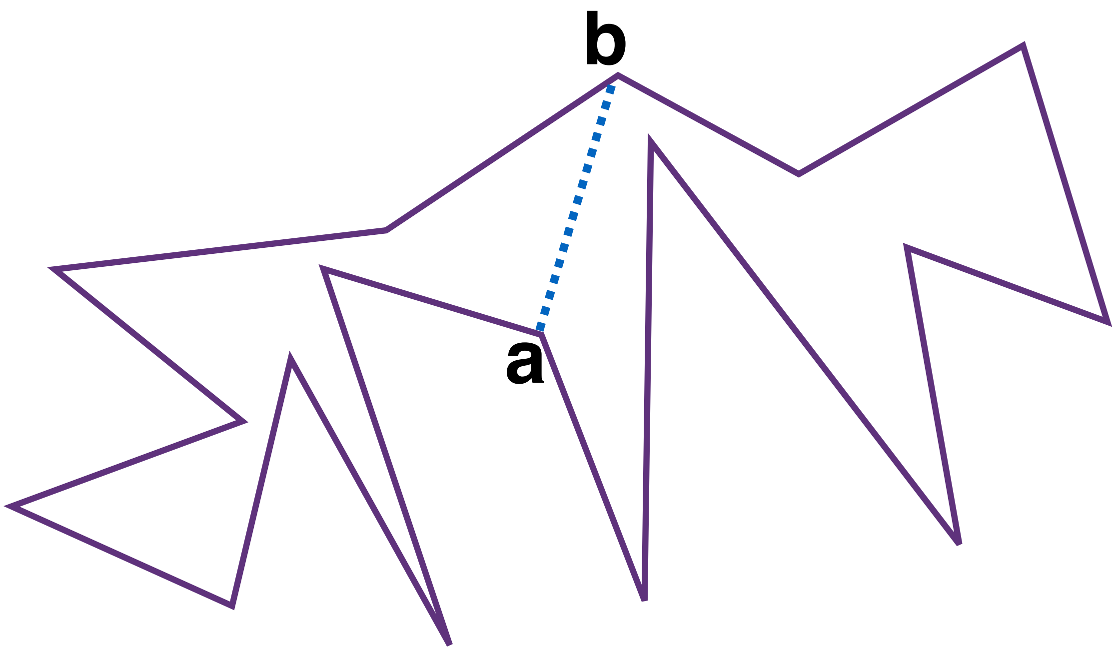
Idea: Find a diagonal, use it to partition P,  recurse on the resulting polygons



// return True if vertices $a, b$ of P form a diagonal

isDiagonal($a, b$, P)

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



let $i^+ = (i == (n-1))?0 : i+1$

```
// input: a, b are points in P,  let n be the size of P
// return true if (a,b) is diagonal
bool isDiagonal(a, b, P):
```

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

  //Check edge $(p_i, p_{i^+})$

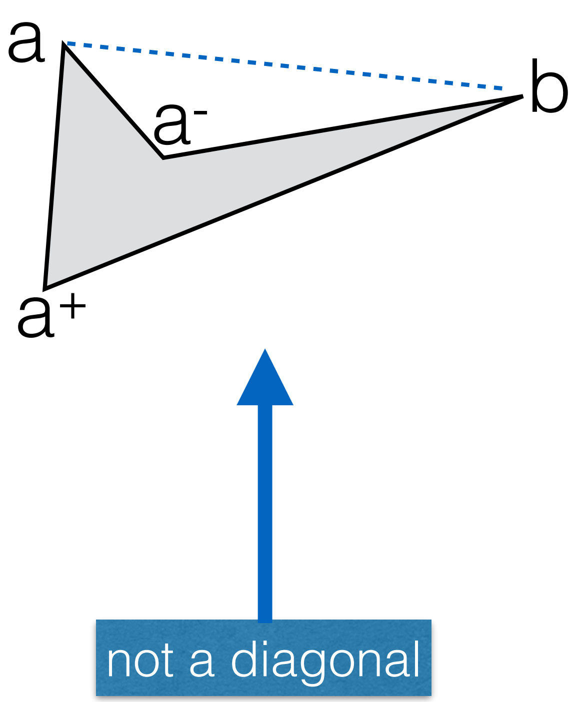
  - if $(p_i == a)$ OR $(p_i == b)$:  continue
  - if $(p_{i^+} == a)$ OR $(p_{i^+} == b)$: continue
  - if  intersect($a, b, p_i, p_{i^+}$): return False

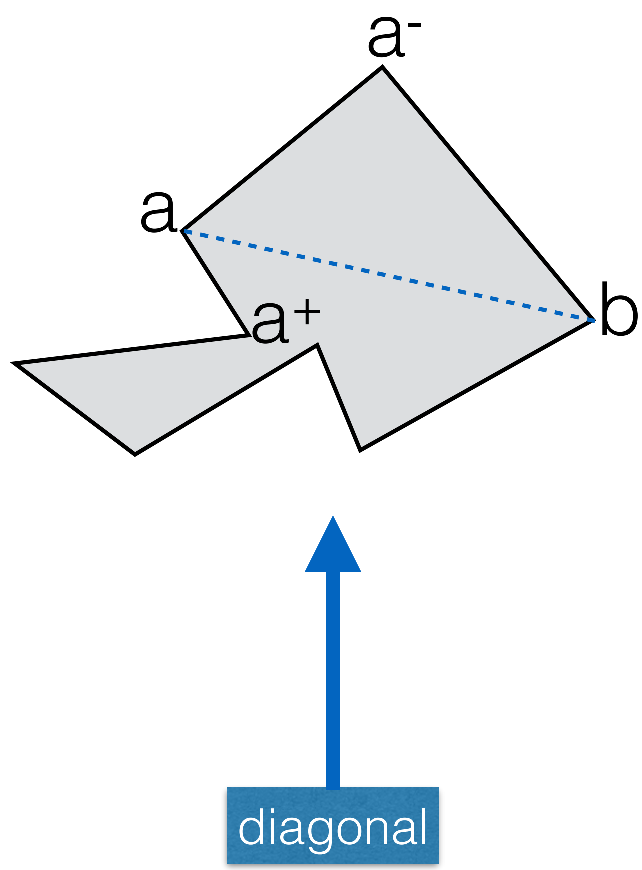
  //if we got here, we know that ab intersects no edge.

  //the only  thing left to check is whether it's inside or outside P

- return true if inside P, false if outside P

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



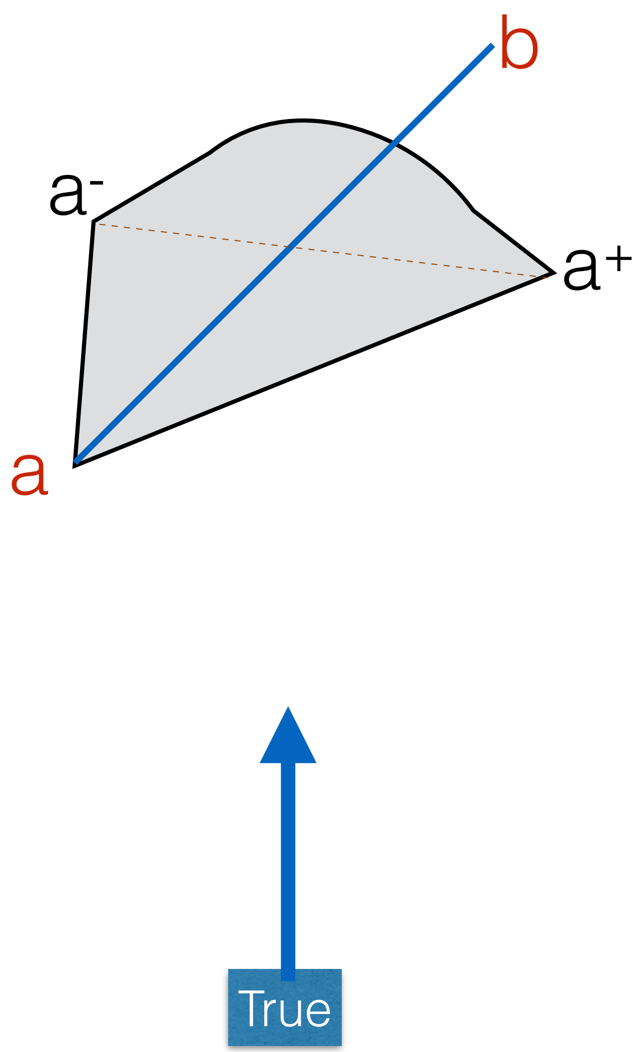



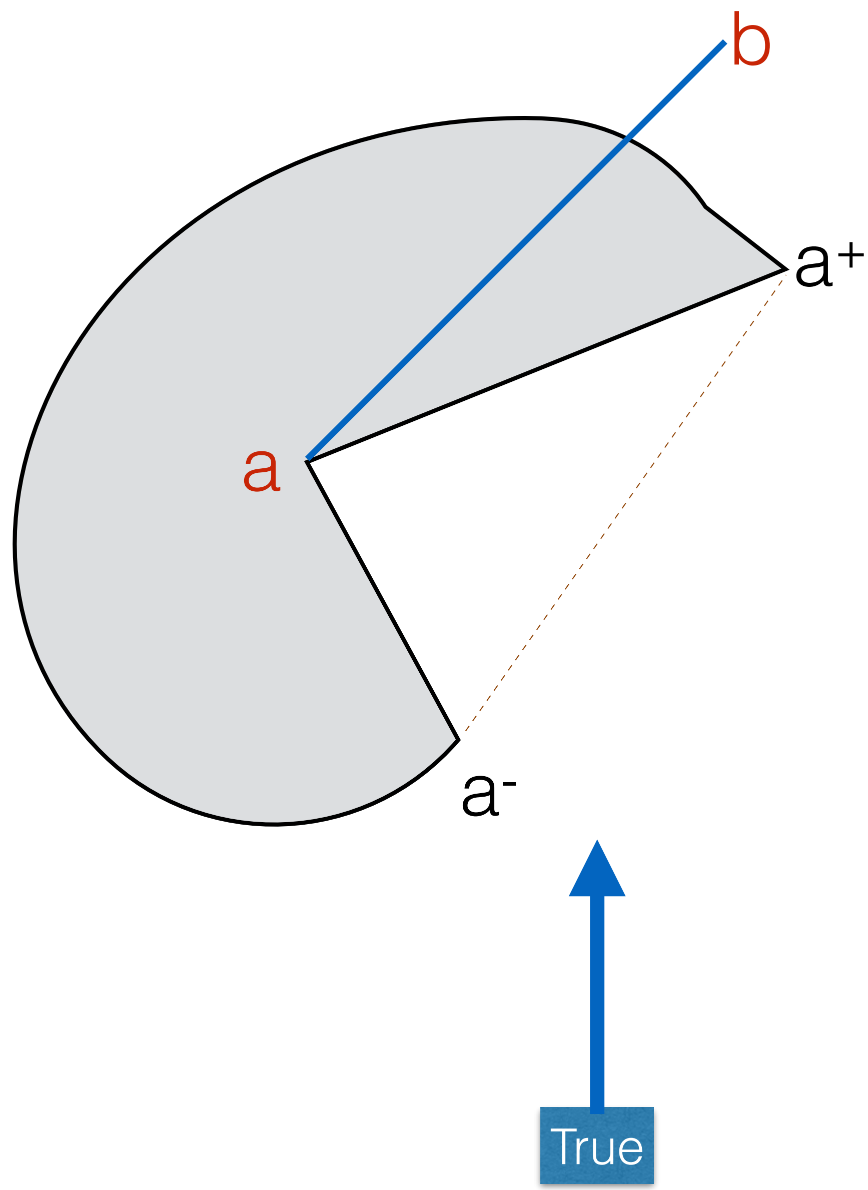

ab outside cone $a^{-}, a, a^{+}$

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

//return True if ab is in the cone determined by $a^-, a, a^+$

**bool InCone(a, b):**

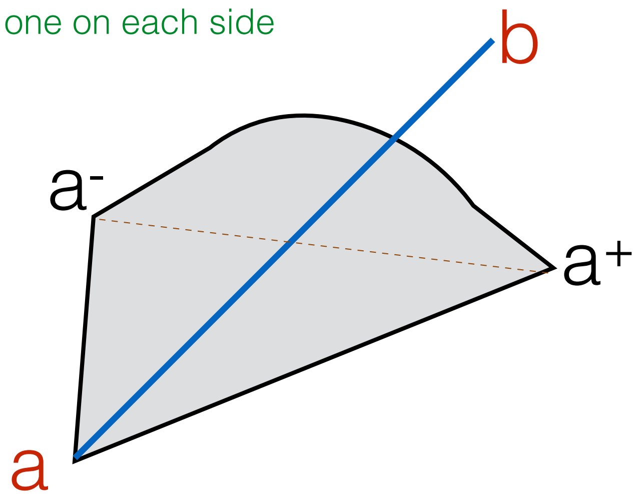


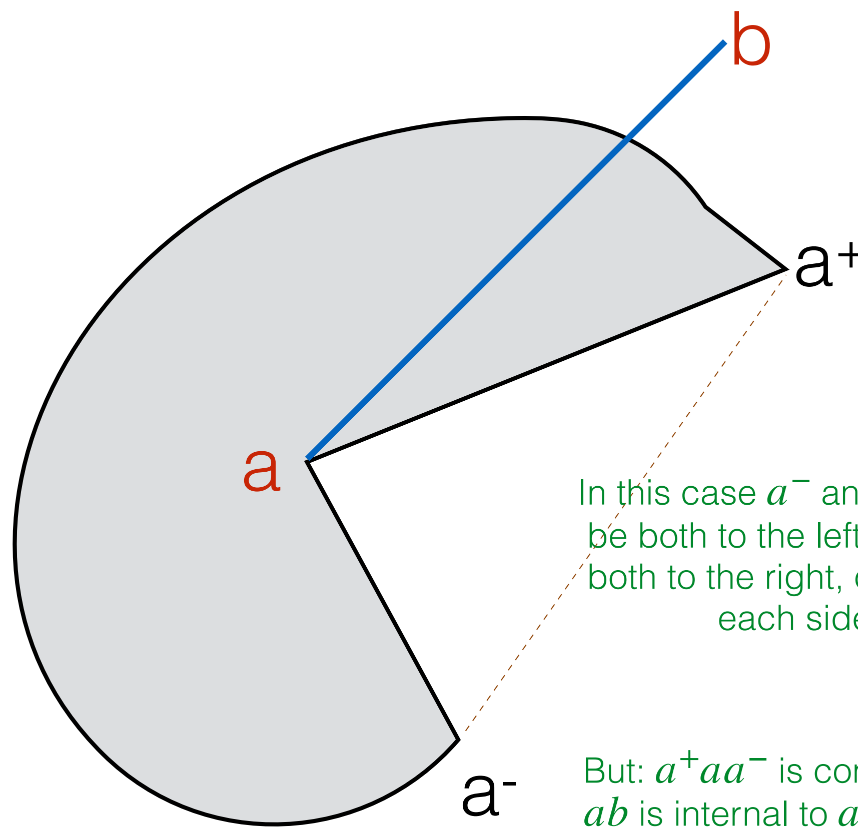



//return True if ab is in the cone determined by $a^{-}, a, a^{+}$

**bool InCone(a, b):**

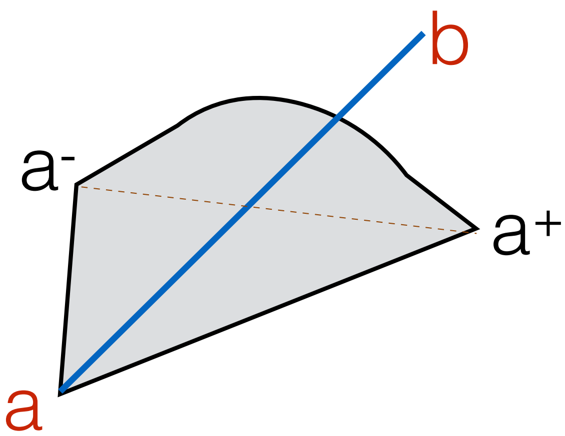
In this case $a^{-}$ and $a^{+}$ must be one on each side

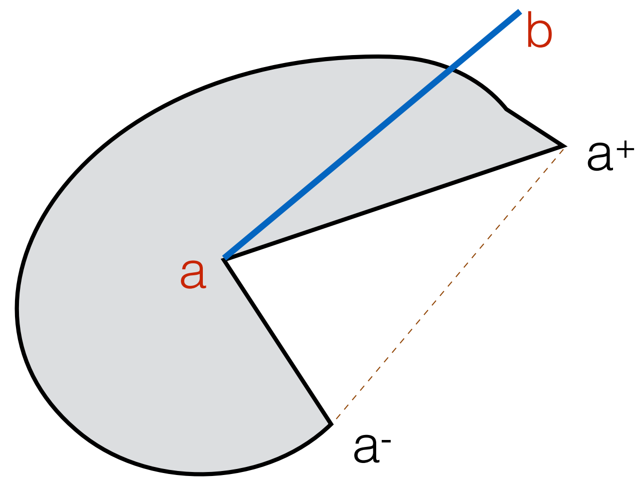



In this case $a^{-}$ and $a^{+}$ may be both to the left of ab, or both to the right, or one on each side

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




//return True if ab is in the cone determined by $a^-, a, a^+$

bool **InCone**(a, b, P)

- $a^- =$ point before a
- $a^+ =$ point after a

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

//if a is convex vertex

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

//else a  is reflex vertex

- return   !(  LeftOn$(b, a, a^-)$  and LeftOn$(a, b, a^+)$   )

# Putting it all together: Is ab a diagonal?

//input: a, b are points in P

//return true if (a,b) is diagonal

bool isDiagonal(a,b, P):

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

  //Checking edge $(p_i, p_{(i+1)\%n})$

  - let $i^+ = (i == (n-1))?0 : i+1$
  - if $(p_i == a)$ OR $(p_i == b)$: continue
  - if $(p_{(i+1) mod\ n} == a)$ OR $(p_{(i+1) mod\ n} == b)$: continue
  - if intersect($a, b, p_i, p_{(i+1)\%n}$): return False

O(n)

  //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

O(1)
- return inCone(a, b, P) ~~and inCone(b,a,P)~~ //only one necessary

==> Can check if an edge(a,b) is a diagonal of P in O(n) time

# More efficient

//input: a, b are points in P

//return true if (a,b) is diagonal

bool isDiagonal(a,b, P):

O(1)
- if !(inCone(a, b, P) ~~and inCone(b,a,P))~~ : return false

check this first

- for i=0; i< n; i++
  - let $i^+ = (i == (n-1))?0 : i+1$
  - if $(p_i == a)$ OR $(p_i == b)$: continue
  - if $(p_{(i+1) mod\ n} == a)$ OR $(p_{(i+1) mod\ n} == b)$: continue
  - if intersect($a, b, p_i, p_{(i+1)\%n}$): return False .

O(n)

- return true //if we got here, we know that ab intersects no edge

## So we know how to check if a segment is a diagonal, but how to find a diagonal?

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

$O(n^3)$

We can use this to triangulate

# 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, partition the polygon and 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 $O(n^4)$

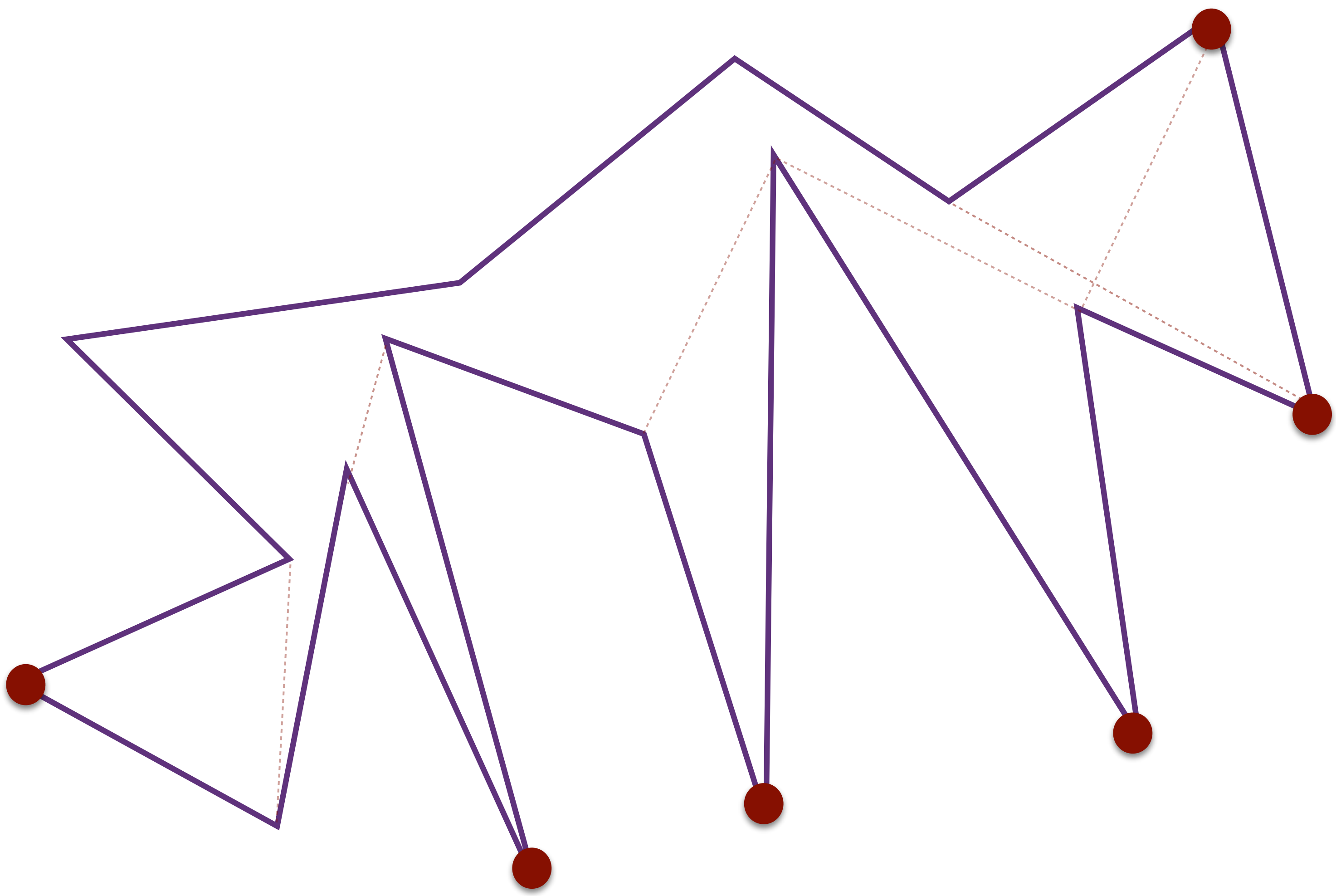
- **Algorithm 2:** Triangulation by *smartly* finding diagonals
  - A diagonal can be found in $O(n)$ time (using the proof that a diagonal exists)
  - Idea: Find a diagonal, output it, recurse.
  - $O(n^2)$

# Algorithm 3: Triangulation by finding ears

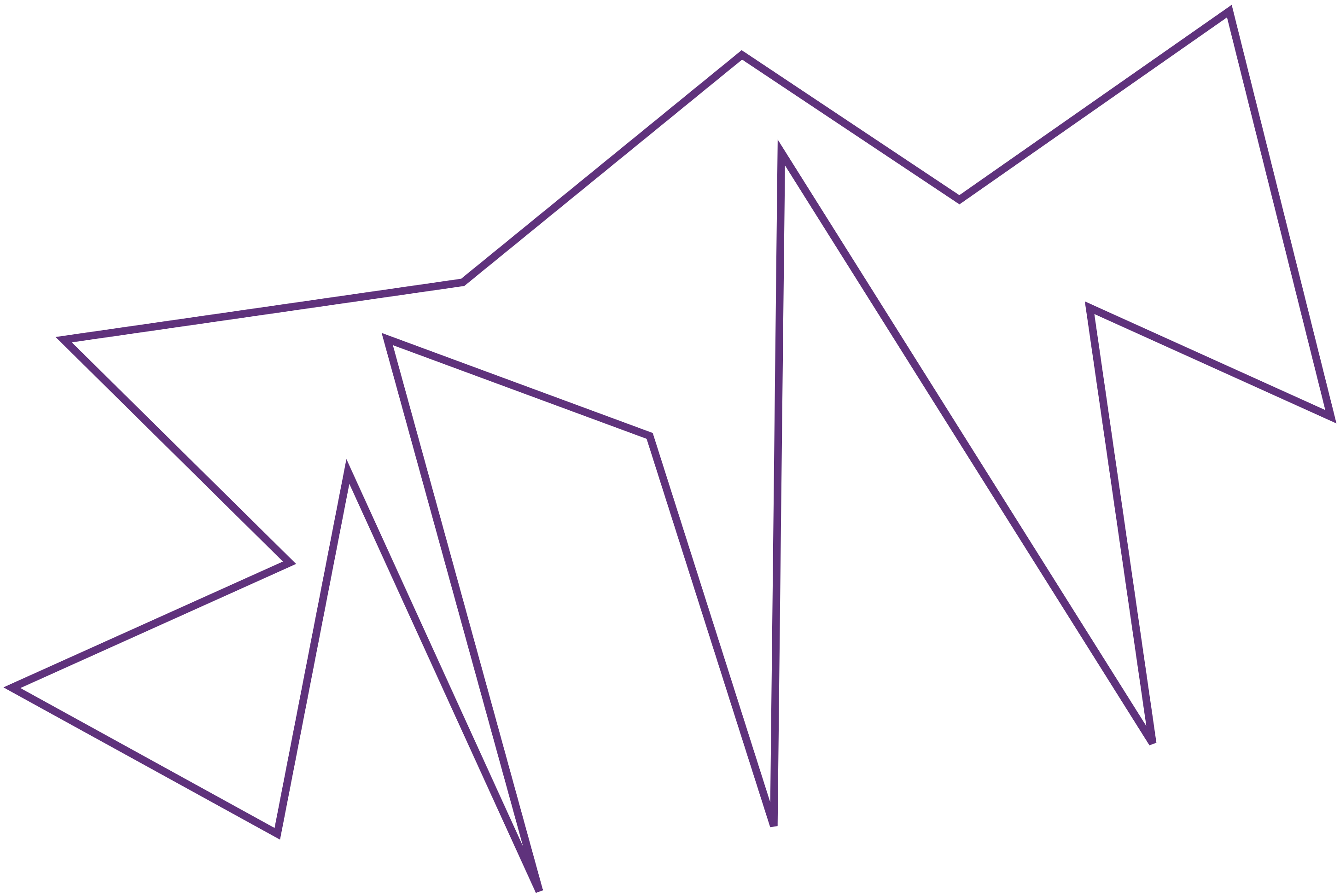

A bat-eared fox peeks from the grass in Hwange National Park, Zimbabwe.
PHOTOGRAPH BY ROY TOFT, NAT GEO IMAGE COLLECTION

# Definition

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

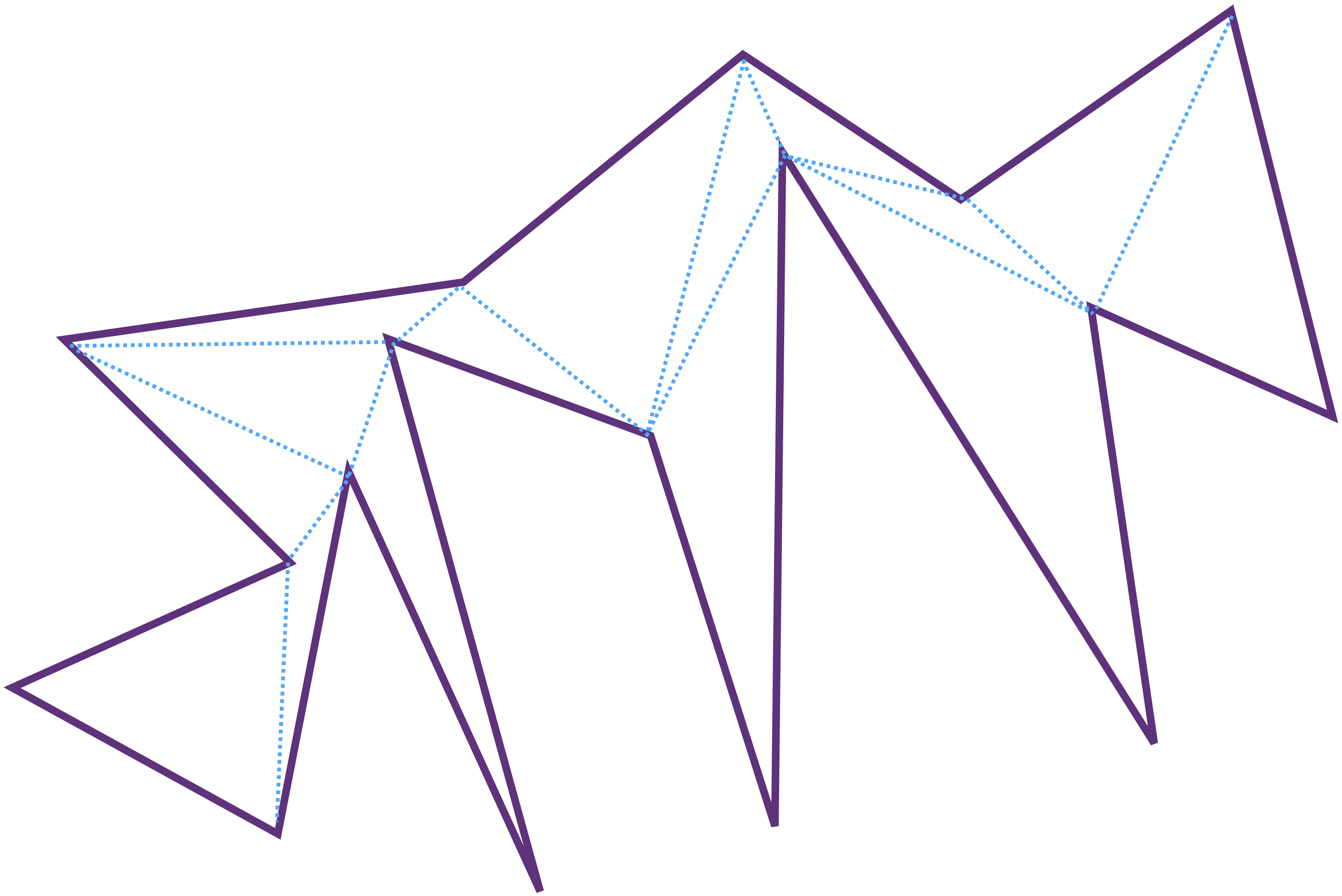


Theorem: Any simple polygon has at least two ears.



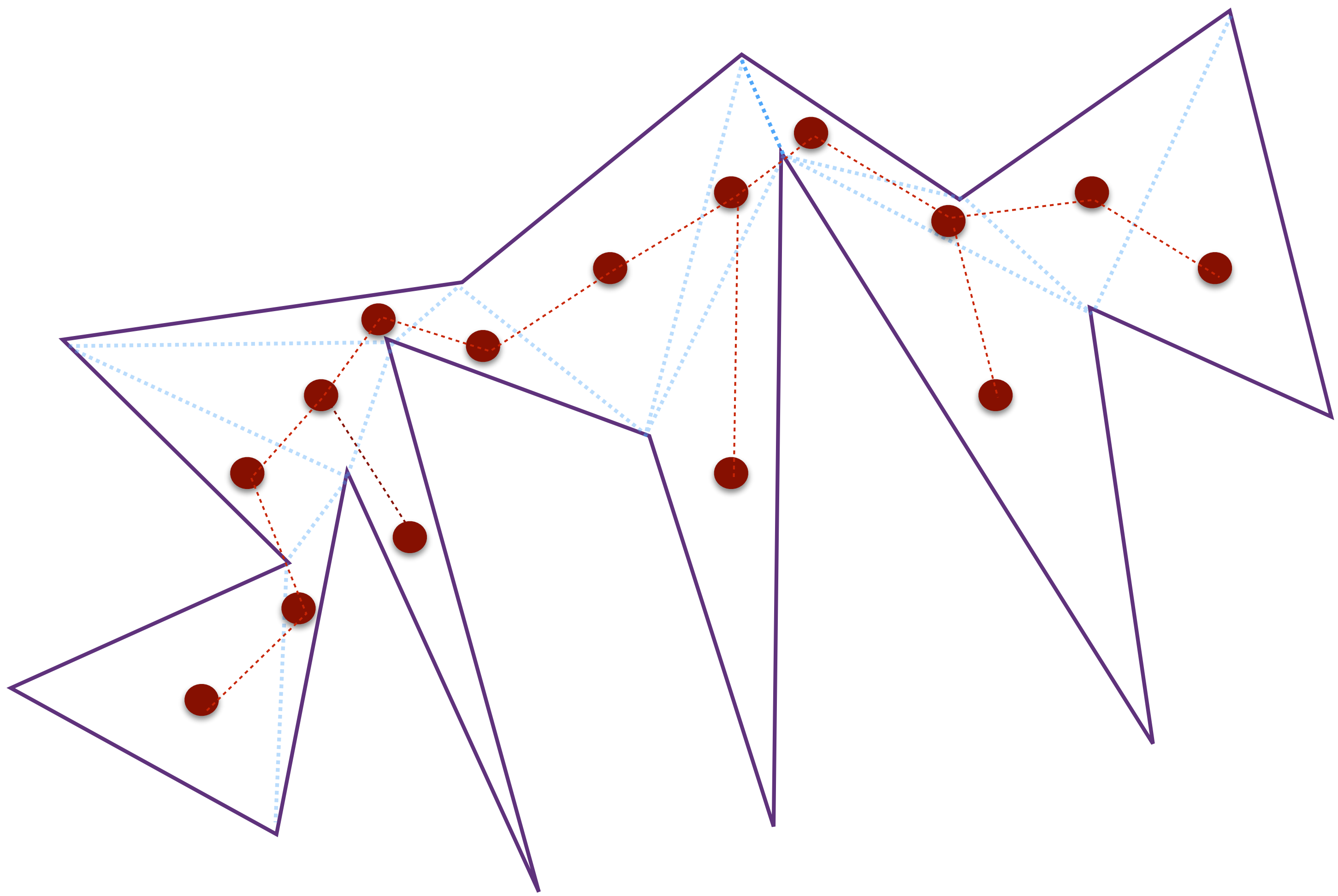
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P.



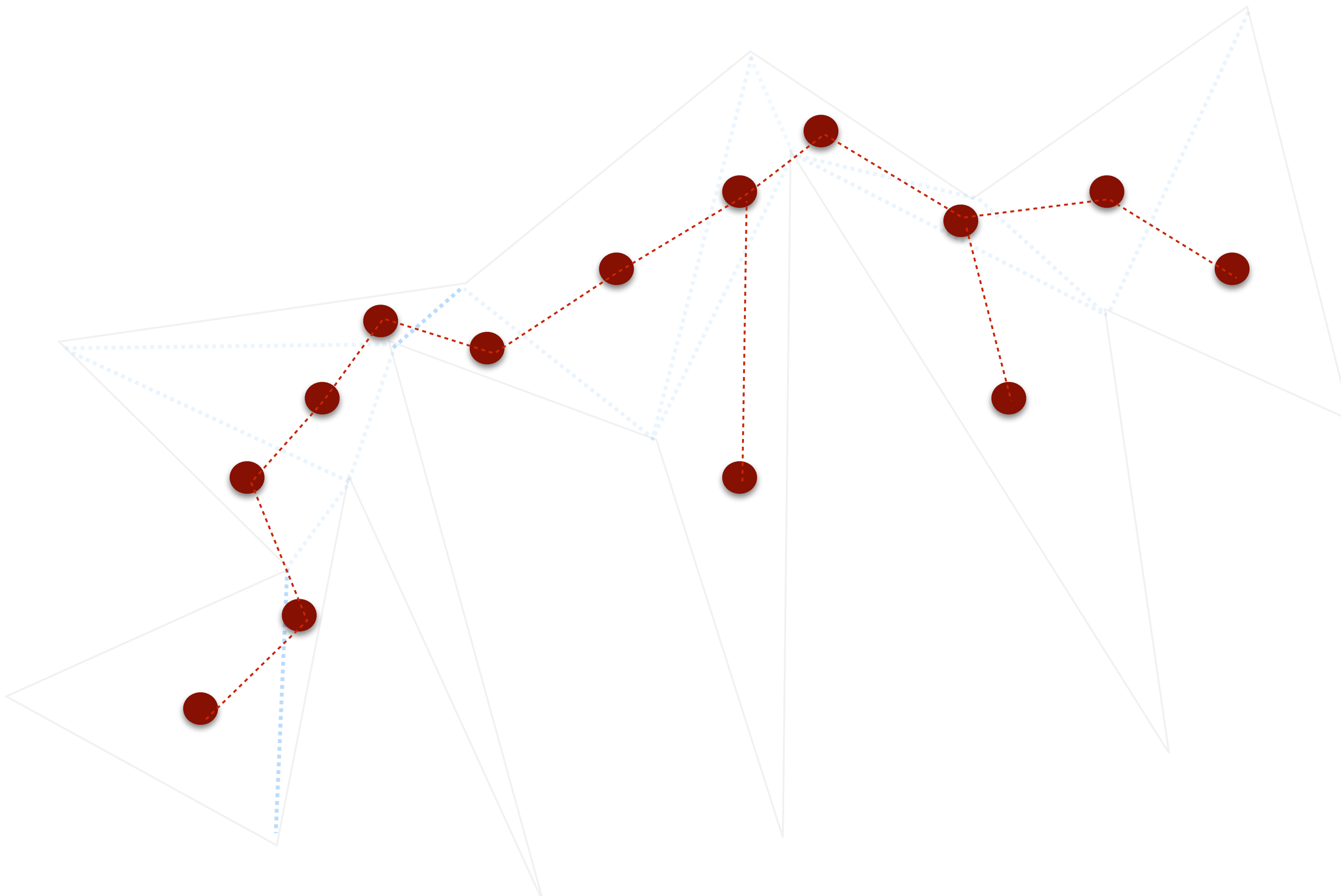
Theorem: Any simple polygon has at least two ears.

Proof: Triangulate P. Consider the dual graph.



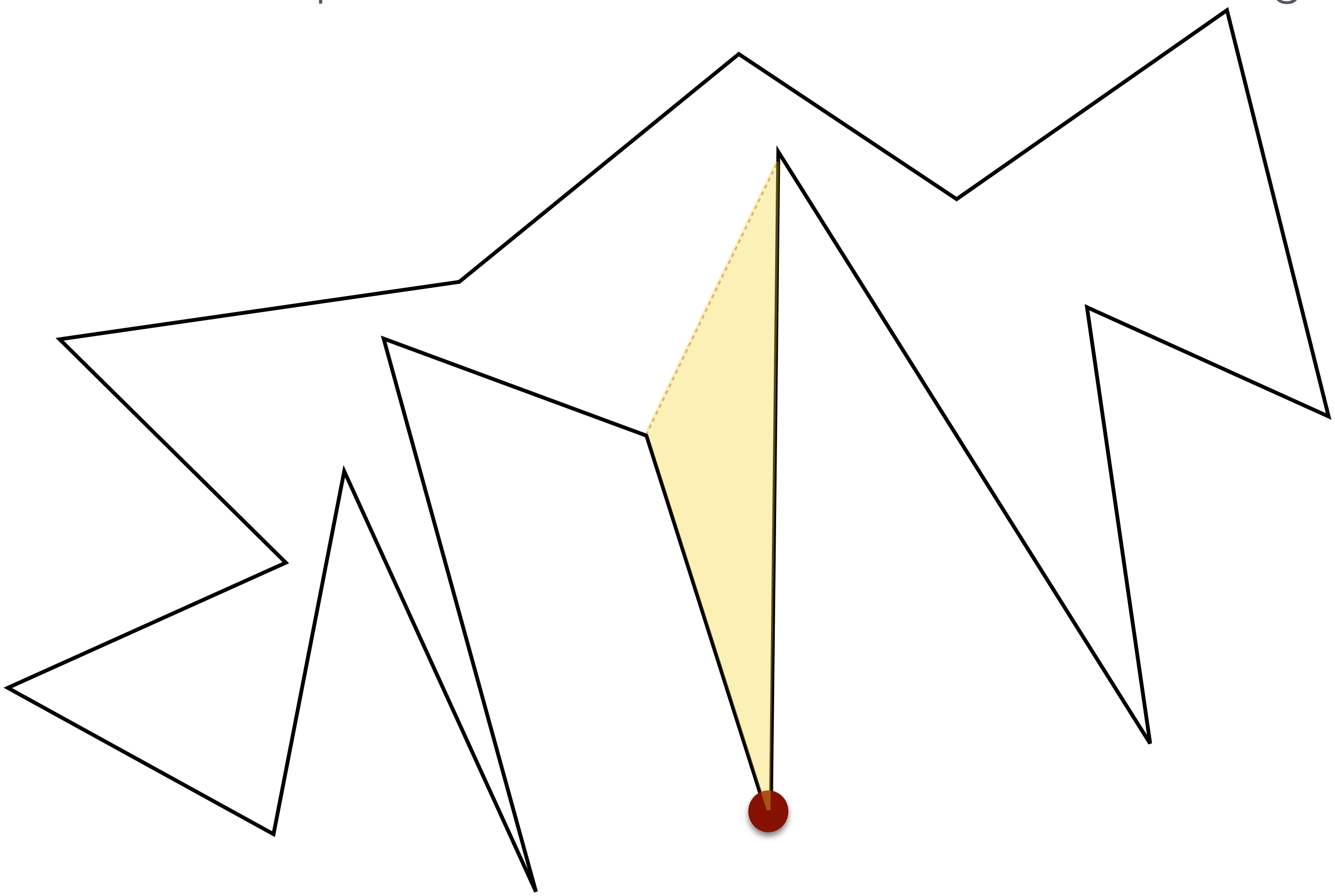
## Theorem: Any simple polygon has at least two ears.

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 vertex p, determine if it's an ear
- When find a ear p: remove it and recurse on the remaining P



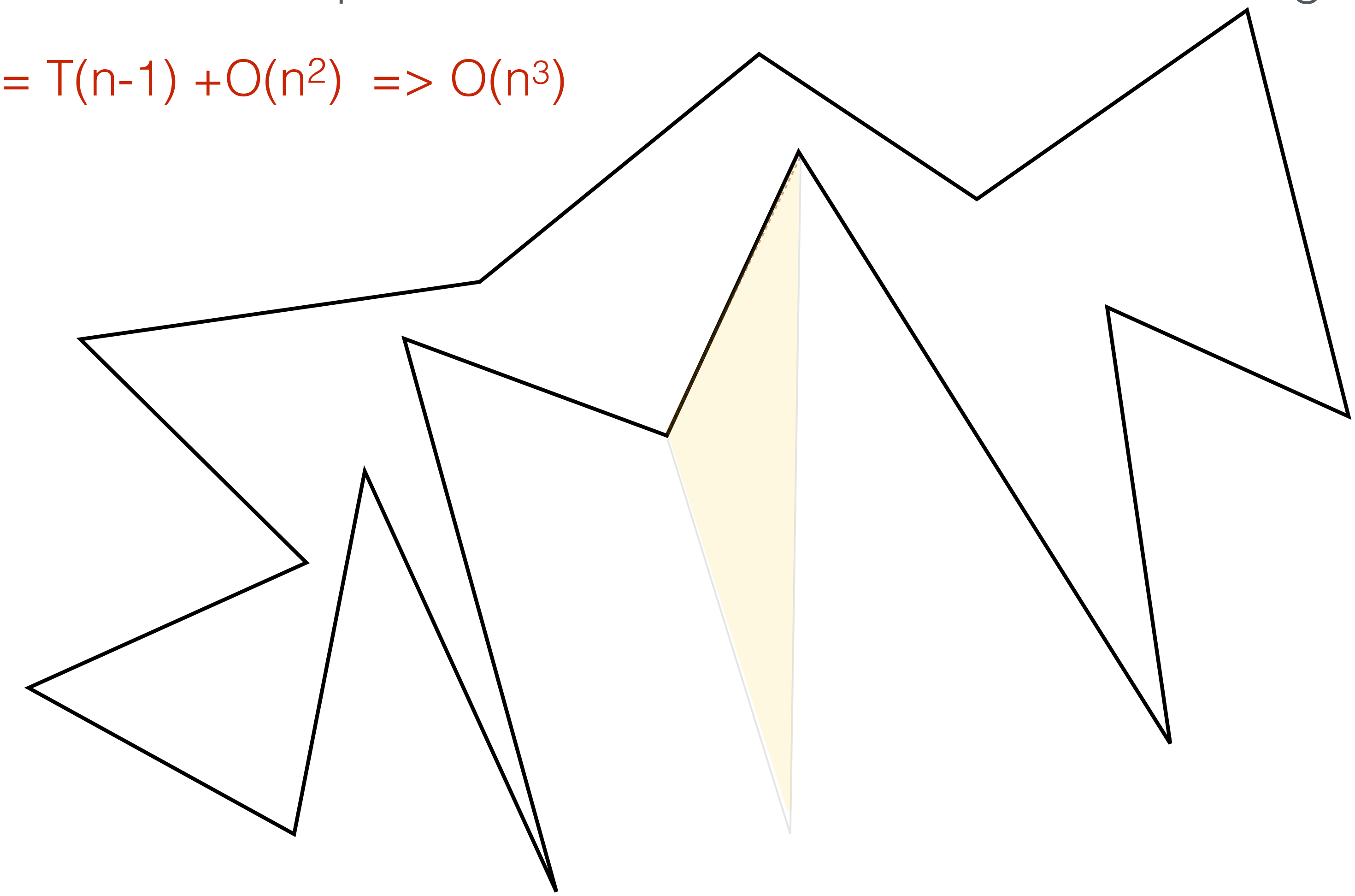
# Algorithm 3: Triangulation by finding ears

O(n)

O(n) • Traverse P and for each vertex p, determine if it's an ear

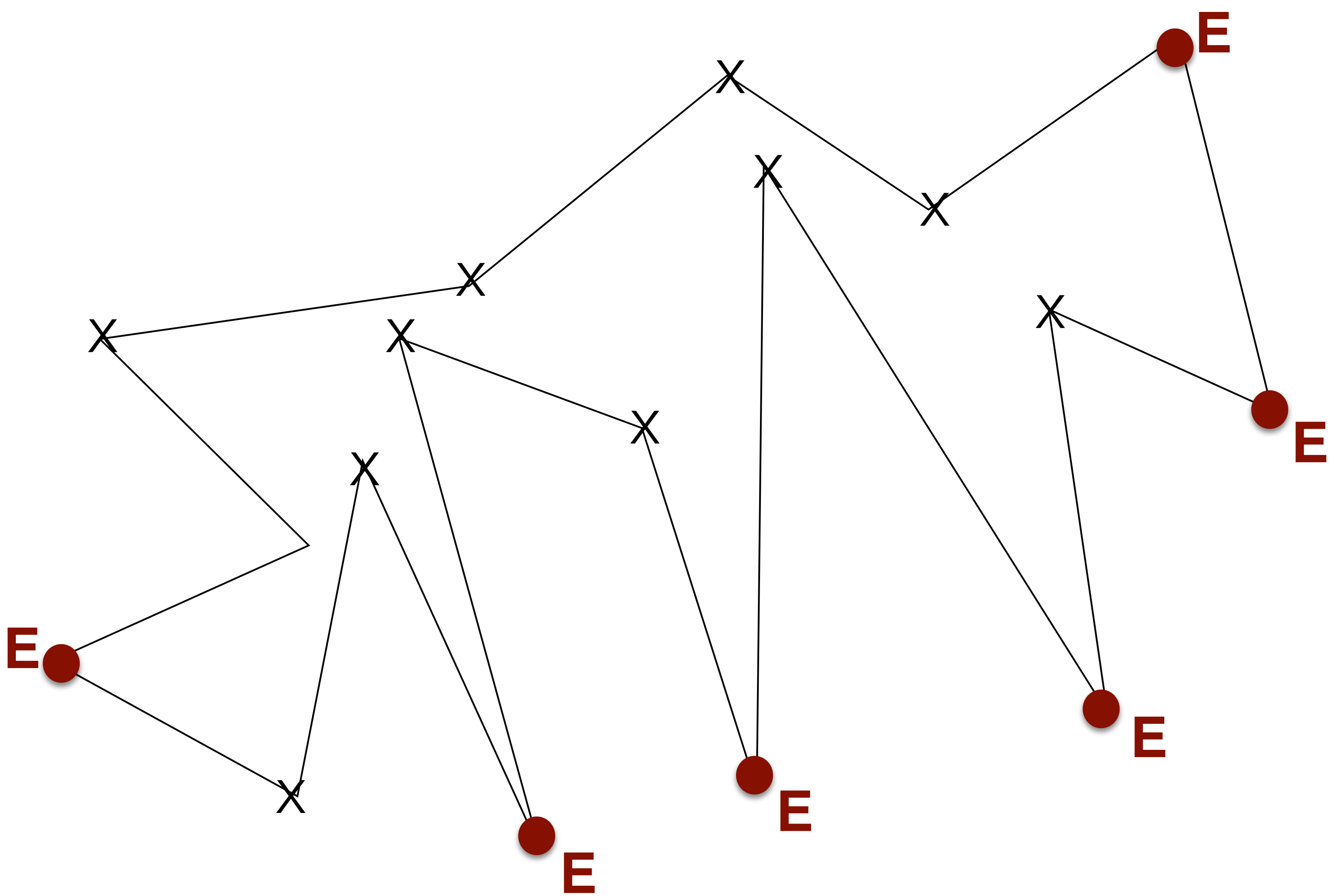
• When find a ear p: remove it and recurse on the remaining P

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



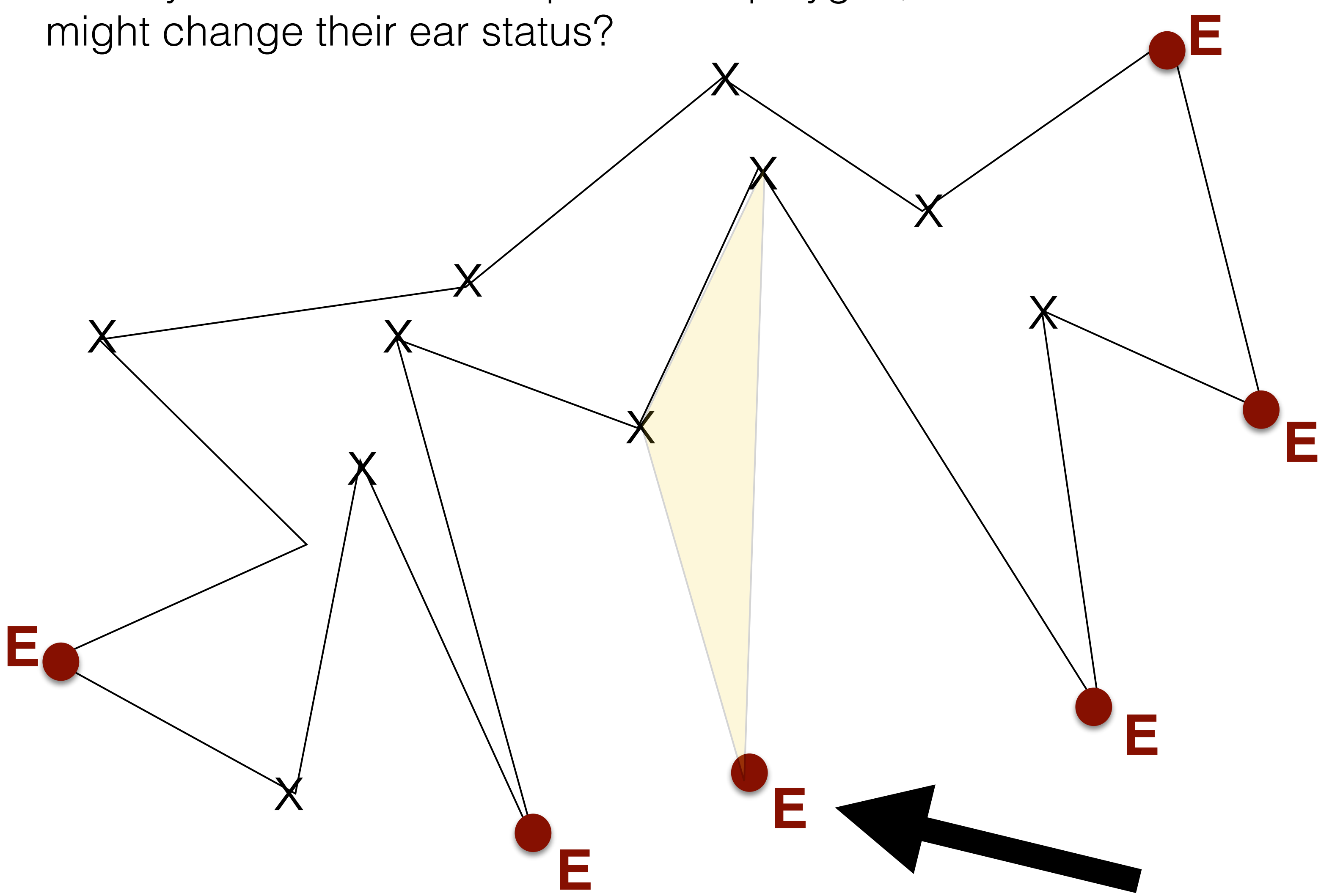
# Algorithm 4: Improved ear removal

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



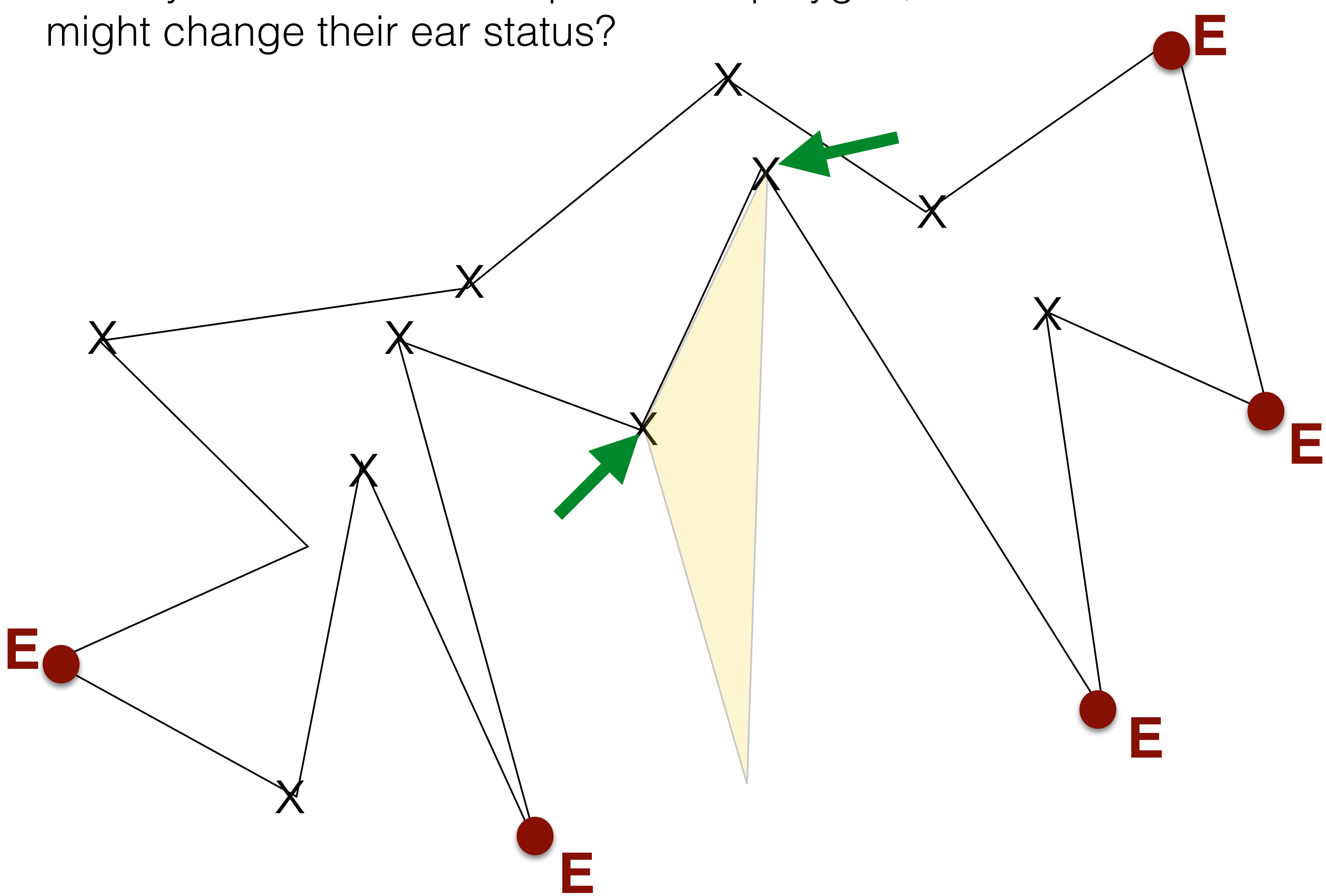
# Algorithm 4: Improved ear removal

- **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?



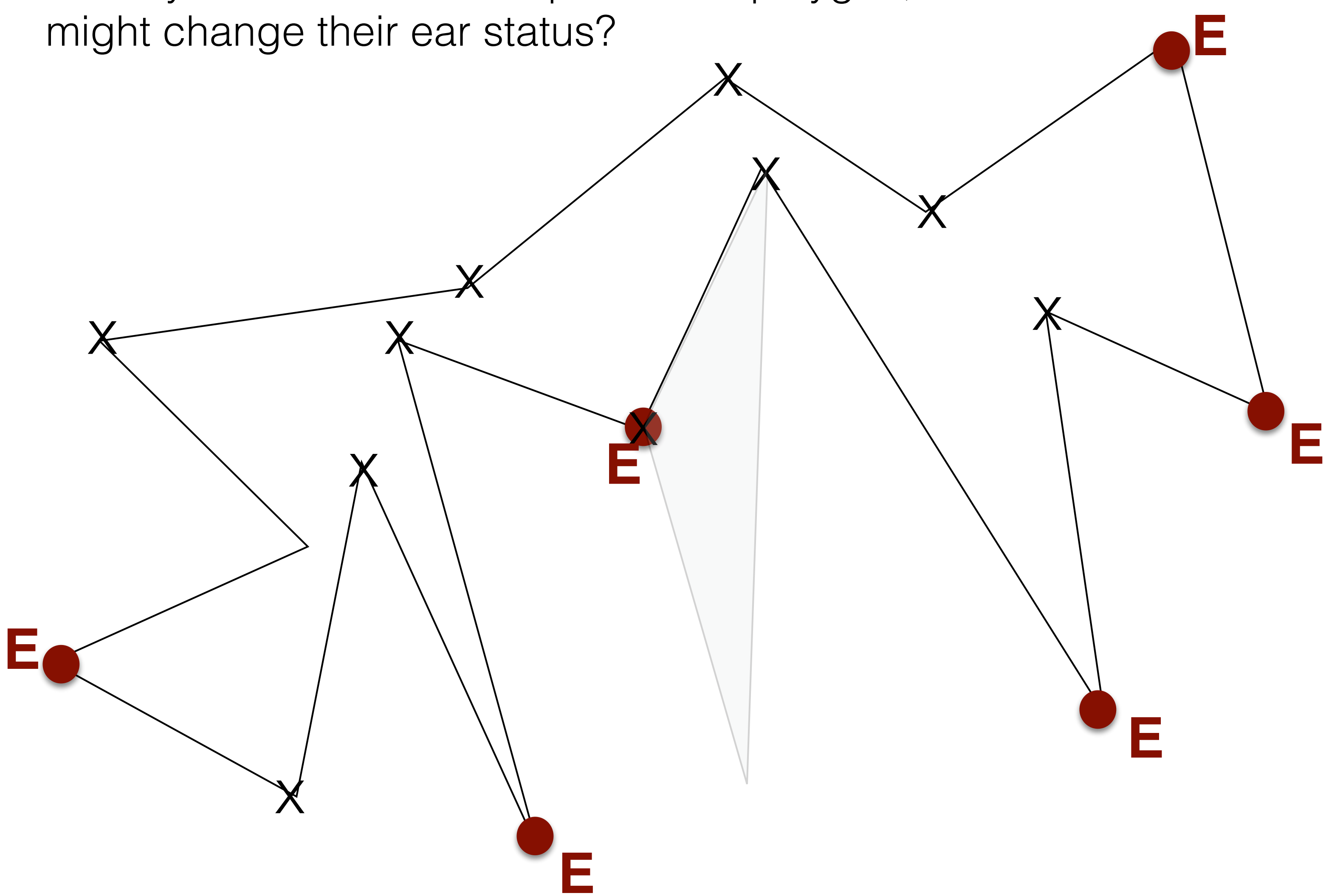
# Algorithm 4: Improved ear removal

- **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?



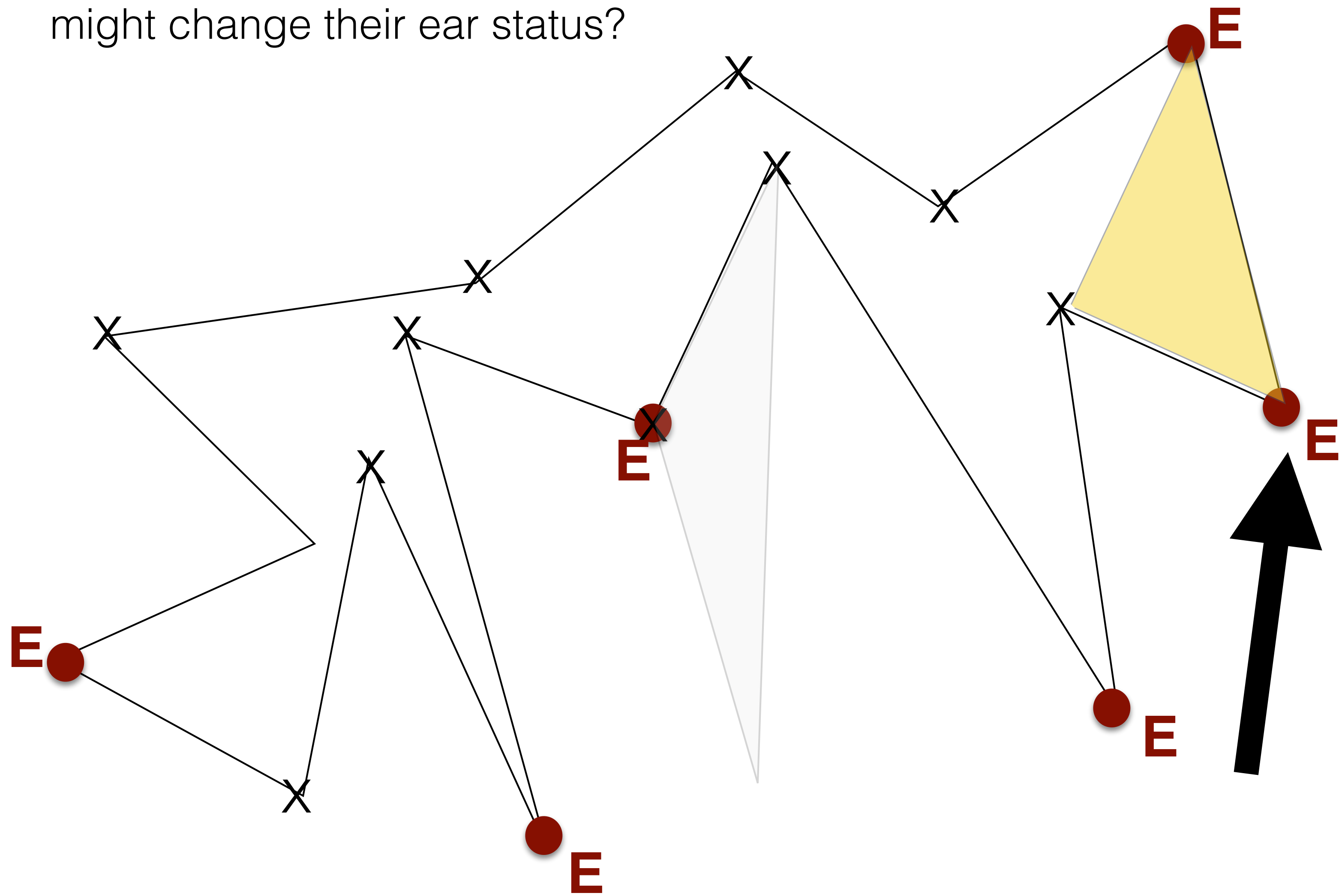
# Algorithm 4: Improved ear removal

- **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?



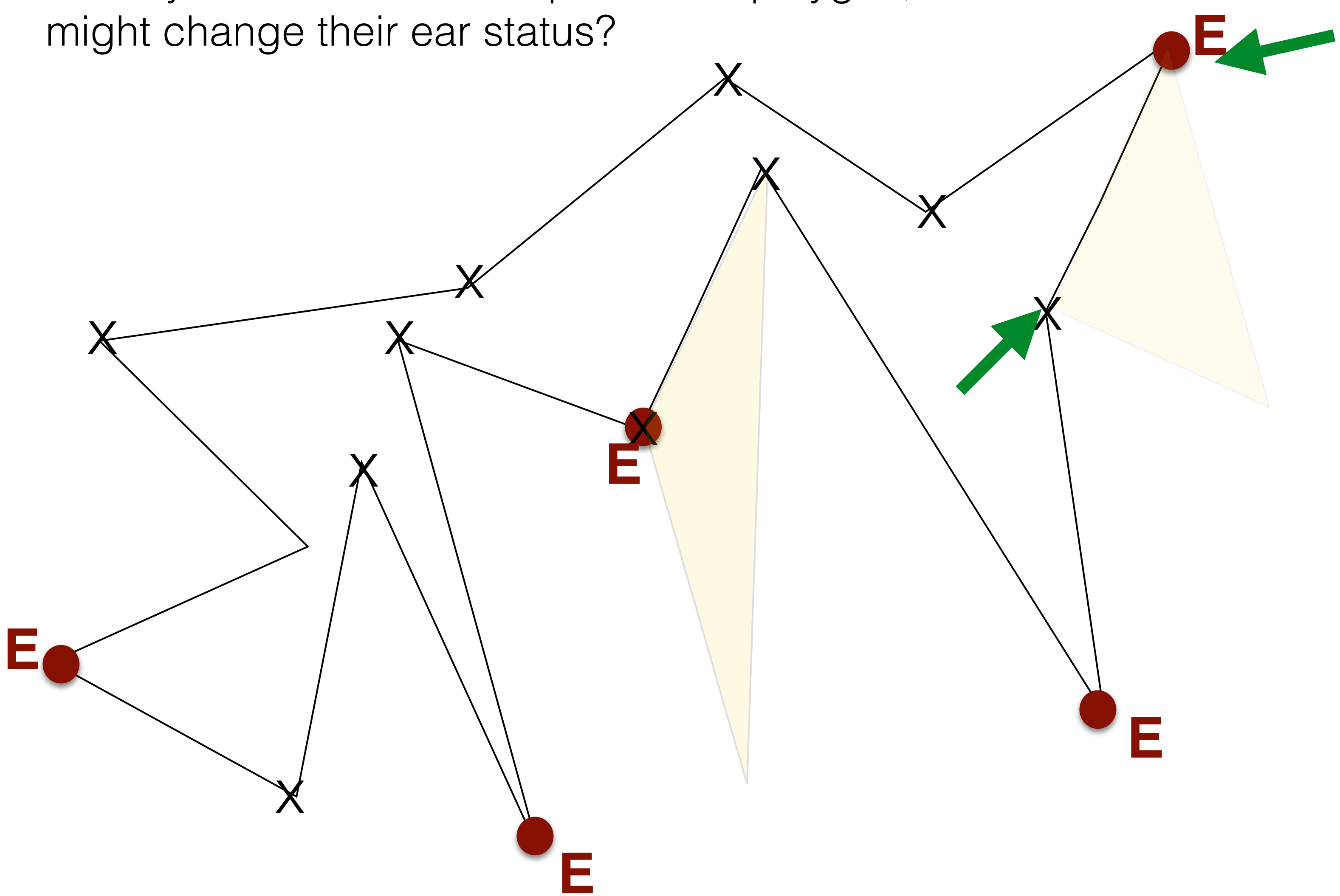
# Algorithm 4: Improved ear removal

- **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?



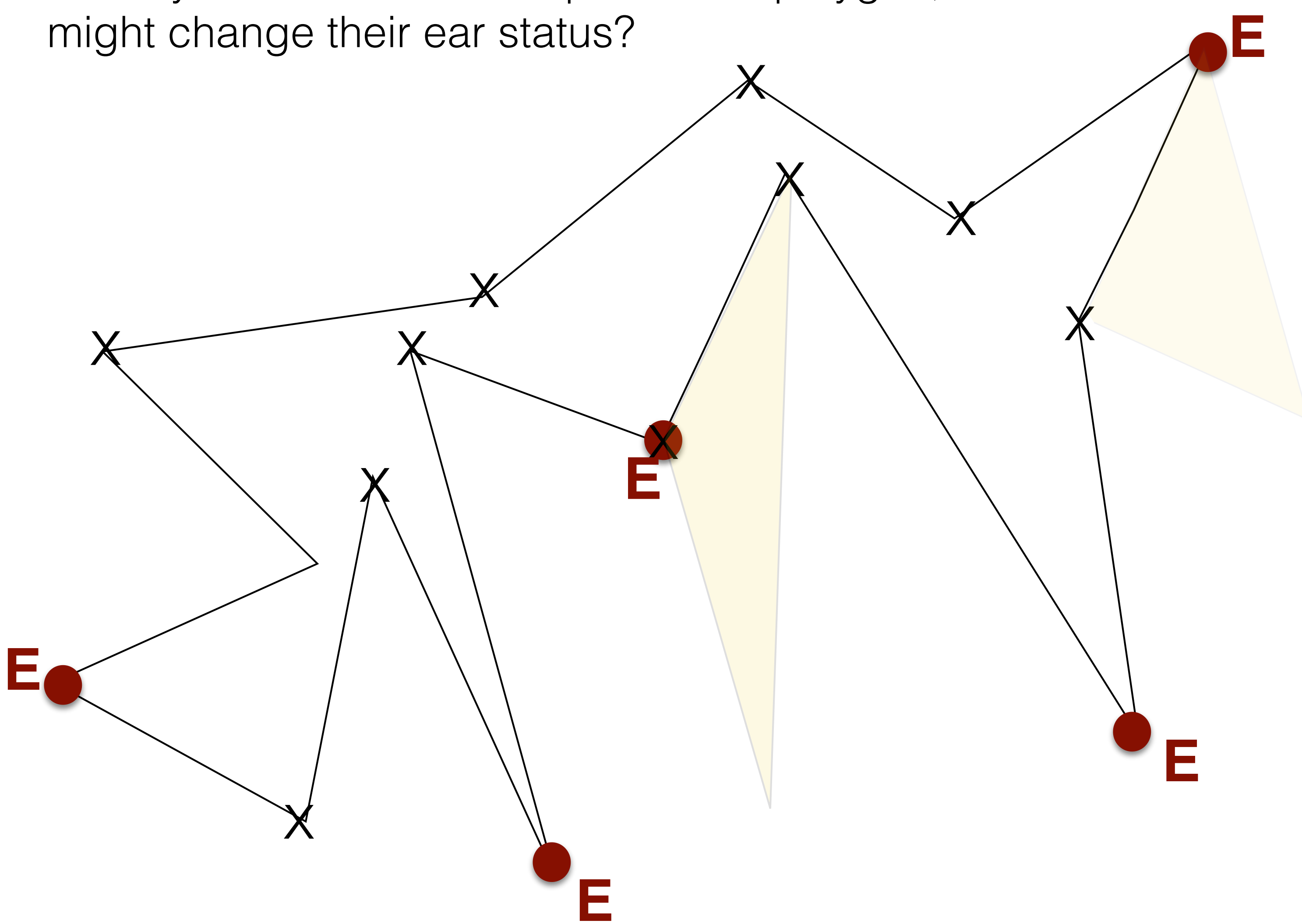
# Algorithm 4: Improved ear removal

- **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?



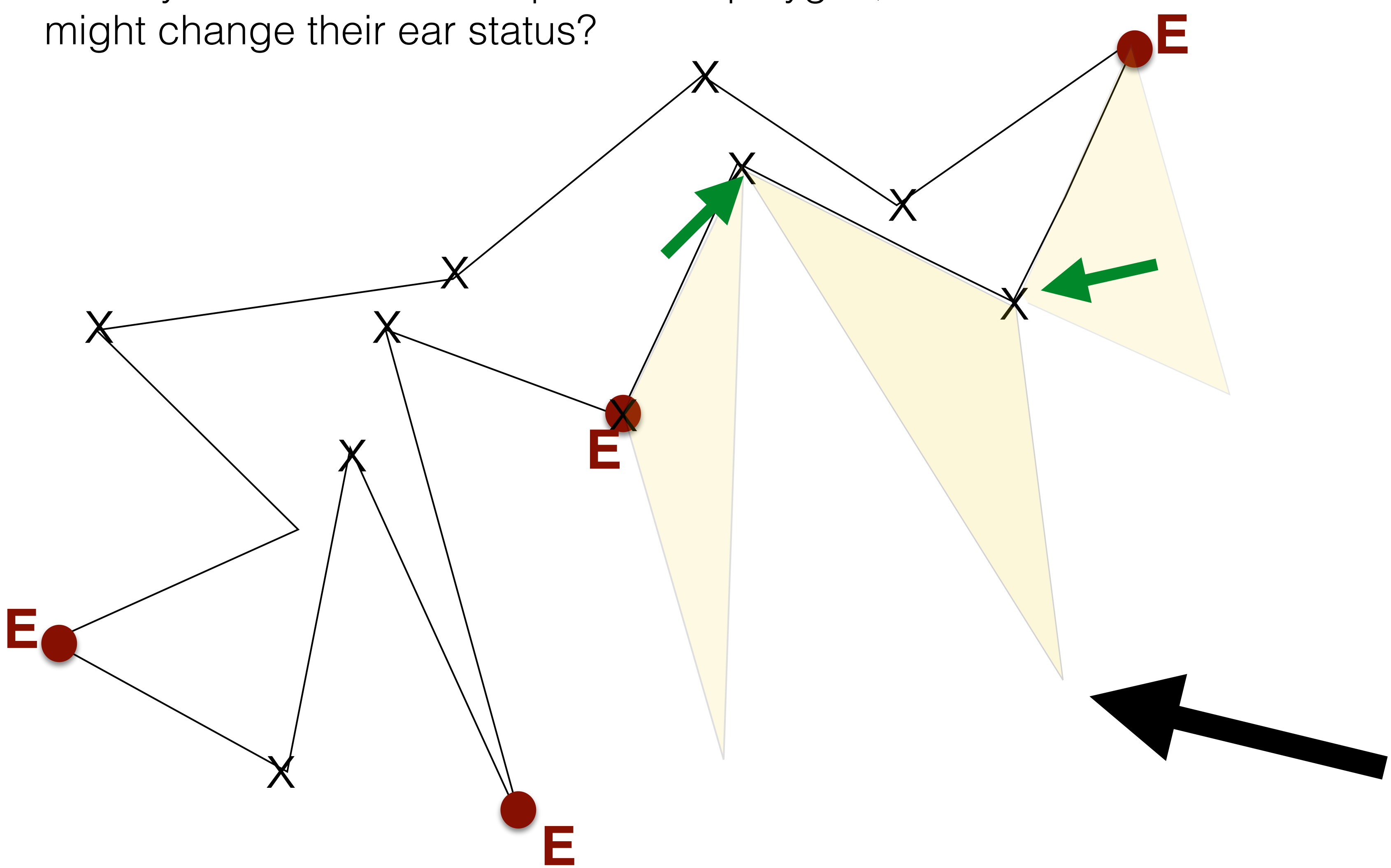
# Algorithm 4: Improved ear removal

- **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?



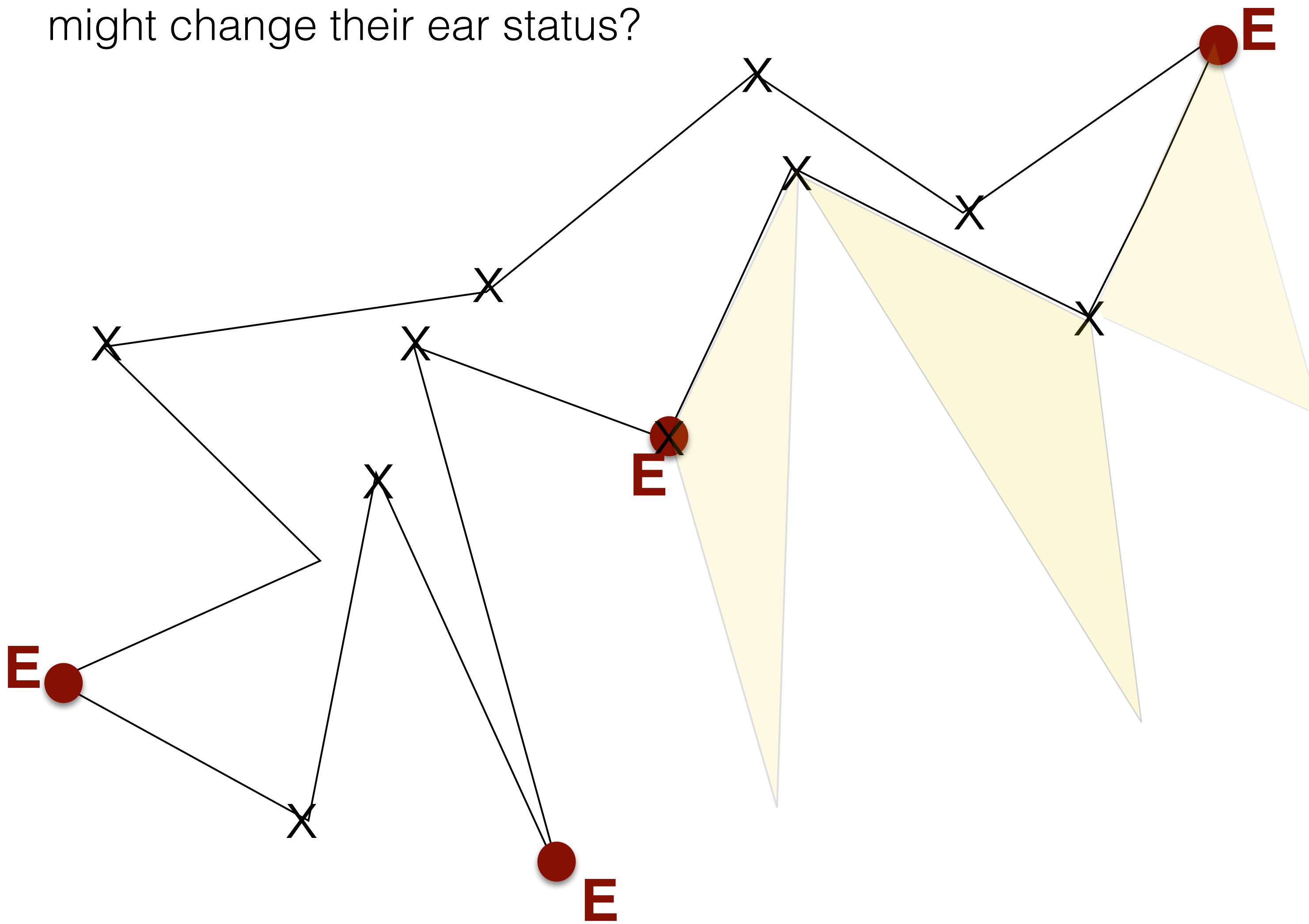
# Algorithm 4: Improved ear removal

- **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?



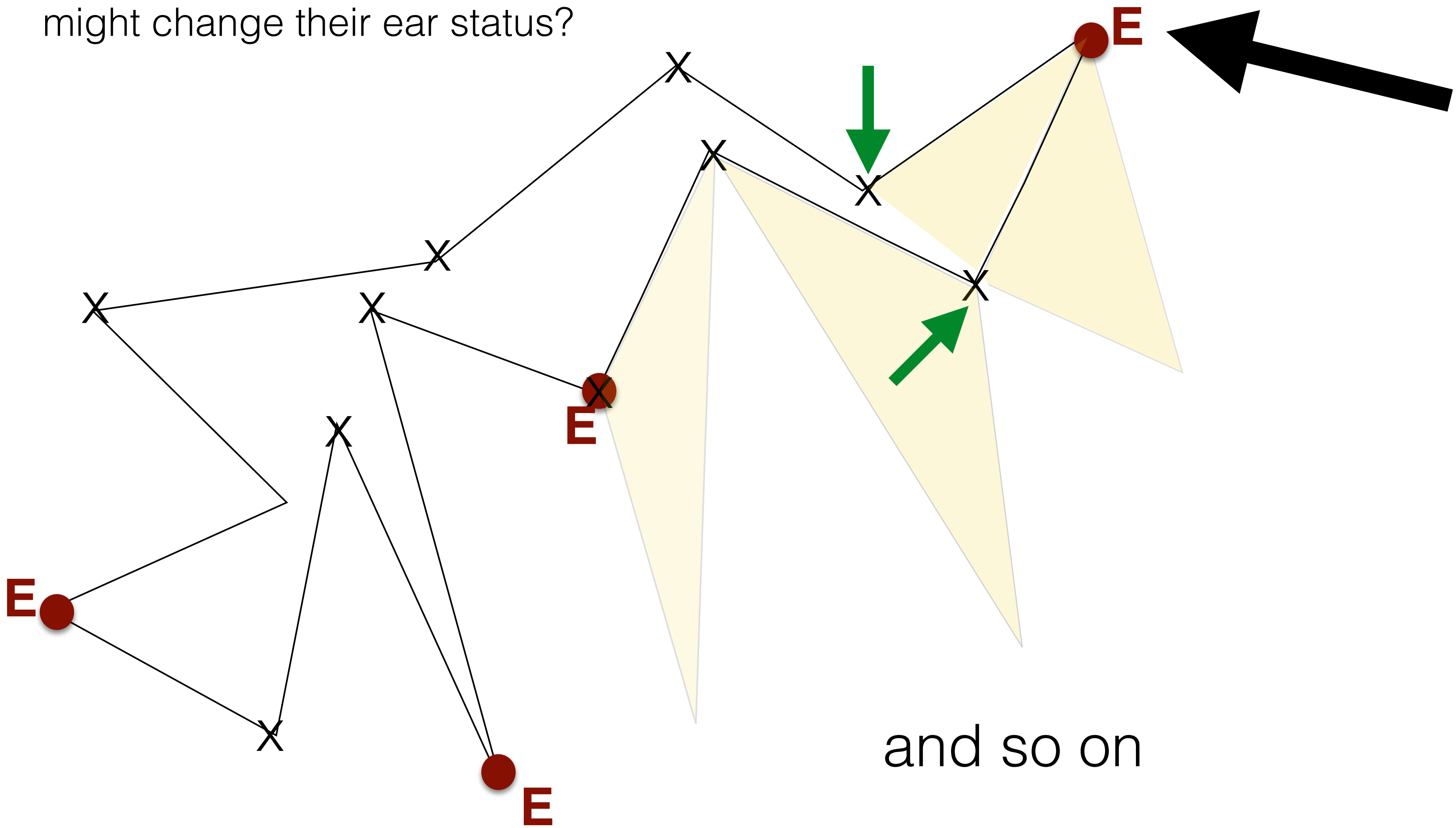
# Algorithm 4: Improved ear removal

- **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?



# 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

- 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[i] is ear if isDiagonal($p^- p^+$)

- while n>3 do:
    - i=0
    - while i < P.size():
        - if p[i] is labeled as 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[i] is ear if isDiagonal($p^- p^+$)

O(n²)

- while n>3 do:
  - i=0
  - while i < P.size():
    - if p[i] is labeled as 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
- …
- Many papers with improved bounds
- …
- 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)

practical

not practical

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