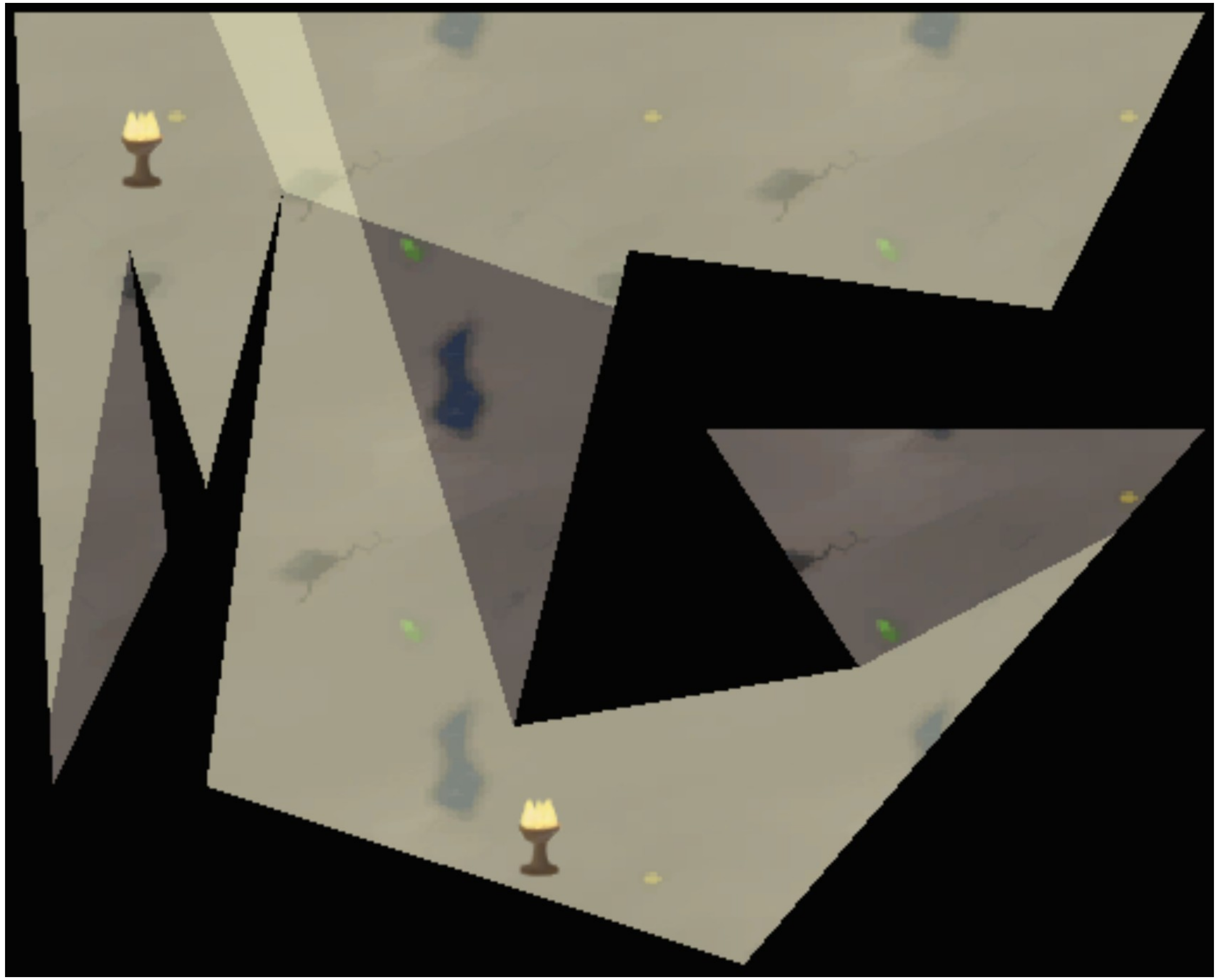
Polygon Triangulation

Geometric Algorithms

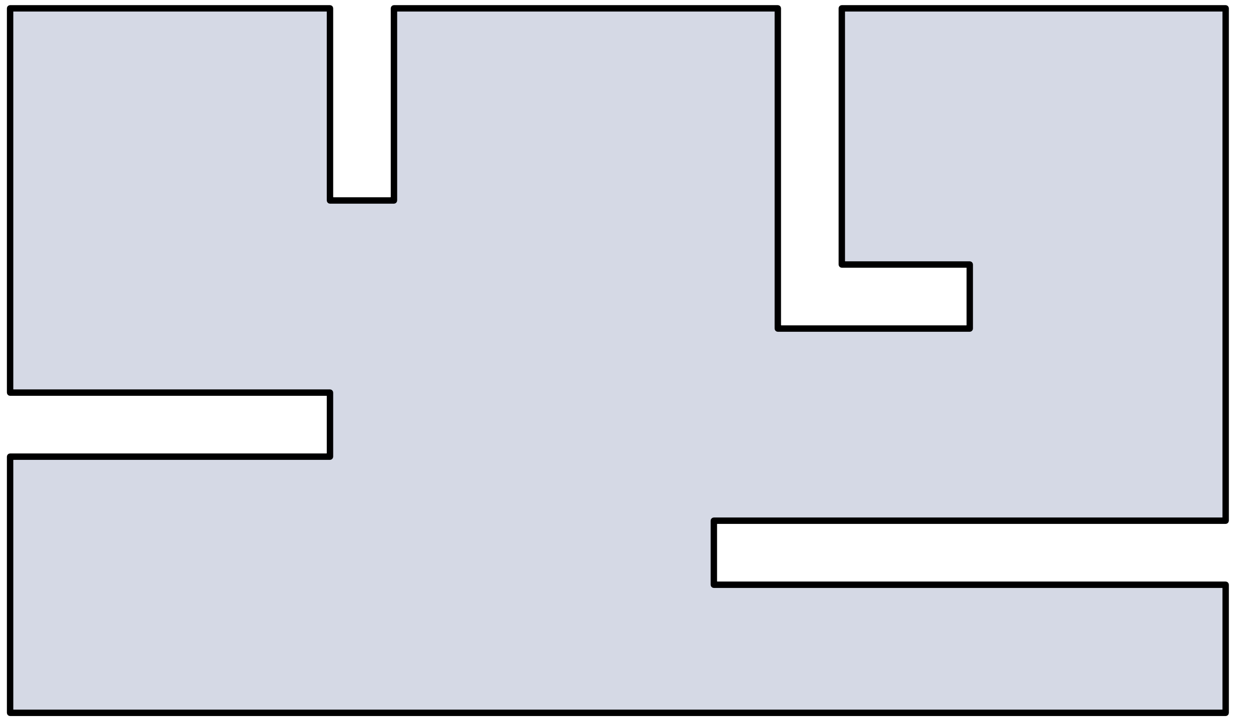
The Art Gallery Problem



https://kbuchin.github.io/ruler/art/

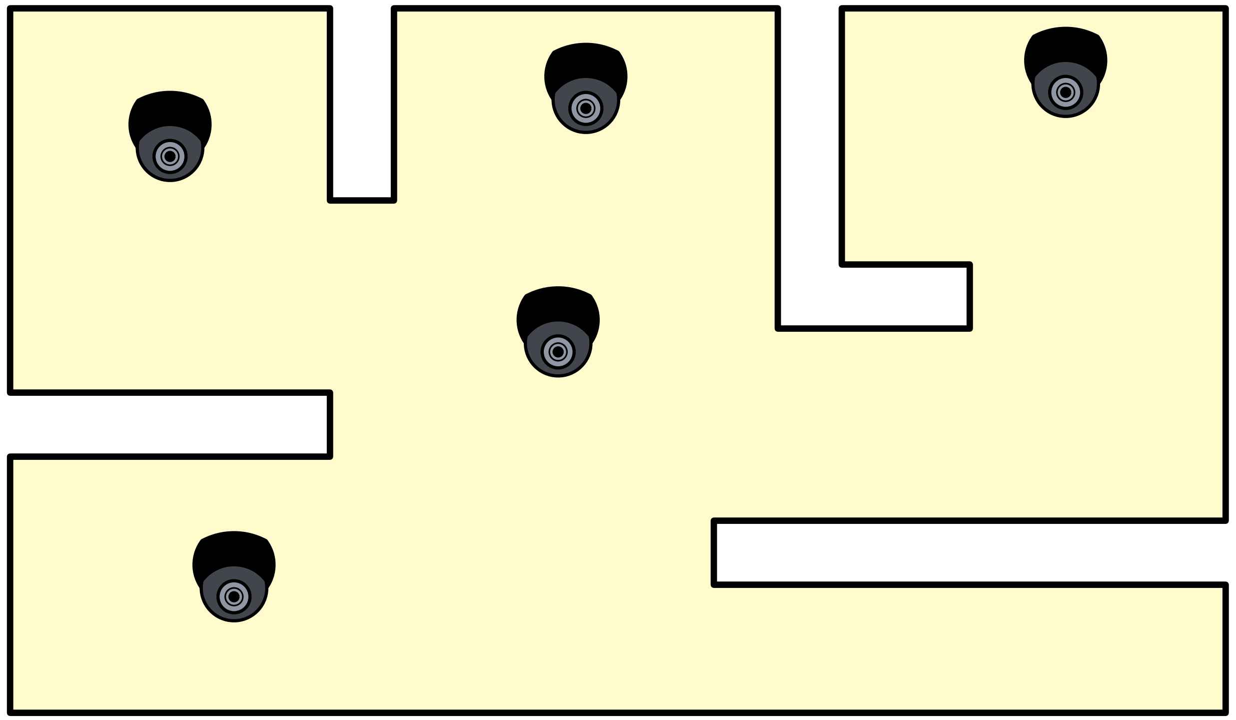
The Art Gallery Problem

Problem: Install 360°-cameras for the surveillance of an art gallery such that the whole gallery is seen.



The Art Gallery Problem

Problem: Install 360°-cameras for the surveillance of an art gallery such that the whole gallery is seen.

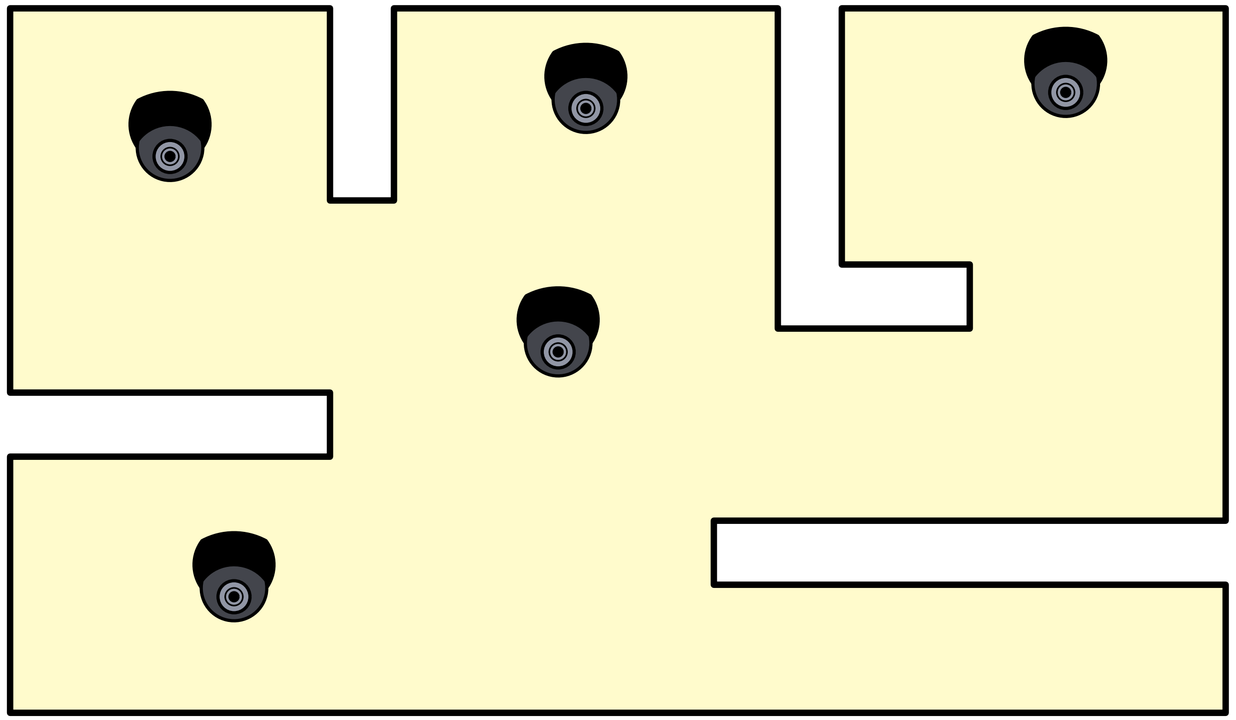


The Art Gallery Problem

Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.

Assumption:

Gallery is a simple polygon P with n vertices (no holes)



The Art Gallery Problem

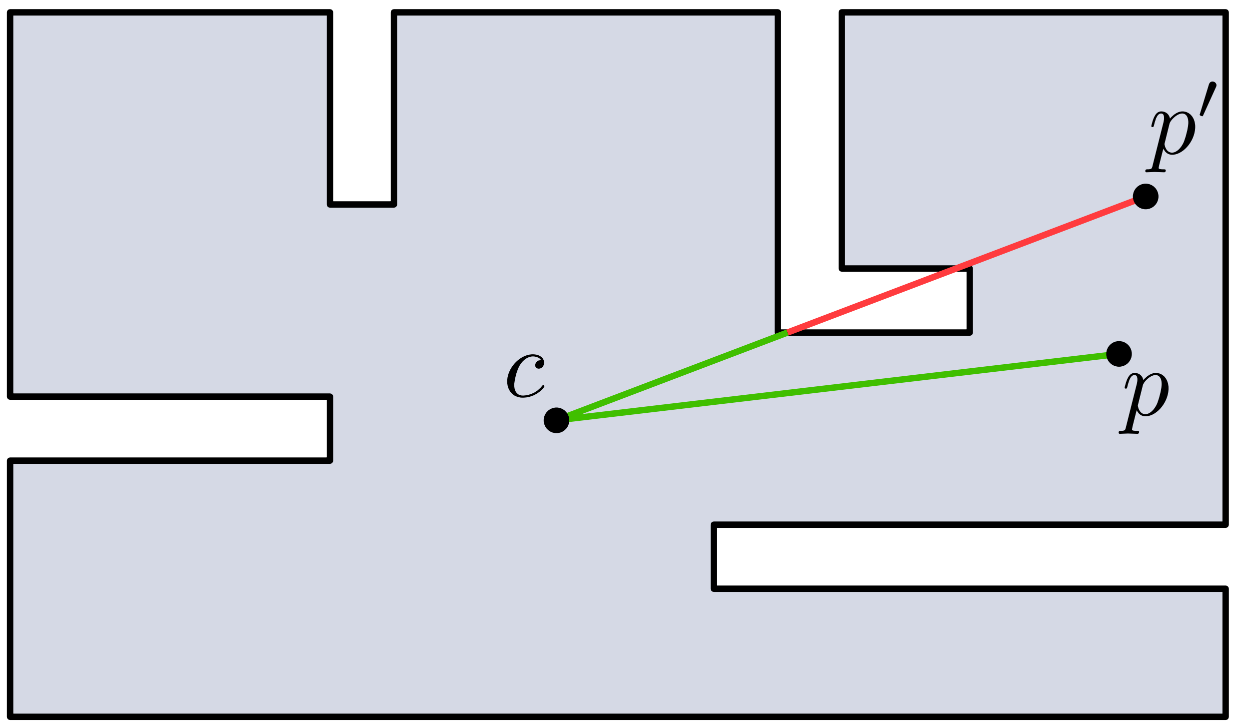
Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.

Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$



The Art Gallery Problem

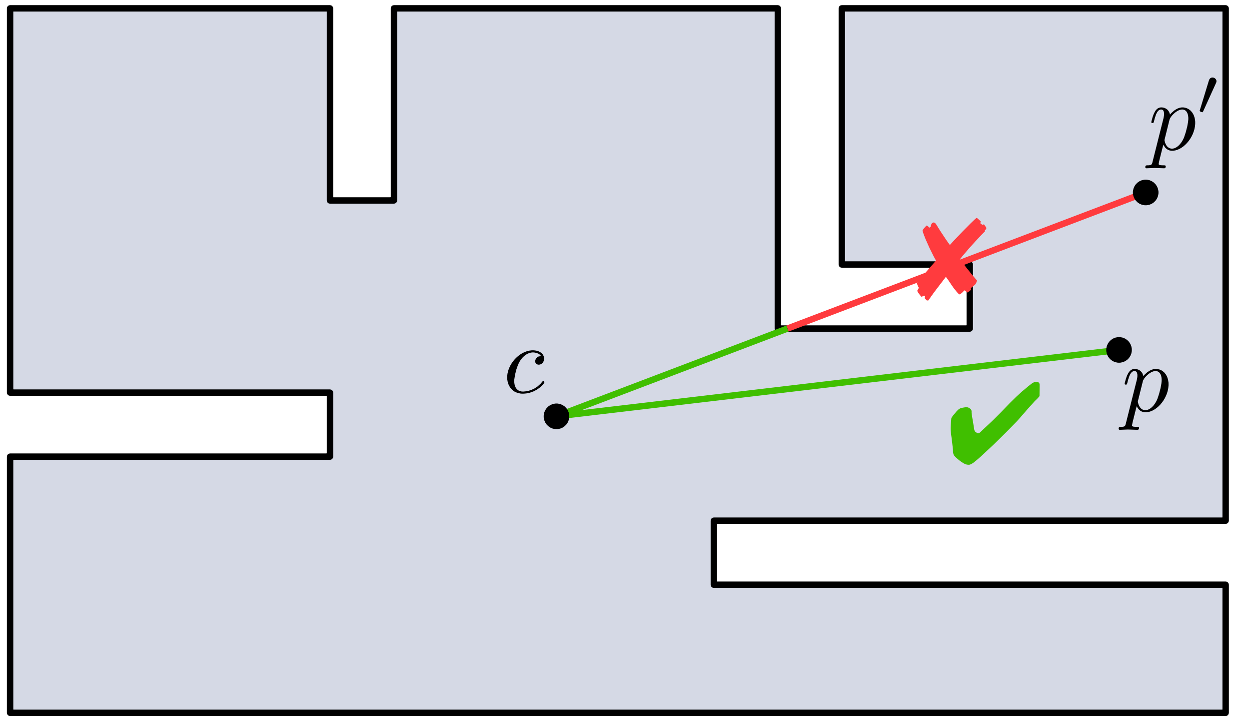
Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.

Assumption:

Gallery is a simple polygon P with n vertices (no holes)

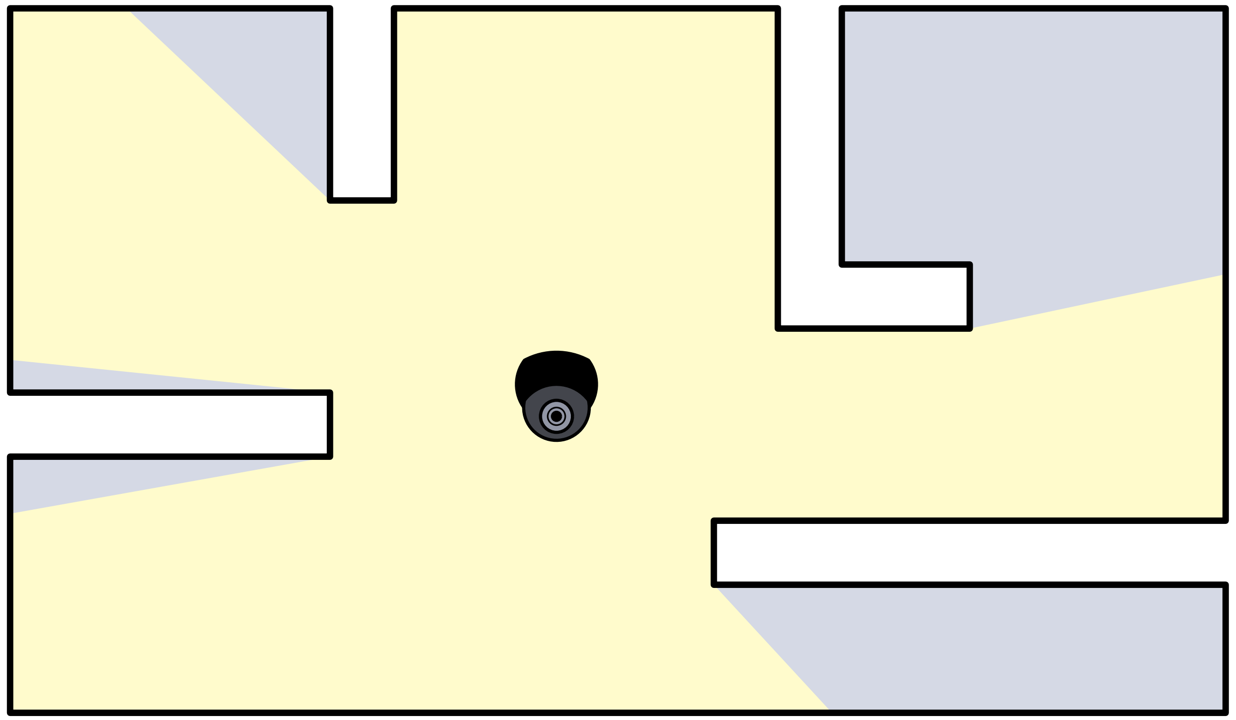
Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$



The Art Gallery Problem

Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.



Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definition:

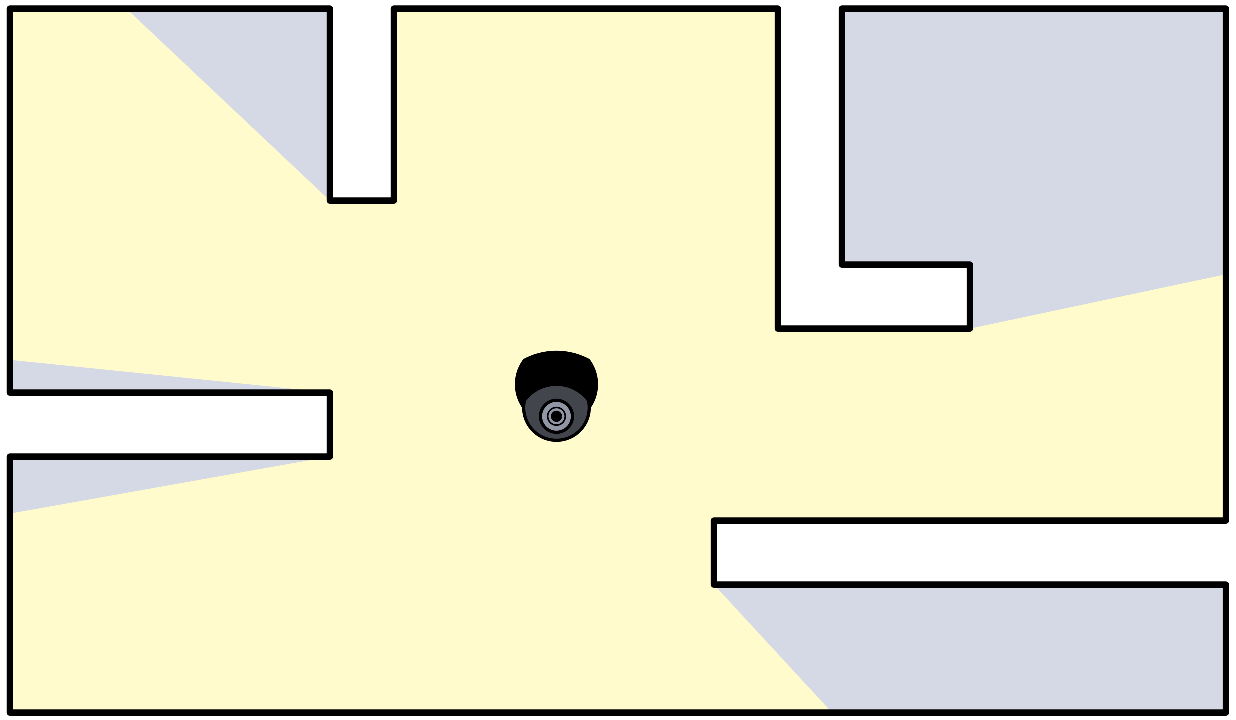
Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$

Observation:

Every camera sees a star-shaped region

The Art Gallery Problem

Problem: Install 360°-cameras for the surveillance of an art gallery such that the whole gallery is seen.



Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$

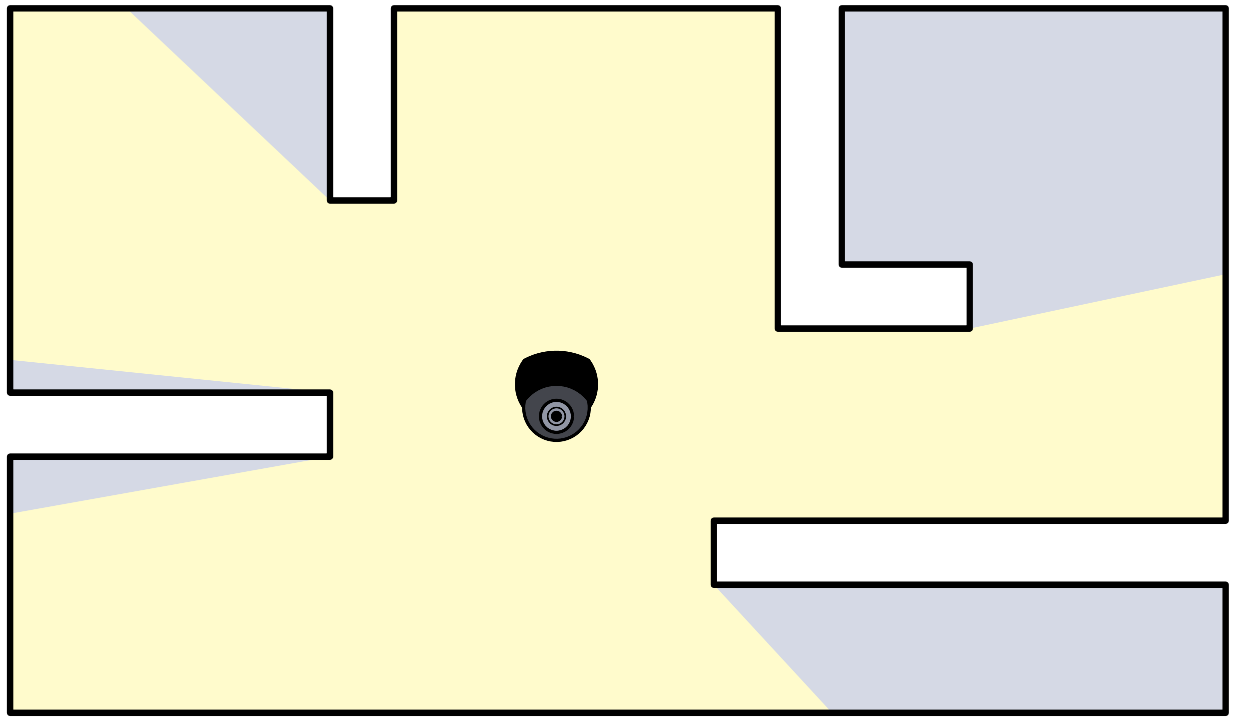
Observation:

Every camera sees a star-shaped region

Goal: Use as few cameras as possible!

The Art Gallery Problem

Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.



Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$

Observation:

Every camera sees a star-shaped region

Goal: Use as few cameras as possible!

$\Rightarrow$ Number depends on complexity n and shape of P

The Art Gallery Problem

Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.

Assumption:

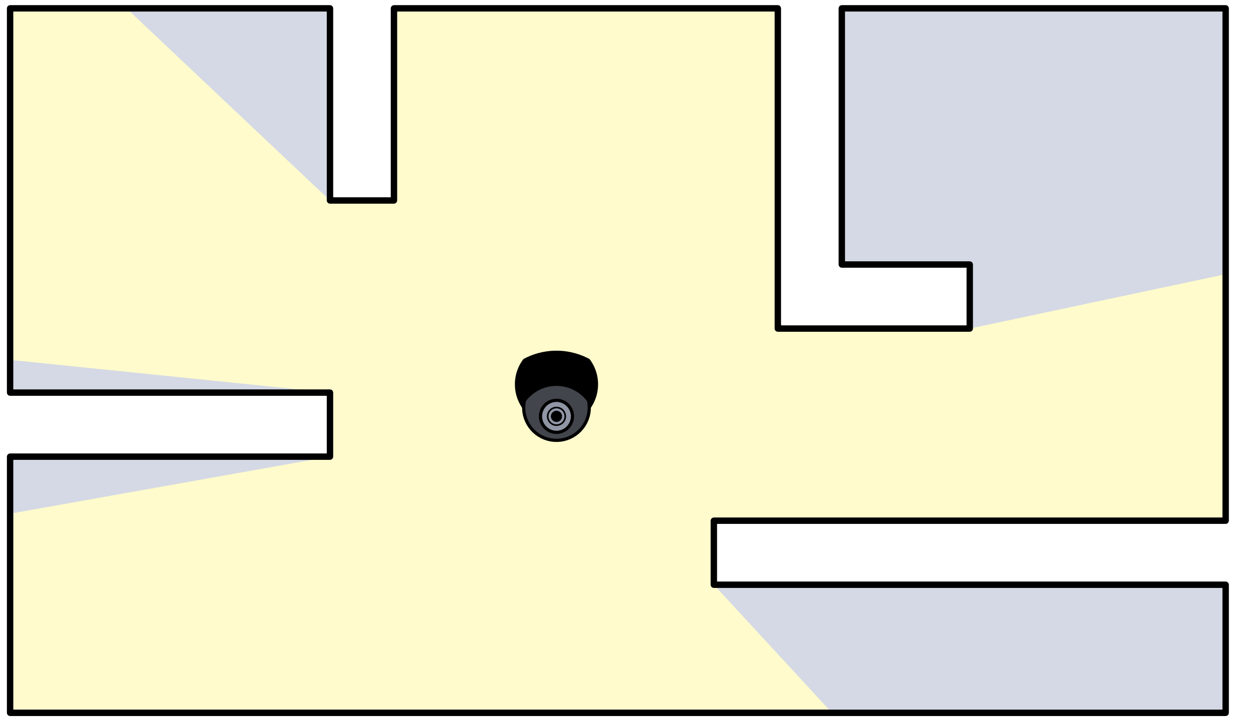
Gallery is a simple polygon P with n vertices (no holes)

Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$

Observation:

Every camera sees a star-shaped region

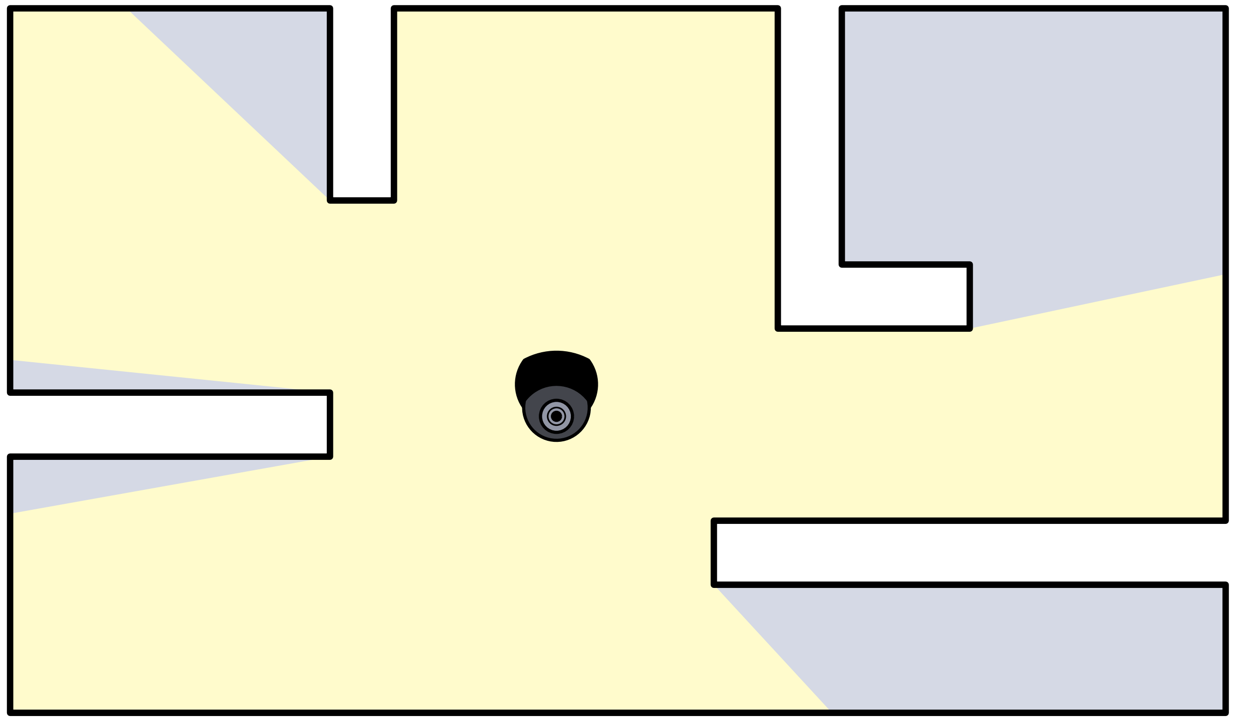


Goal: Use as few cameras as possible! NP-hard!

$\Rightarrow$ Number depends on complexity n and shape of P

The Art Gallery Problem

Problem: Install 360°-cameras for the surveillance of an art gallery such that the whole gallery is seen.



Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$

Observation:

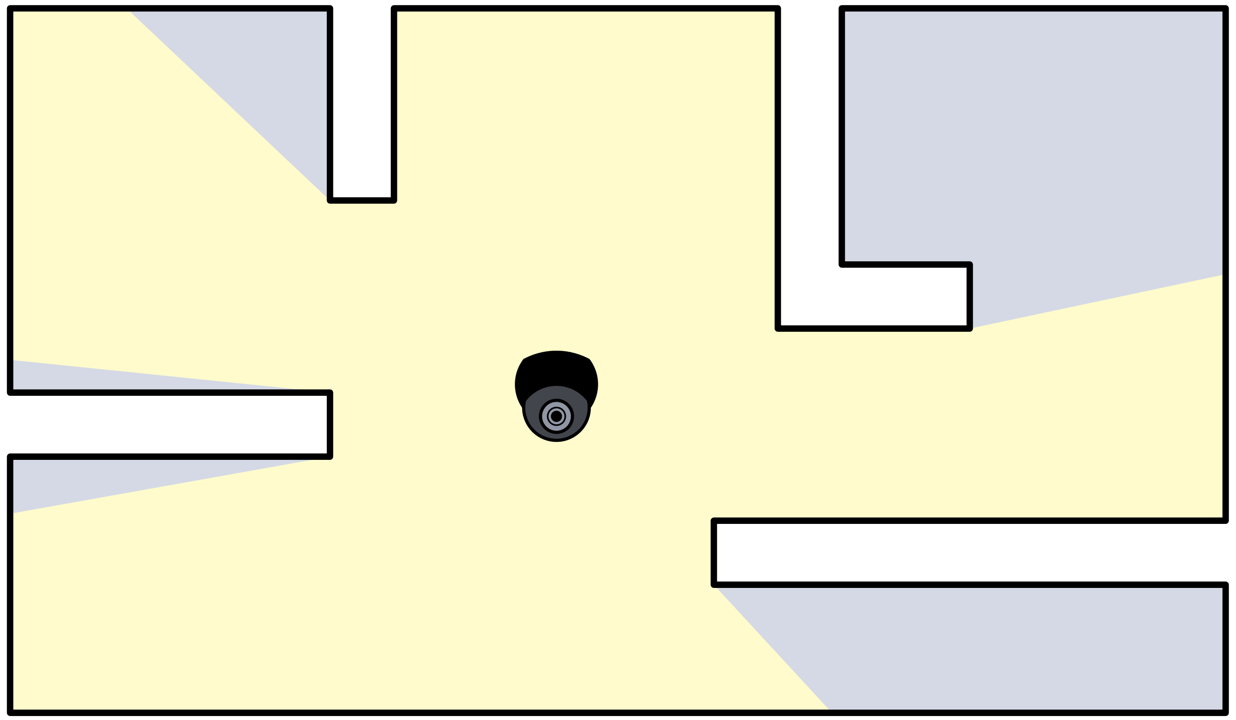
Every camera sees a star-shaped region

Goal: Use as few cameras as possible!

Simple upper and lower bounds?

The Art Gallery Problem

Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.



Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definition:

Point $p \in P$ is visible from $c \in P$ if $\overline{cp} \in P$

Observation:

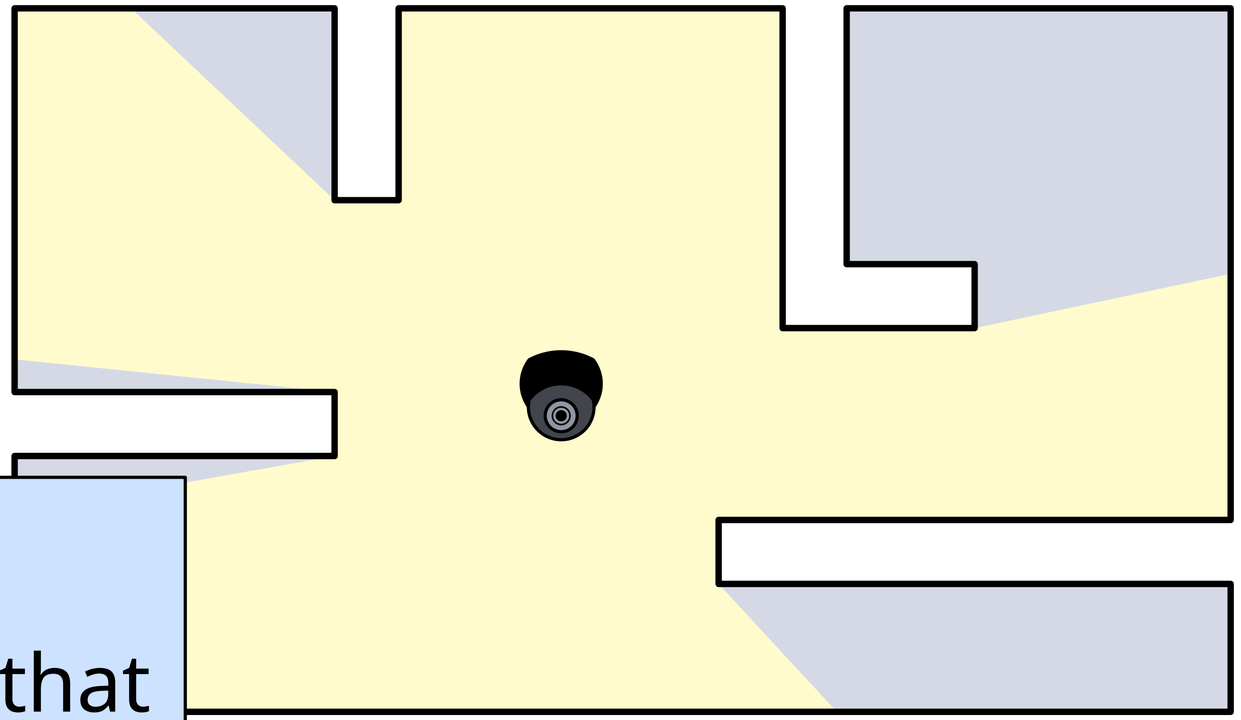
Every camera sees a star-shaped region

Goal: Use as few cameras as possible!

Simple upper and lower bounds? between 1 and n

The Art Gallery Problem

Problem: Install $360°$-cameras for the surveillance of an art gallery such that the whole gallery is seen.



Assumption:

Gallery is a simple polygon P with n vertices (no holes)

Definitio

Point p

Observati

Every c

Goal: Use

Try to find bounds.

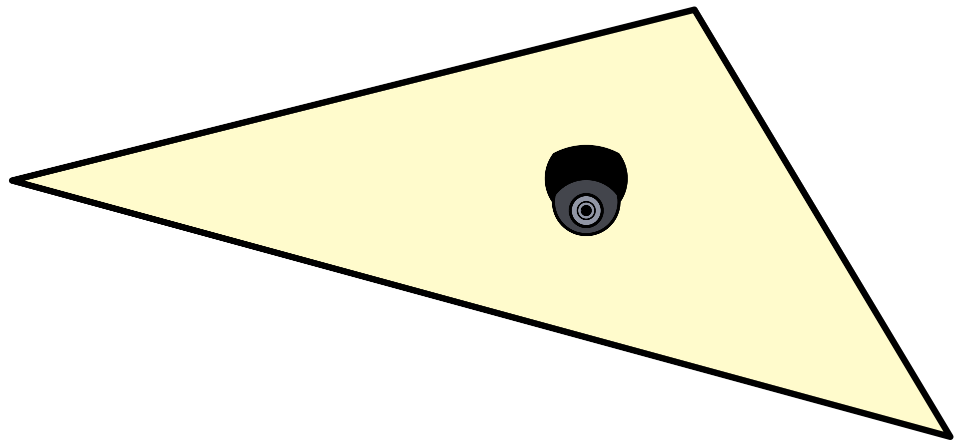
Upper bound: prove, for any polygon that that many cameras suffices.

Lower bound: construct family of polygons that needs many cameras.

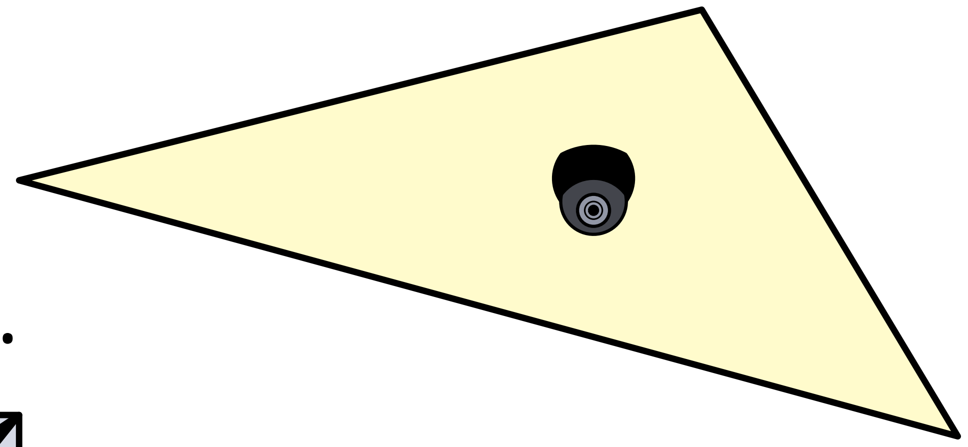
Simple upper and lower bounds? between 1 and n

Simplifying the Problem

Observation: Triangles are easy to guard.

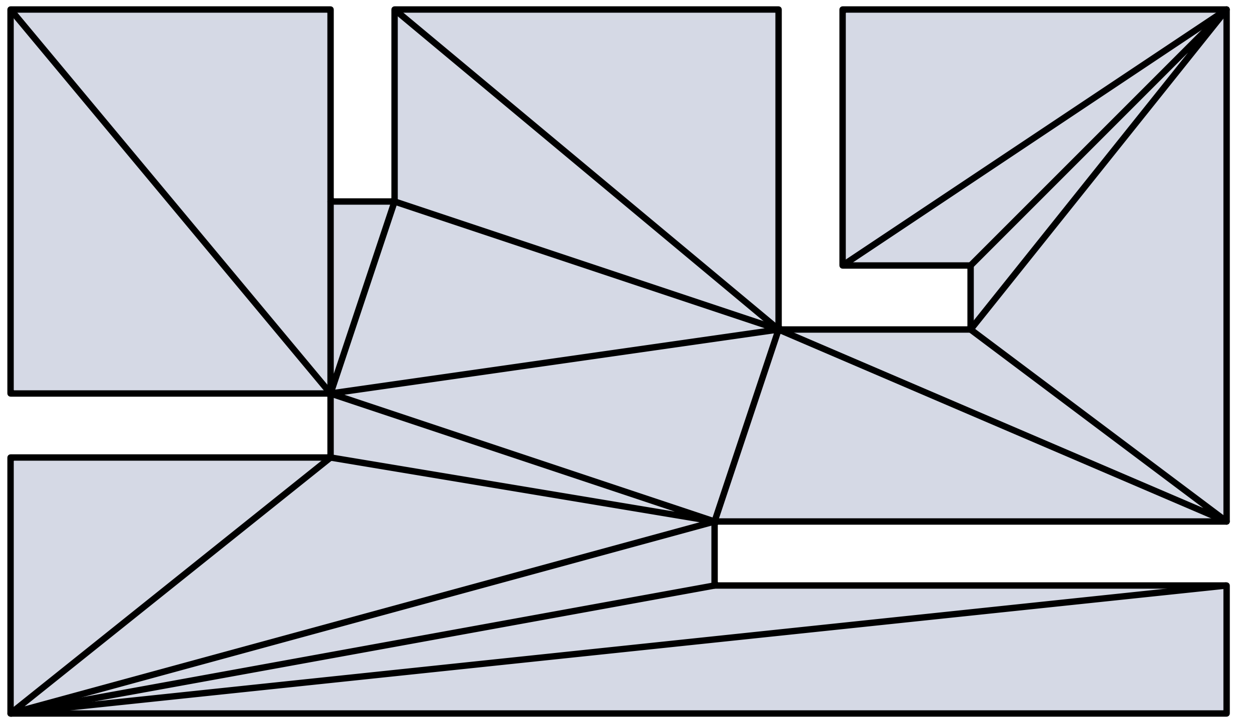


Simplifying the Problem

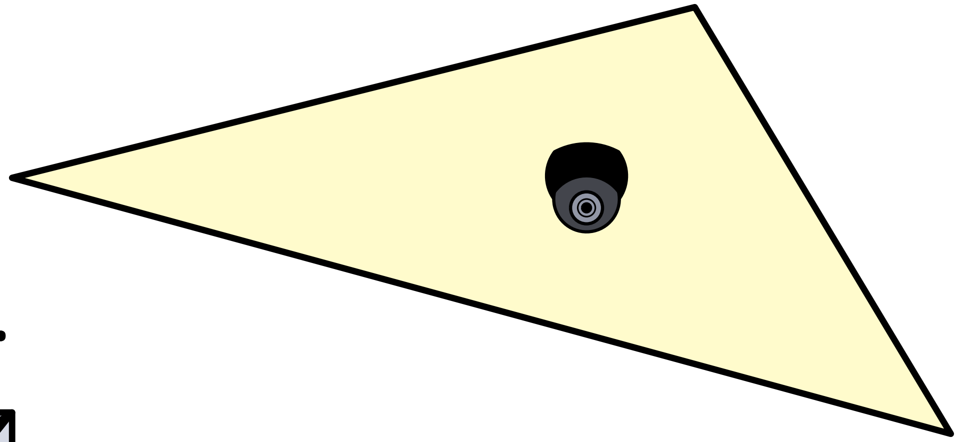


Observation: Triangles are easy to guard.

Idea: Partition P into triangles and guard every triangle.

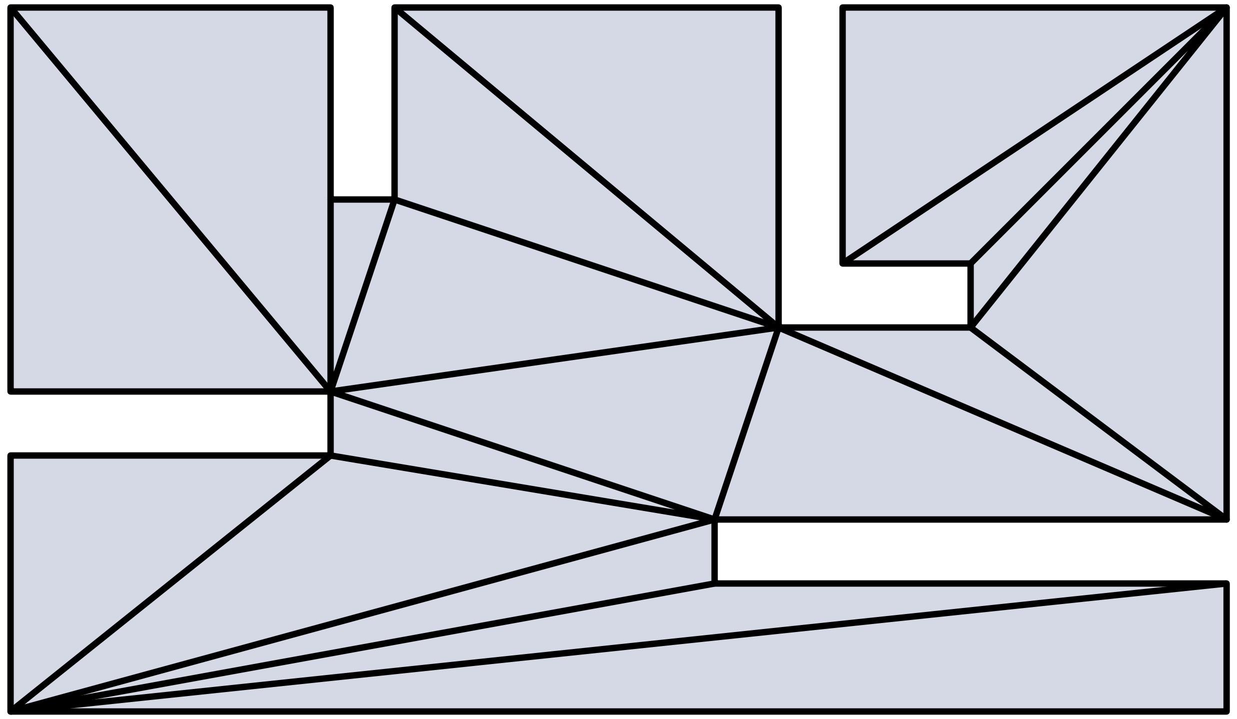


Simplifying the Problem



Observation: Triangles are easy to guard.

Idea: Partition P into triangles and guard every triangle.

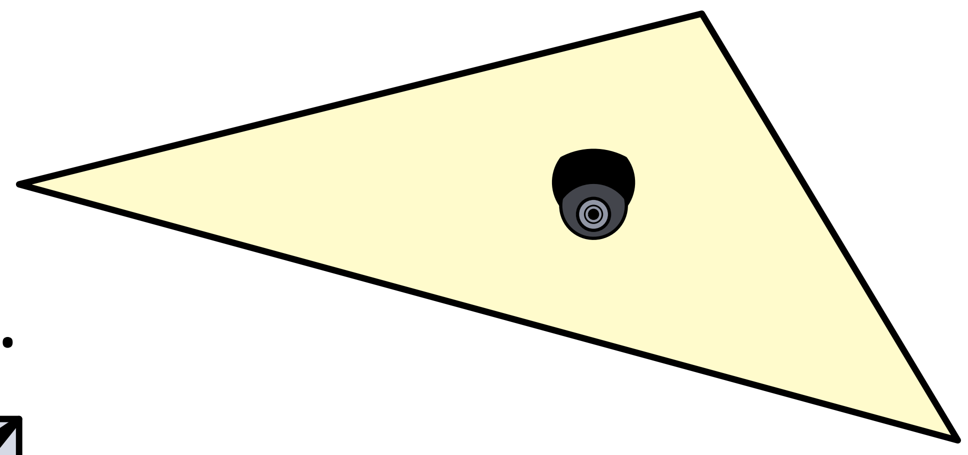


Does a triangulation always exist?

Is it unique?

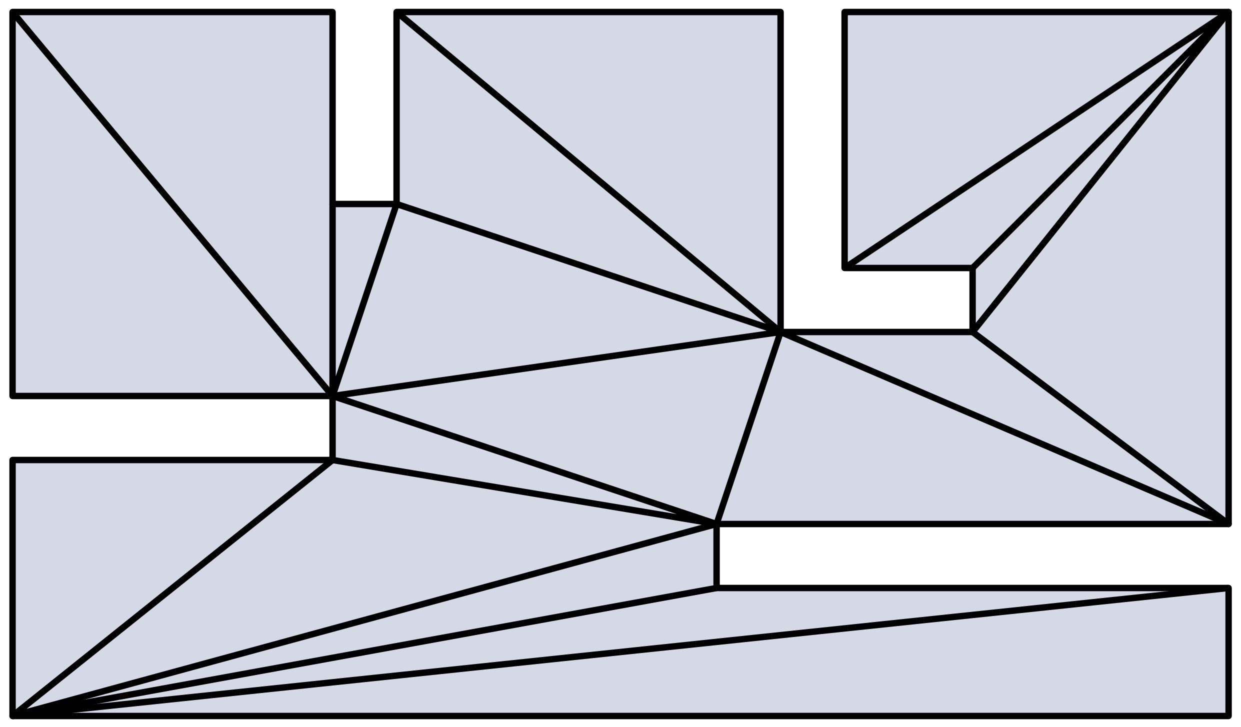
How many triangles does a triangulation have?

Simplifying the Problem



Observation: Triangles are easy to guard.

Idea: Partition P into triangles and guard every triangle.



Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n - 2$ triangles.

Existence of Triangulation

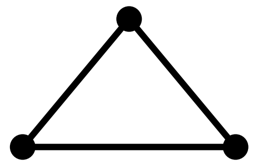
Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n

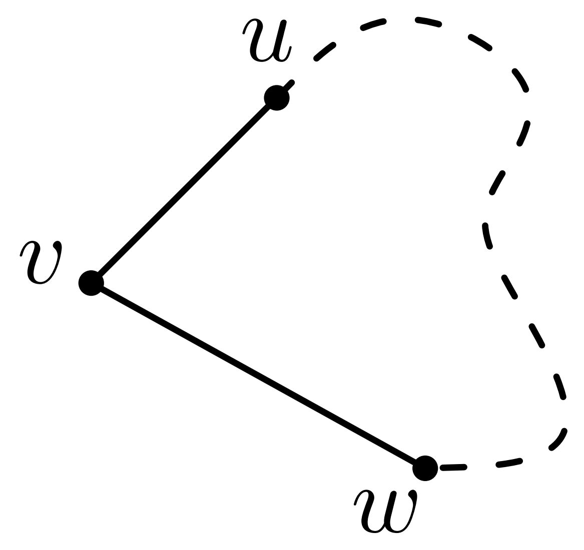
$n = 3:$  trivial

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n

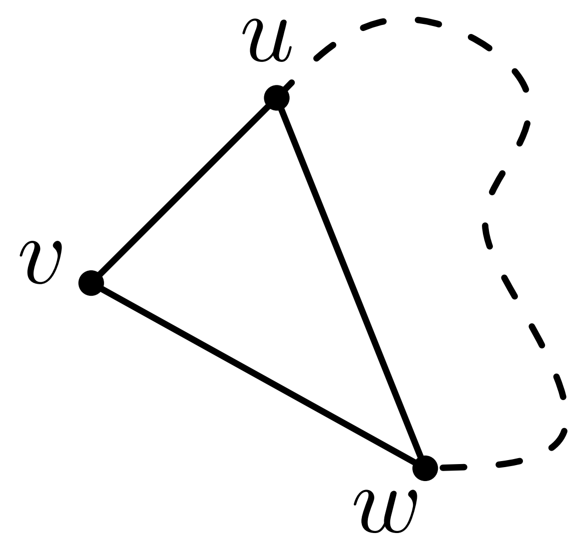
$n > 3$: Let v be the left-most vertex, and u, w its neighbors



Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n



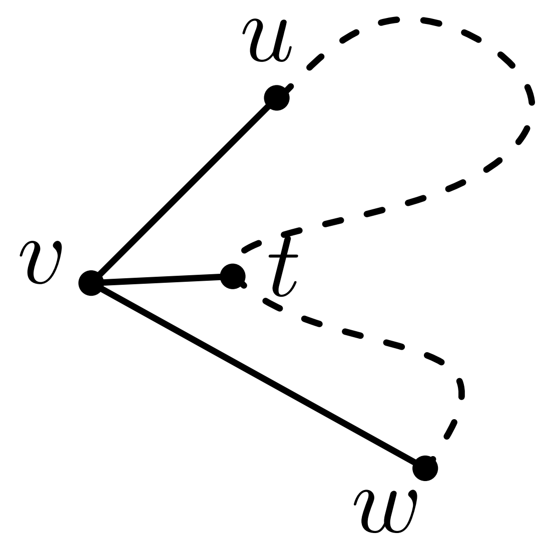
$n > 3$: Let v be the left-most vertex, and u, w its neighbors

Cases: (i) $\overline{uw}$ is a diagonal

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n



$n > 3$: Let v be the left-most vertex, and u, w its neighbors

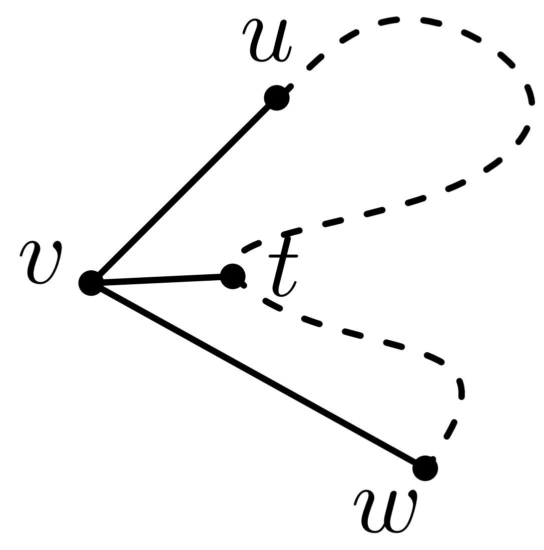
Cases: (i) $\overline{uw}$ is a diagonal

(ii) otherwise let $t \in \triangle uvw$ be furthest from $\overline{uw}$, then $\overline{vt}$ is a diagonal

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n



$n > 3$: Let v be the left-most vertex, and u, w its neighbors

Cases: (i) $\overline{uw}$ is a diagonal

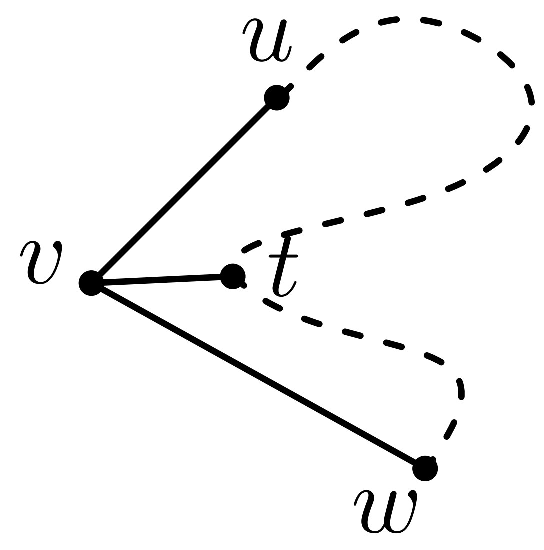
(ii) otherwise let $t \in \triangle uvw$ be furthest from $\overline{uw}$, then $\overline{vt}$ is a diagonal

In both cases: partition into polygons of size m and $n-m+2$,

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n



$n > 3$: Let v be the left-most vertex, and u, w its neighbors

Cases: (i) $\overline{uw}$ is a diagonal

(ii) otherwise let $t \in \triangle uvw$ be furthest from $\overline{uw}$, then $\overline{vt}$ is a diagonal

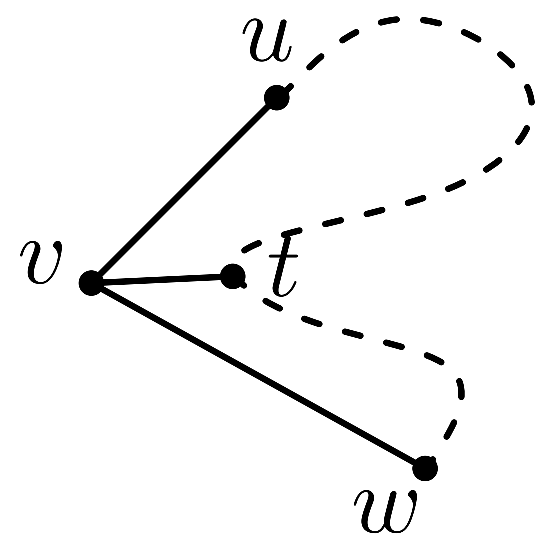
In both cases: partition into polygons of size m and $n-m+2$,

and apply induction hypothesis to get $n-2$ triangles.

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n



$n > 3$: Let v be the left-most vertex, and u, w its neighbors

Cases: (i) $\overline{uw}$ is a diagonal

(ii) otherwise let $t \in \triangle uvw$ be furthest from $\overline{uw}$, then $\overline{vt}$ is a diagonal

In both cases: partition into polygons of size m and $n-m+2$,

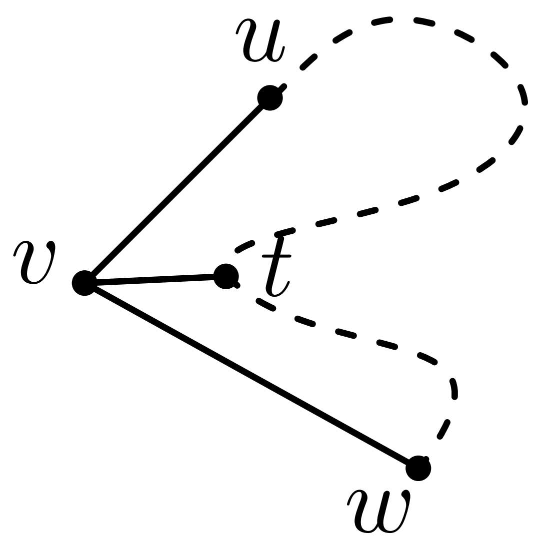
and apply induction hypothesis to get $n-2$ triangles.

Does the proof provide an algorithm?
Running time?

Existence of Triangulation

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Proof : Induction on n



$n > 3$: Let v be the left-most vertex, and u, w its neighbors

Cases: (i) $\overline{uw}$ is a diagonal

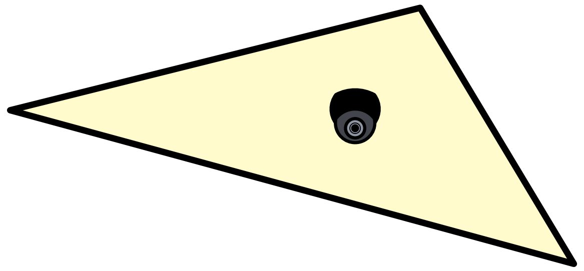
(ii) otherwise let $t \in \triangle uvw$ be furthest from $\overline{uw}$, then $\overline{vt}$ is a diagonal

In both cases: partition into polygons of size m and $n-m+2$,

and apply induction hypothesis to get $n-2$ triangles.

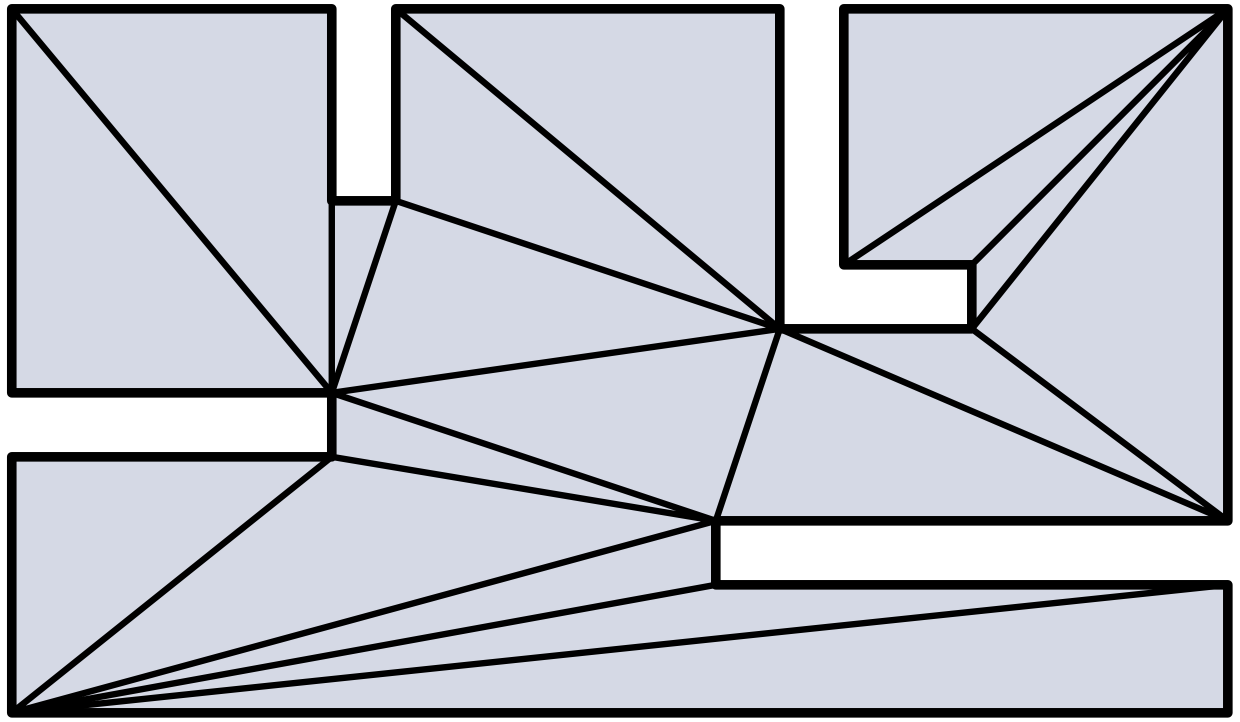
Proof results in recursive $O(n^2)$-algorithm!

Simplifying the Problem



Observation: Triangles are easy to guard.

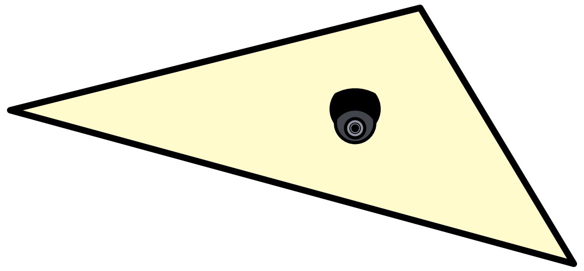
Idea: Partition P into triangles and guard every triangle.



Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n - 2$ triangles.

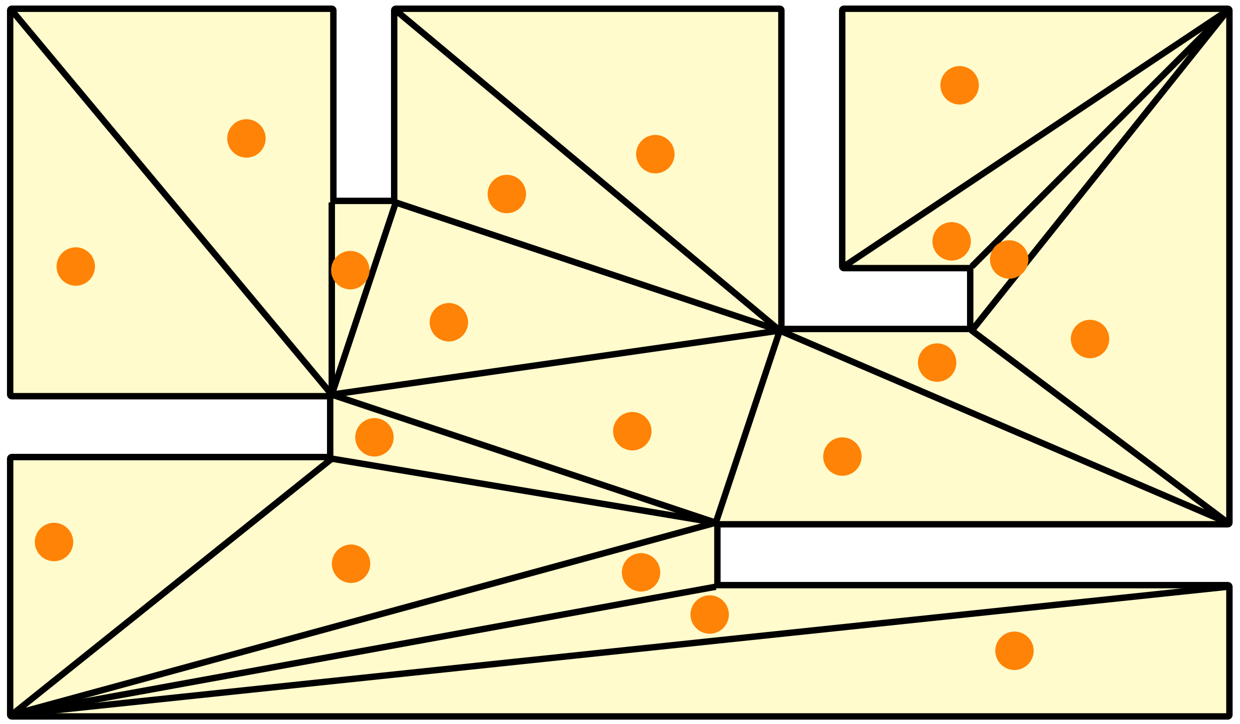
Simplifying the Problem

Observation: Triangles are easy to guard.



Idea: Partition P into triangles and guard every triangle.

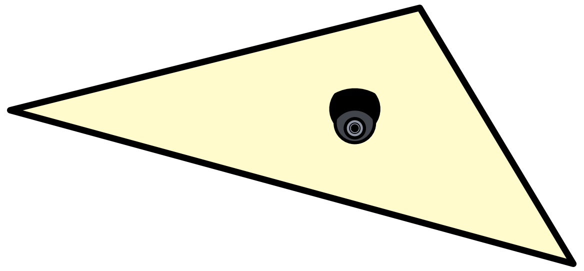
- P can be guarded with $n-2$ cameras



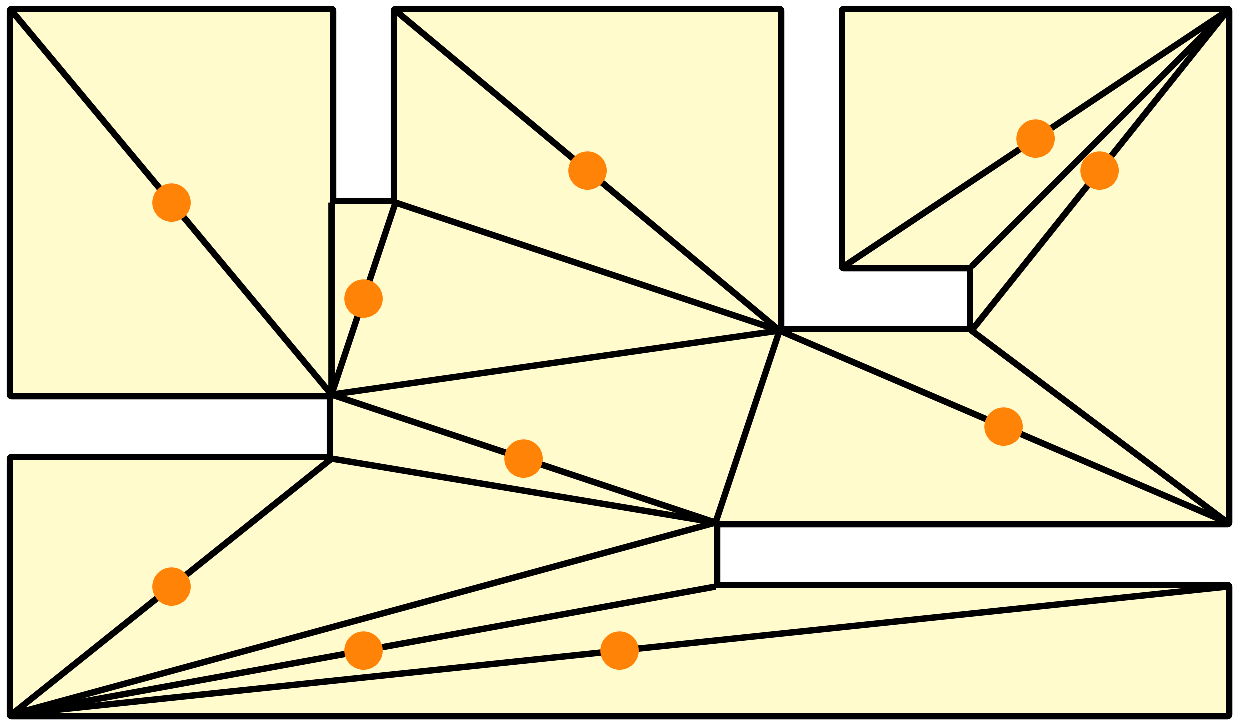
Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Simplifying the Problem

Observation: Triangles are easy to guard.



Idea: Partition P into triangles and guard every triangle.

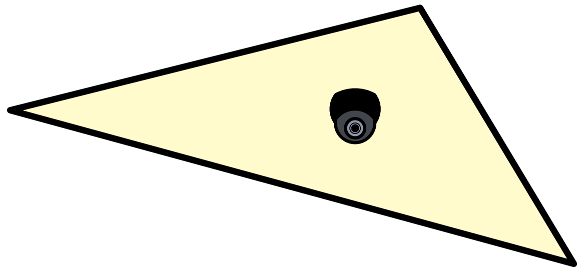


- P can be guarded with $n-2$ cameras
- P can be guarded with $\approx n/2$ cameras

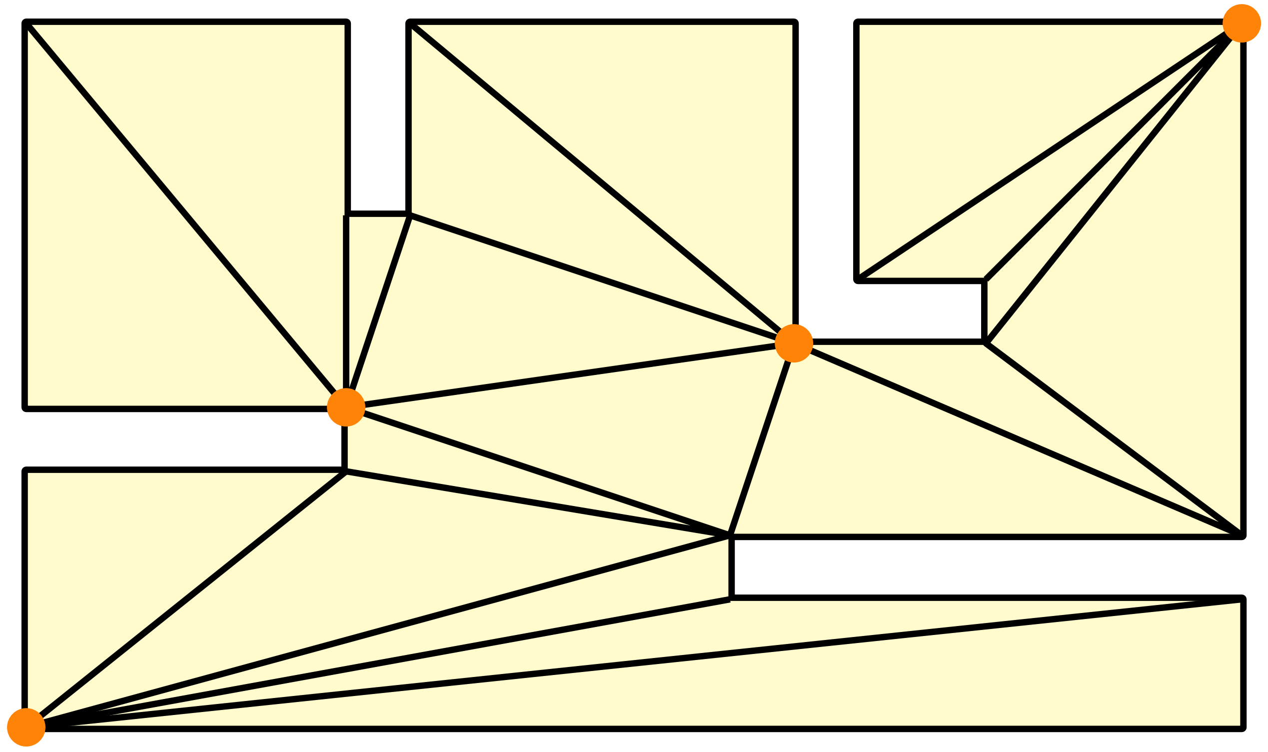
Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

Simplifying the Problem

Observation: Triangles are easy to guard.



Idea: Partition P into triangles and guard every triangle.



- P can be guarded with $n-2$ cameras
- P can be guarded with $\approx n/2$ cameras
- P can be guarded with even fewer vertex-guards (guards on vertices)

Theorem 1: Every simple polygon with n vertices has a triangulation; every such triangulation consists of $n-2$ triangles.

The Art Gallery Theorem [Chvátal ’75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

The Art Gallery Theorem [Chvátal ’75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Proof:

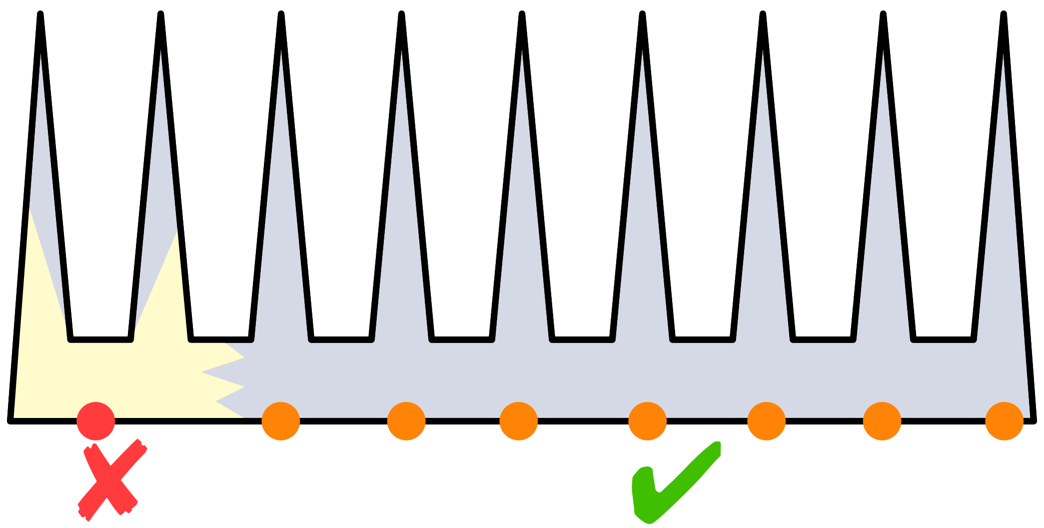
- For arbitrary large n find a simple polygon which needs $\approx n/3$ cameras

The Art Gallery Theorem [Chvátal ’75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Proof:

- For arbitrary large n find a simple polygon which needs $\approx n/3$ cameras

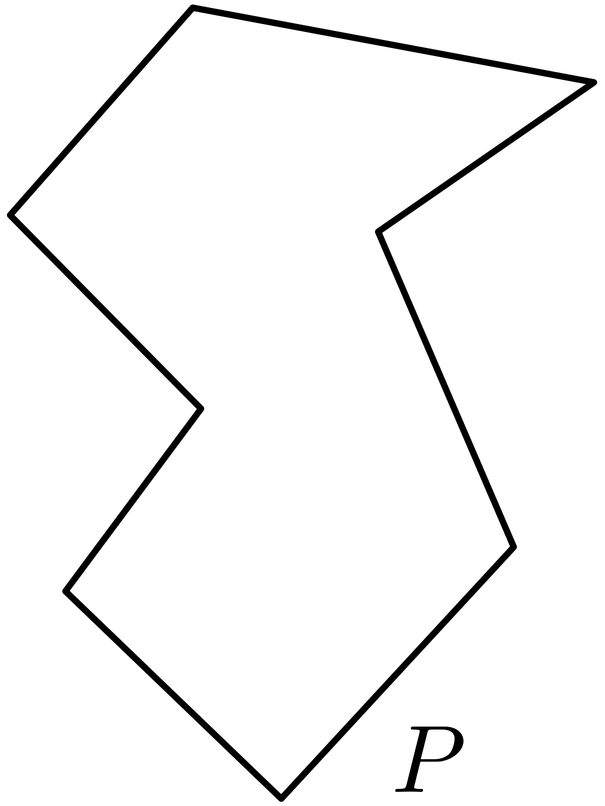


The Art Gallery Theorem [Chvátal ’75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Proof:

- P can be triangulated

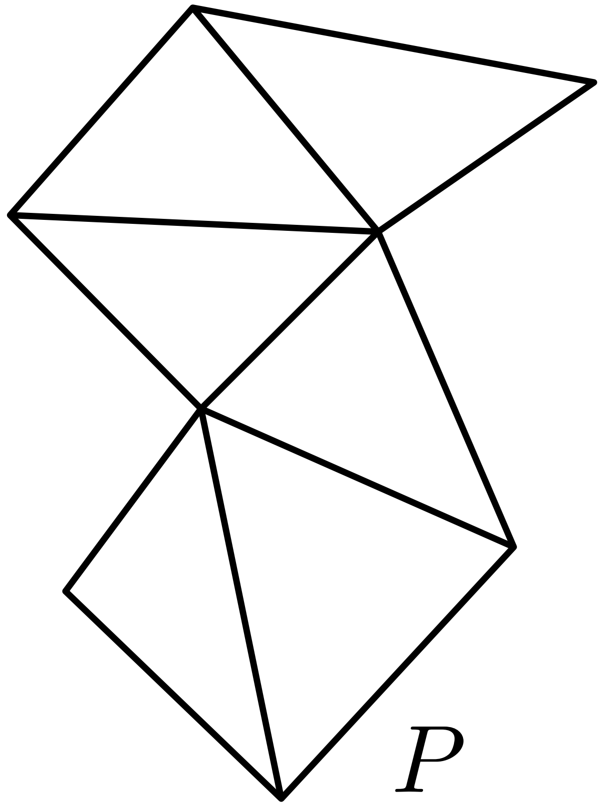


The Art Gallery Theorem [Chvátal '75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Proof:

- P can be triangulated

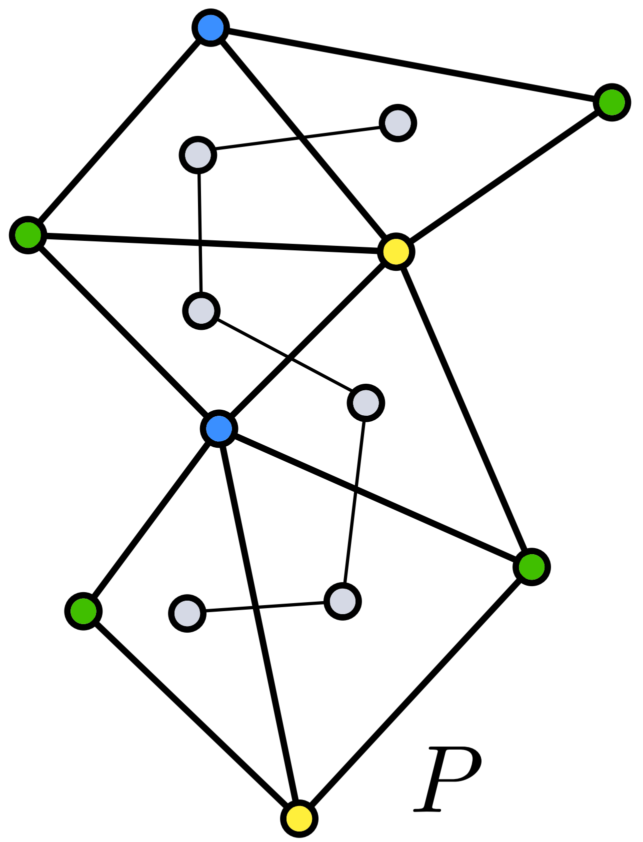


The Art Gallery Theorem [Chvátal '75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Proof:

- P can be triangulated
- Triangulation can be 3-colored (induction or consider dual graph)

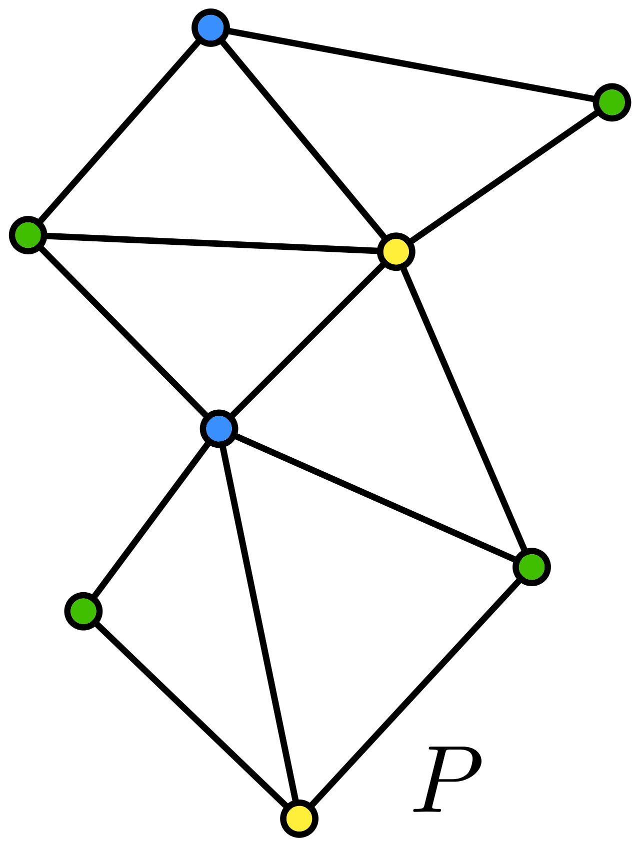


The Art Gallery Theorem [Chvátal '75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Proof:

- P can be triangulated
- Triangulation can be 3-colored (induction or consider dual graph)
- Smallest color class has $\lfloor \frac{n}{3} \rfloor$ vertices (pigeon-hole principle)

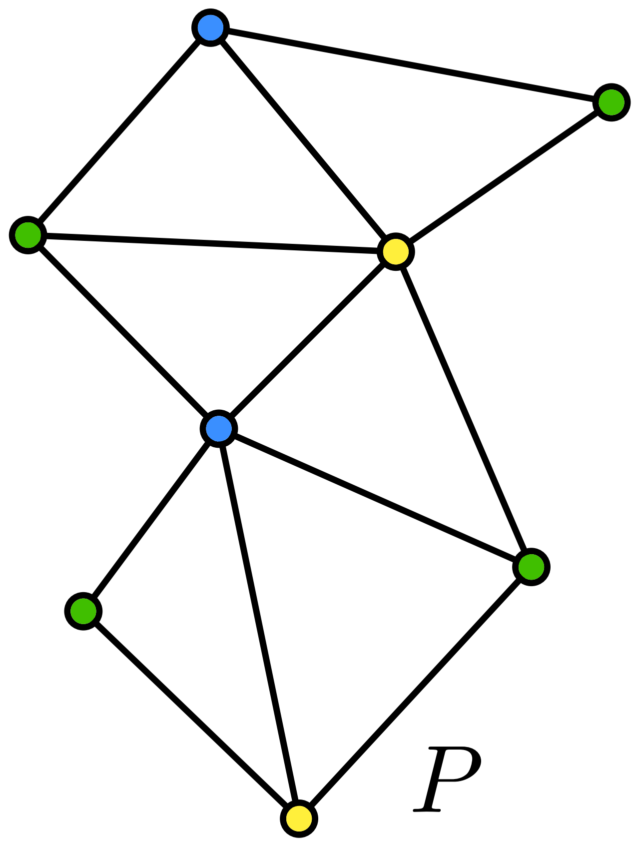


The Art Gallery Theorem [Chvátal ’75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Algorithm:

- compute triangulation
- compute dual graph
- color triangulation
- select smallest color class

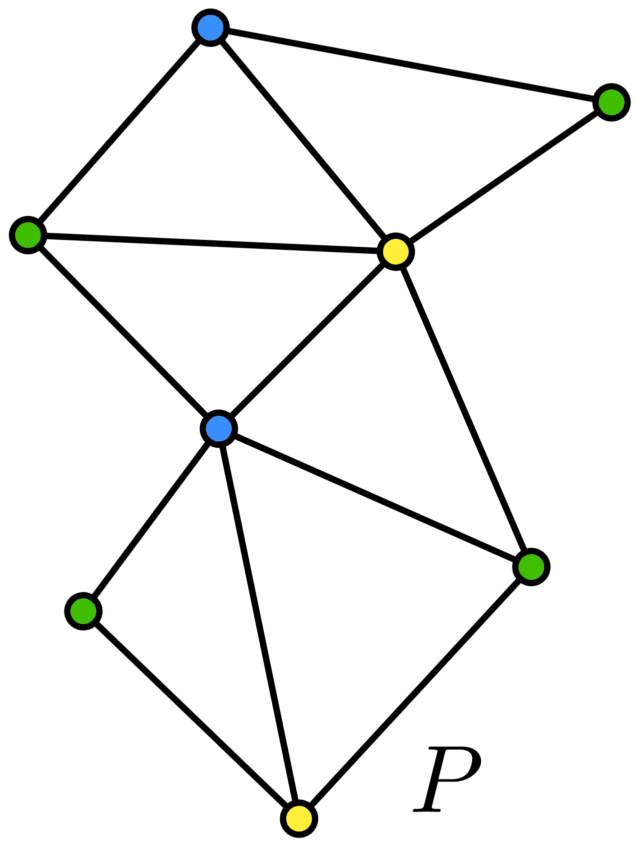


The Art Gallery Theorem [Chvátal '75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Algorithm:

- compute triangulation
- compute dual graph
- color triangulation
- select smallest color class



Running time?

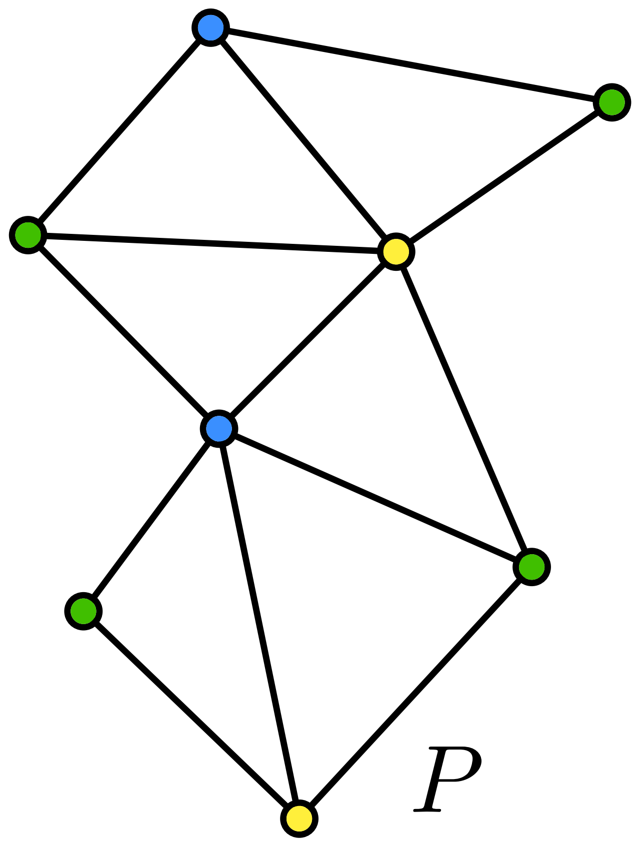
The Art Gallery Theorem [Chvátal '75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Algorithm:

- compute triangulation $O(n^2)$
- compute dual graph $O(n)$
- color triangulation
- select smallest color class

Running time? $O(n^2)$



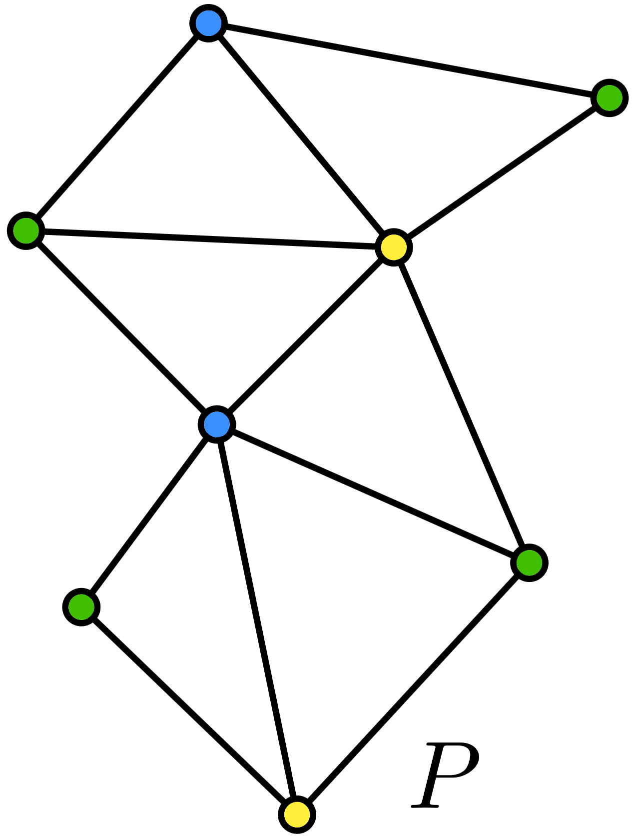
The Art Gallery Theorem [Chvátal ’75]

Theorem 2: $\lfloor n/3 \rfloor$ guards are sometimes necessary and always sufficient to guard a simple polygon with n vertices.

Algorithm:

- compute triangulation $O(n^2)$
- compute dual graph $O(n)$
- color triangulation
- select smallest color class

Now: faster triangulation algorithm



Triangulation: Overview

Idea: Partition into simpler parts and triangulate those.

Triangulation: Overview

Idea: Partition into simpler parts and triangulate those.

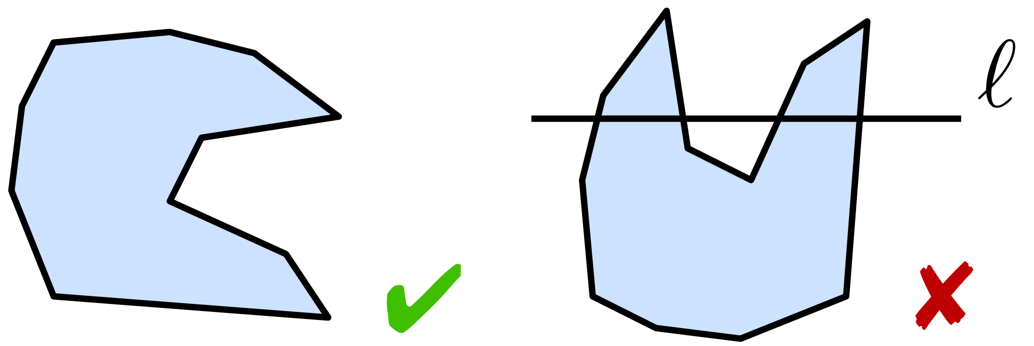
Which polygons are easy to triangulate?

Triangulation: Overview

2-step procedure:

- step 1: partition P into y-monotone subpolygons

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.



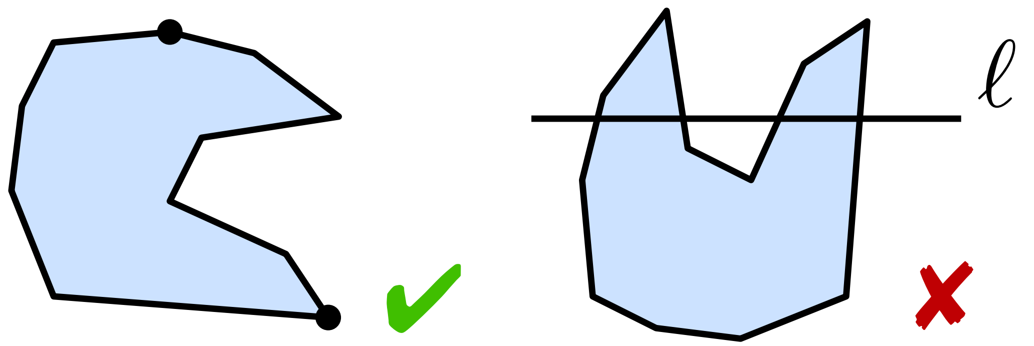
Triangulation: Overview

2-step procedure:

- step 1: partition P into y-monotone subpolygons

boundary from top to bottom never goes up

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.

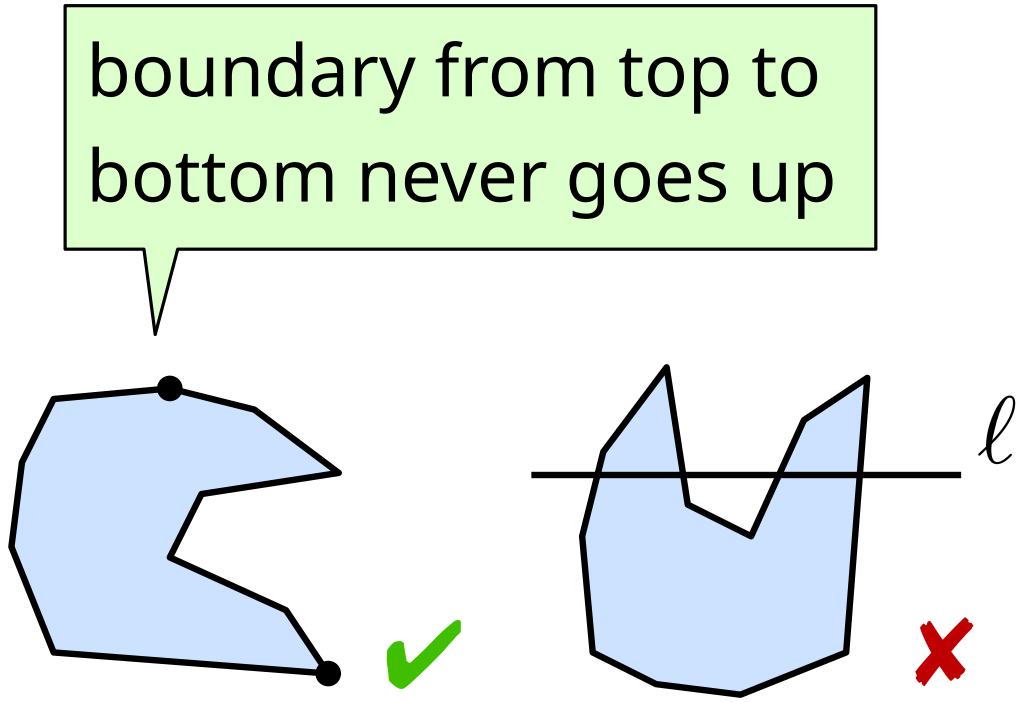


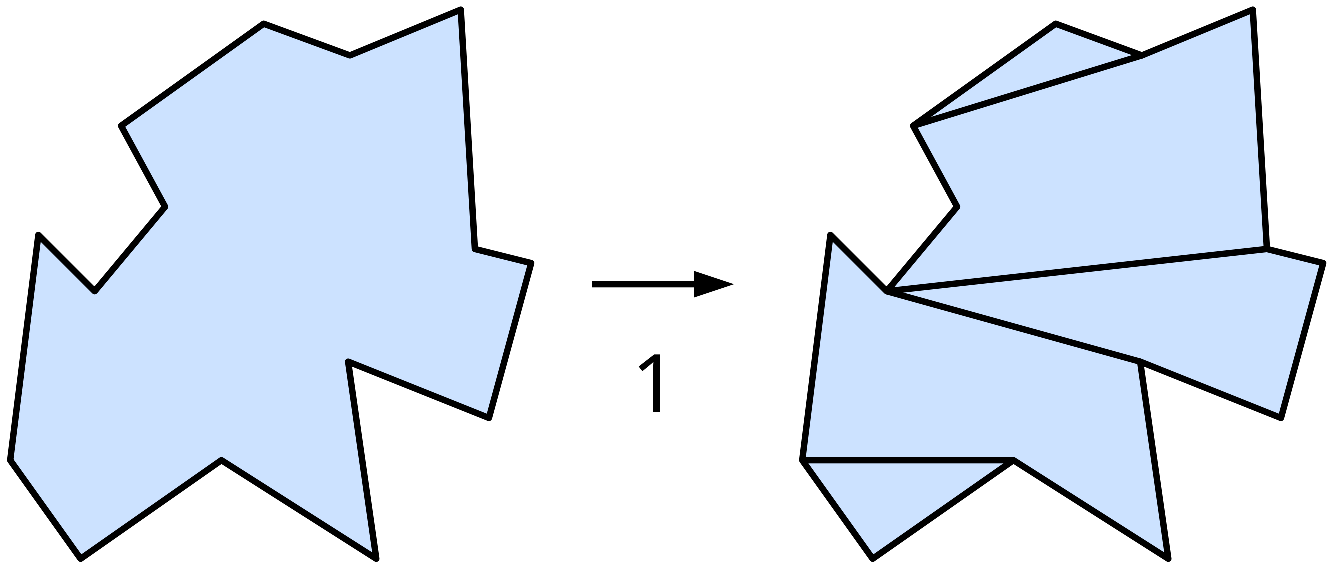
Triangulation: Overview

2-step procedure:

- step 1: partition P into y-monotone subpolygons

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.



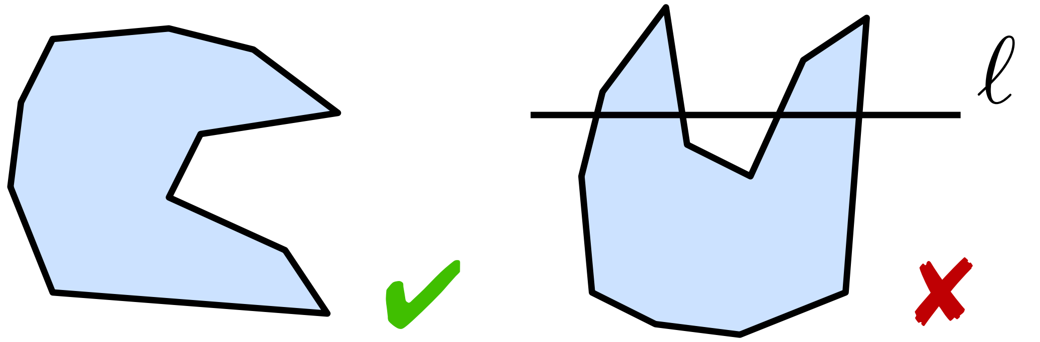


Triangulation: Overview

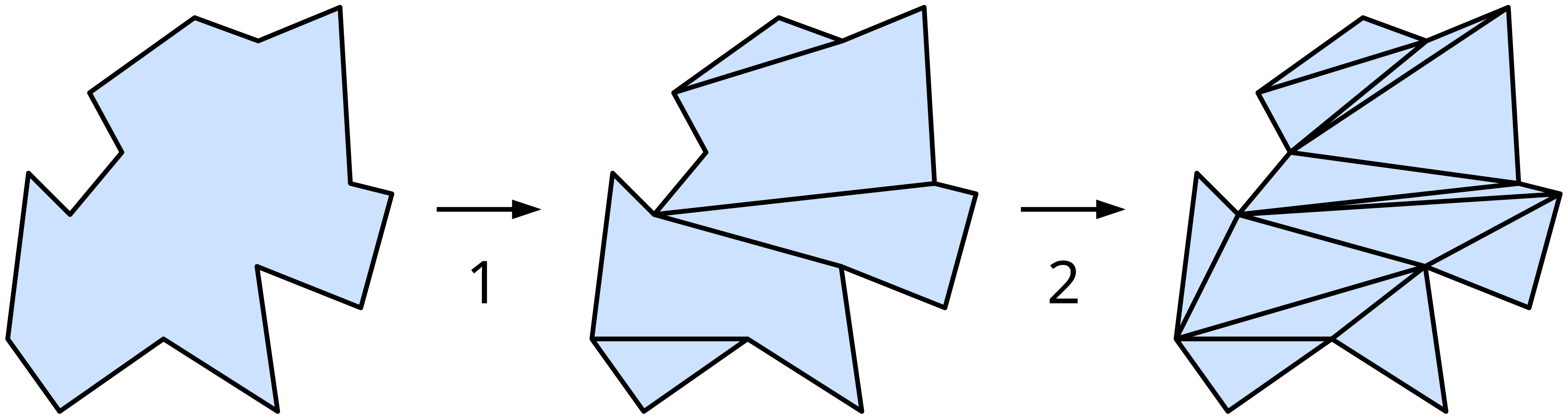
2-step procedure:

- step 1: partition P into y-monotone subpolygons

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.

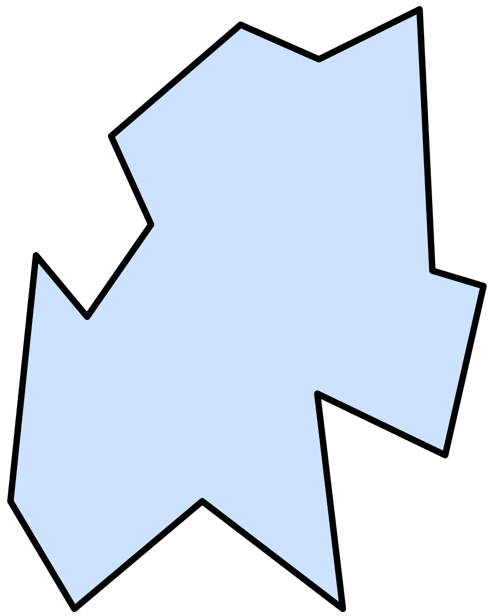


- step 2: triangulate y-monotone subpolyons



Partition into y-monotone Pieces

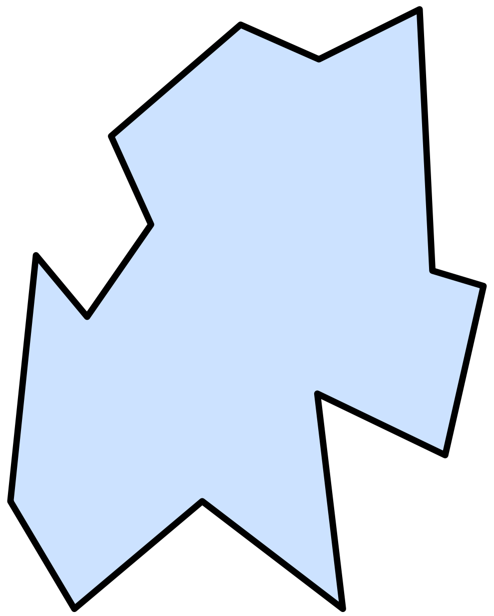
Idea: Distinguish 5 types of vertices



Partition into y-monotone Pieces

Idea: Distinguish 5 types of vertices

– turn vertices:

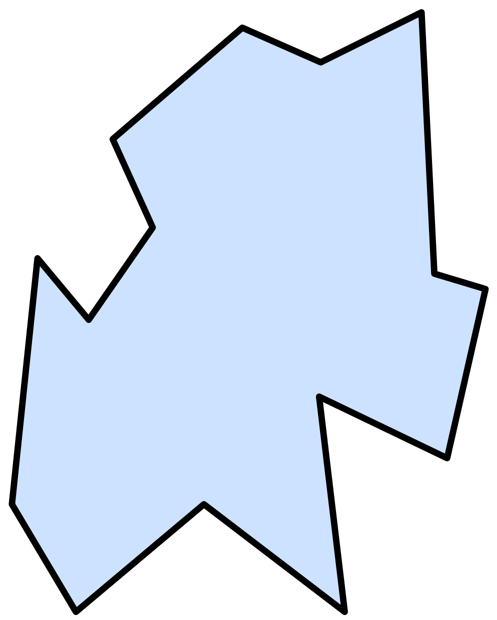


– regular vertex

Partition into y-monotone Pieces

Idea: Distinguish 5 types of vertices

– turn vertices: vertical direction switches

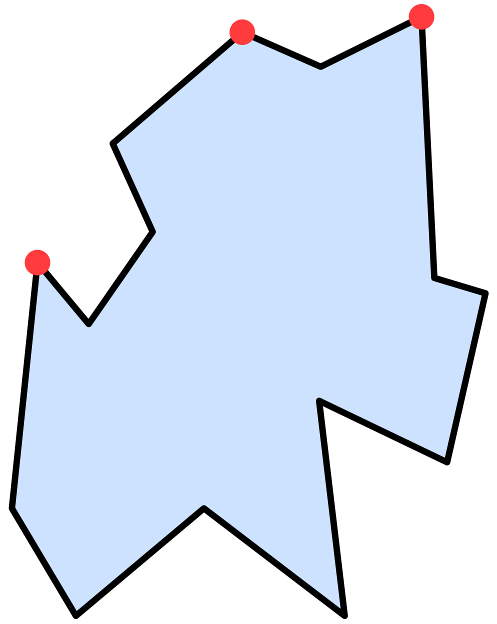


– regular vertex

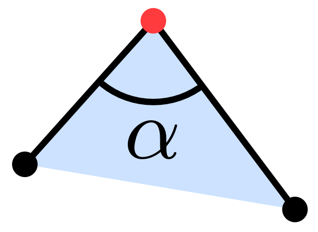
Partition into y-monotone Pieces

Idea: Distinguish 5 types of vertices

– turn vertices: vertical direction switches



- start vertex



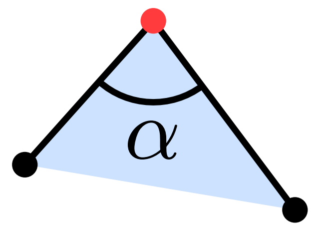
if $\alpha < 180°$

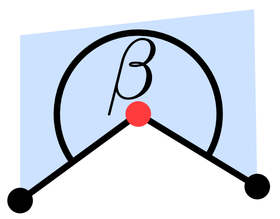
– regular vertex

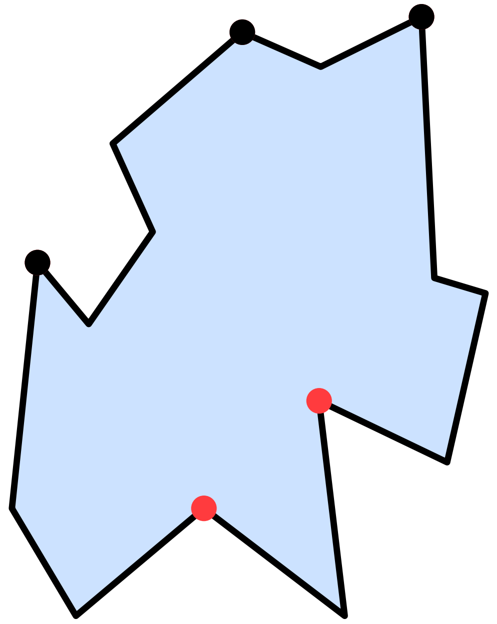
Partition into y-monotone Pieces

Idea: Distinguish 5 types of vertices

– turn vertices: vertical direction switches

- start vertex  if $\alpha < 180°$

- split vertex  if $\beta > 180°$

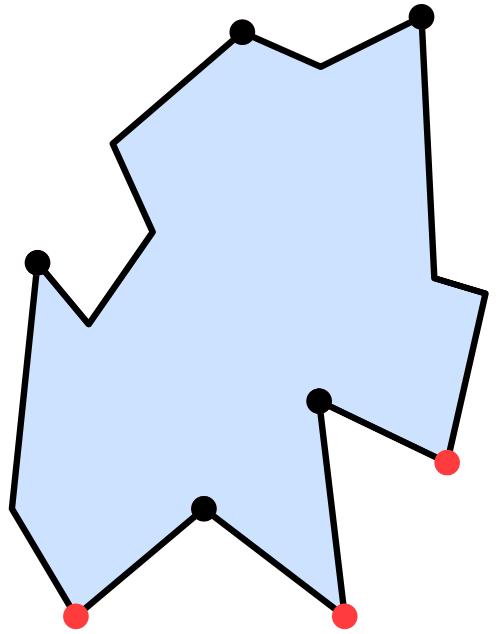


– regular vertex

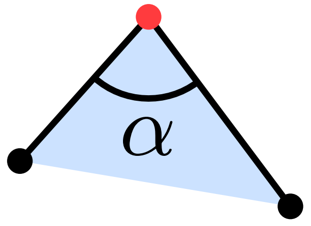
Partition into y-monotone Pieces

Idea: Distinguish 5 types of vertices

– turn vertices: vertical direction switches

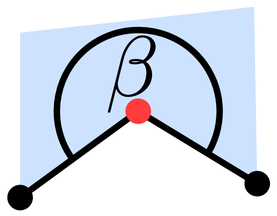


- start vertex



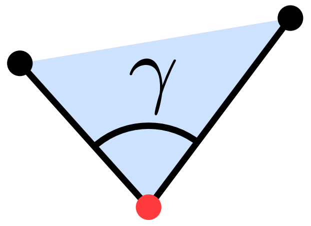
if $\alpha < 180°$

- split vertex



if $\beta > 180°$

- end vertex

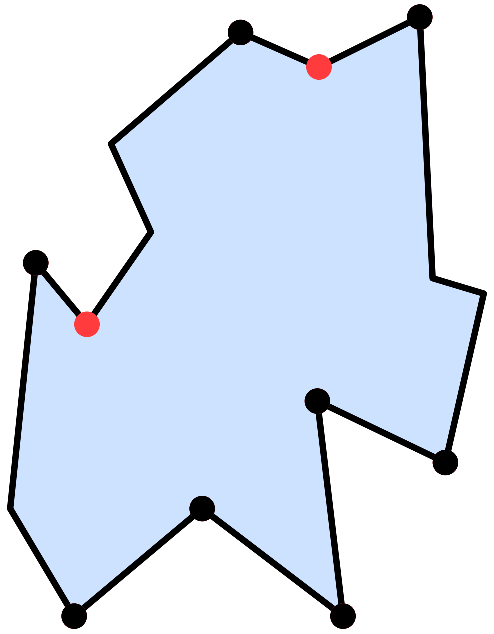


if $\gamma < 180°$

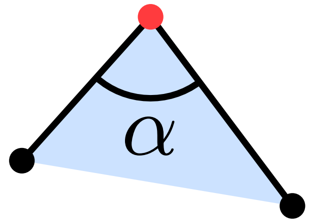
– regular vertex

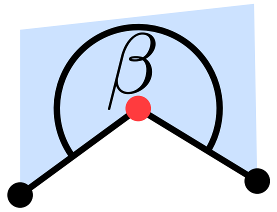
Partition into y-monotone Pieces

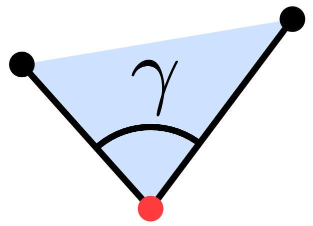
Idea: Distinguish 5 types of vertices

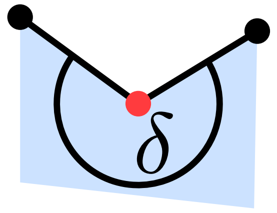


– turn vertices: vertical direction switches

- start vertex 

if $\alpha < 180^\circ$
- split vertex 

if $\beta > 180^\circ$
- end vertex 

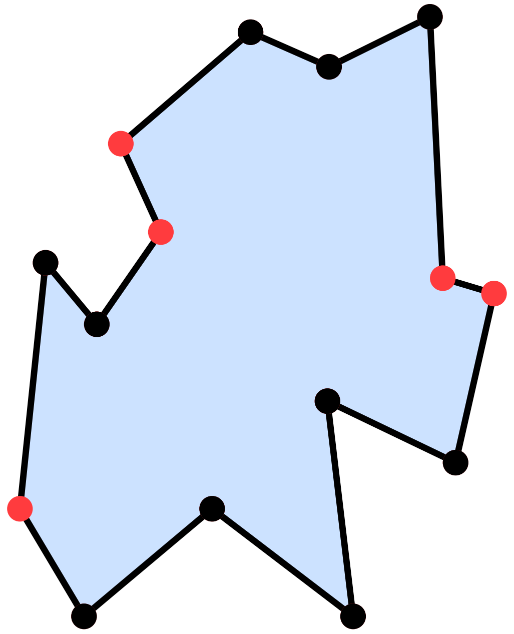
if $\gamma < 180^\circ$
- merge vertex 

if $\delta > 180^\circ$

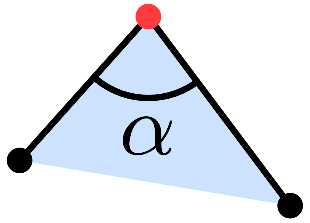
– regular vertex

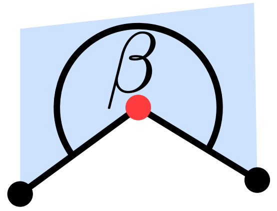
Partition into y-monotone Pieces

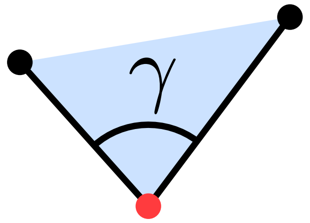
Idea: Distinguish 5 types of vertices

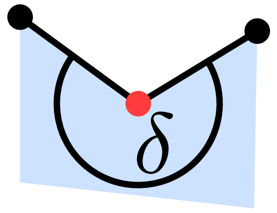


– turn vertices: vertical direction switches

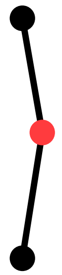
- start vertex 

if $\alpha < 180°$
- split vertex 

if $\beta > 180°$
- end vertex 

if $\gamma < 180°$
- merge vertex 

if $\delta > 180°$

– regular vertex 

Characterization

Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

Characterization

Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

Proof: Suppose P is not y-monotone.

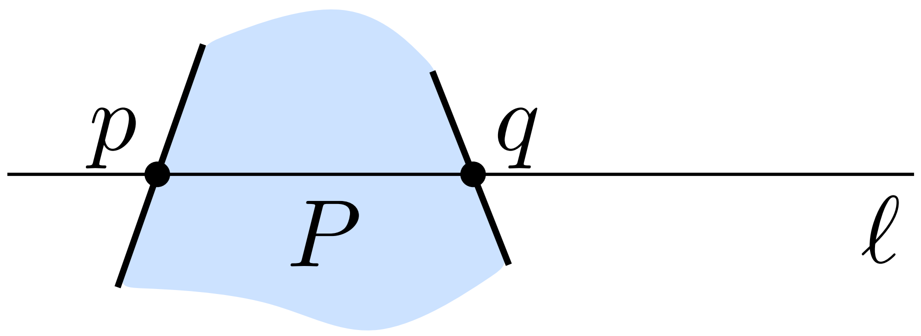
Characterization

Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

Proof: Suppose P is not y-monotone.

Let ℓ be a horizonal line intersecting several components of P.

Cases:



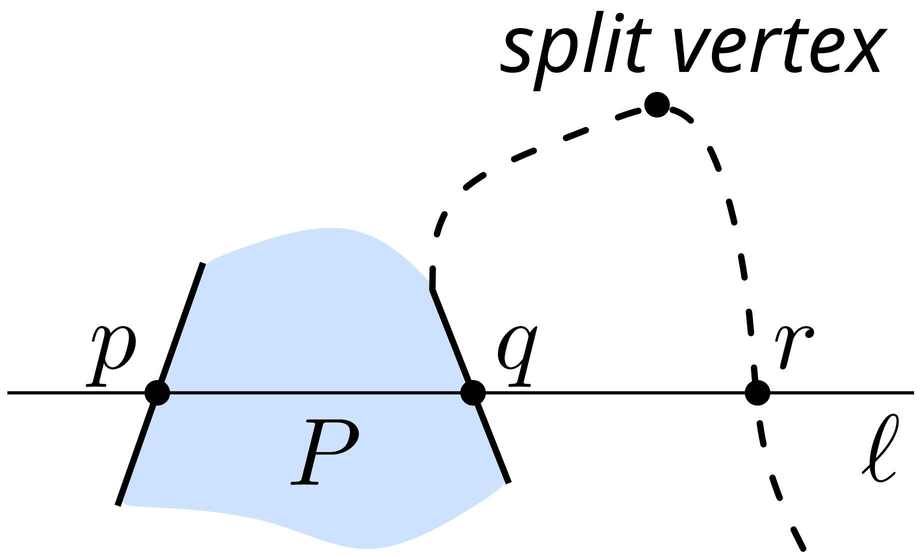
Characterization

Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

Proof: Suppose P is not y-monotone.

Let ℓ be a horizonal line intersecting several components of P.

Cases:



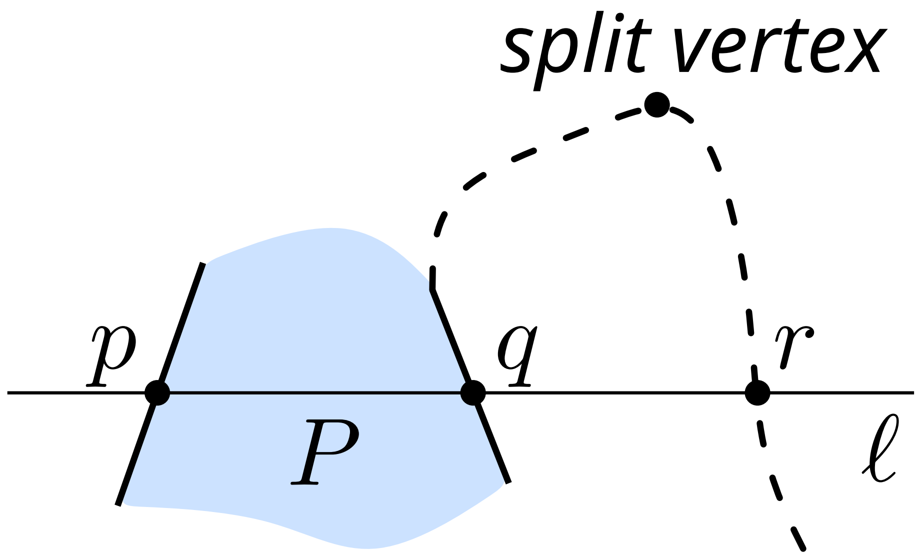
Characterization

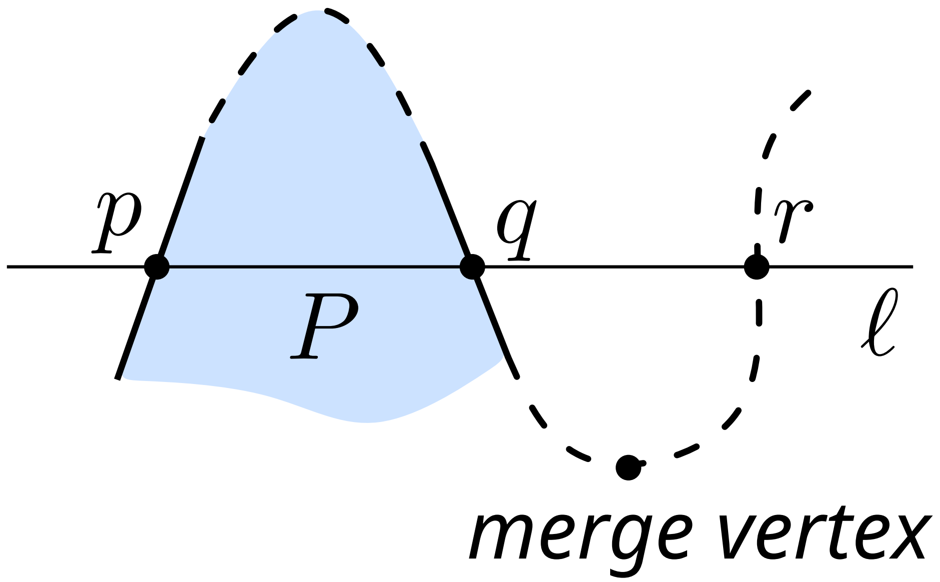
Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

Proof: Suppose P is not y-monotone.

Let ℓ be a horizonal line intersecting several components of P.

Cases:

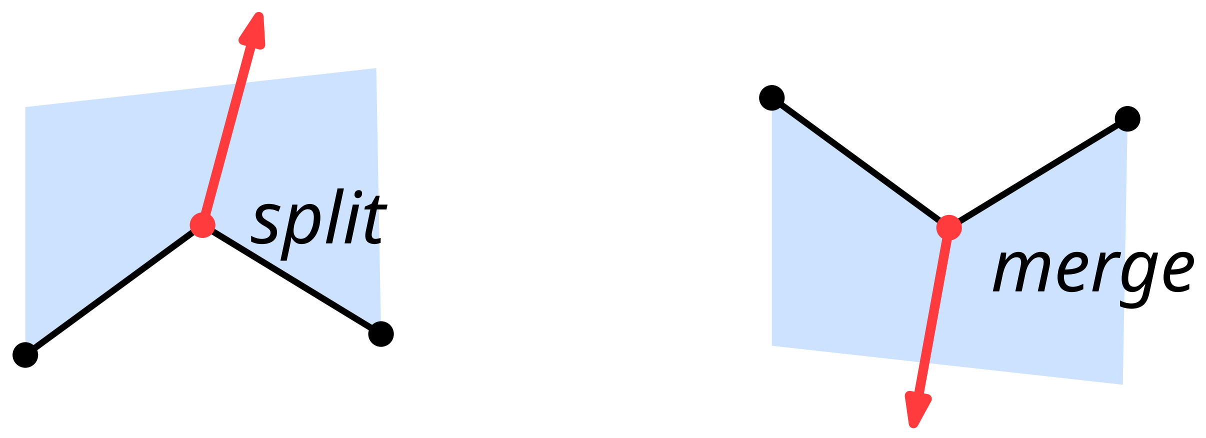




Characterization

Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

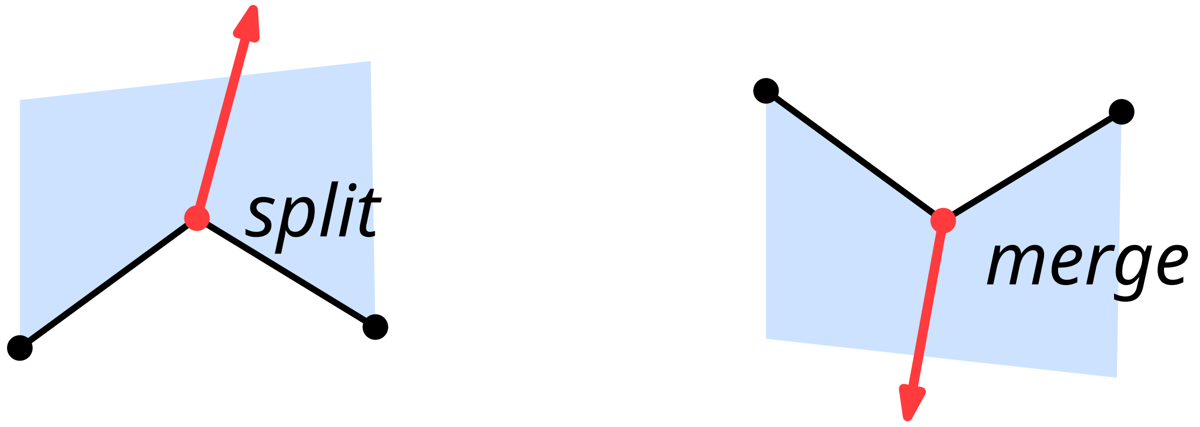
$\Rightarrow$ We need to remove all split and merge vertices by adding diagonals



Characterization

Lemma 1: If a polygon does not contain split and merge vertices then it is y-monotone.

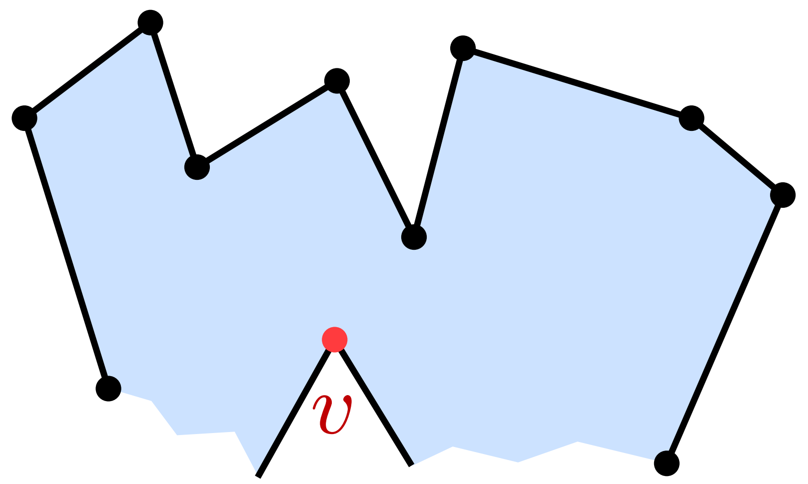
$\Rightarrow$ We need to remove all split and merge vertices by adding diagonals



Careful: Diagonals shouldn't intersect edges of P or other diagonals

Towards a Sweepline Algorithm

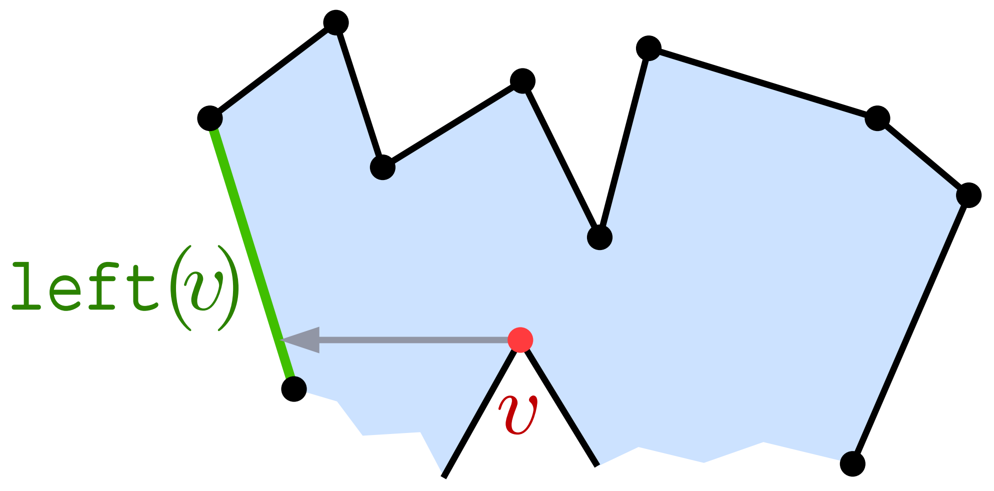
1) Diagonals for split vertices



Towards a Sweepline Algorithm

1) Diagonals for split vertices

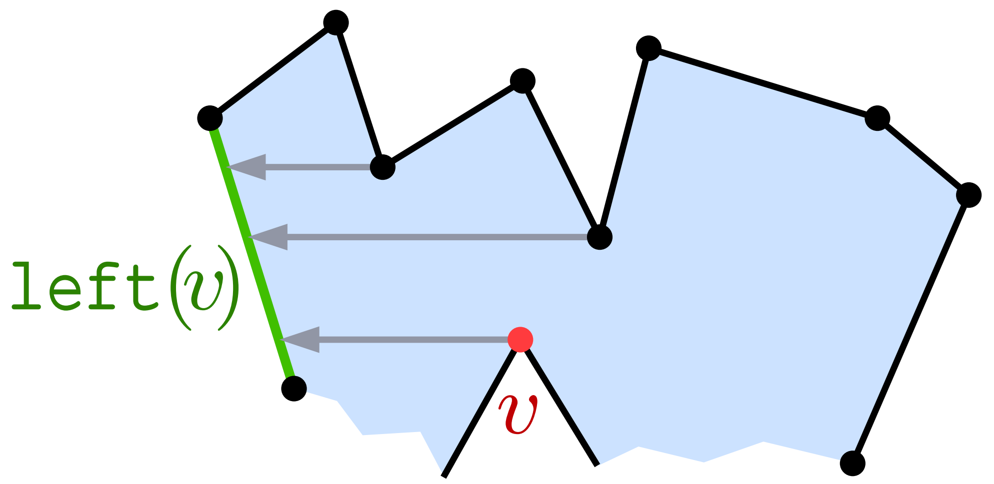
- for every vertex v: compute left edge $\texttt{left}(v)$



Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$

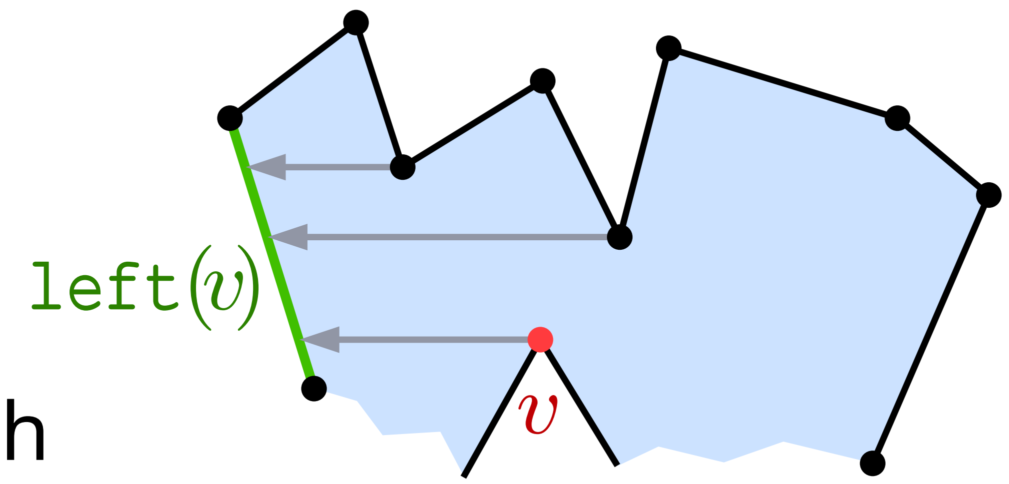


Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$

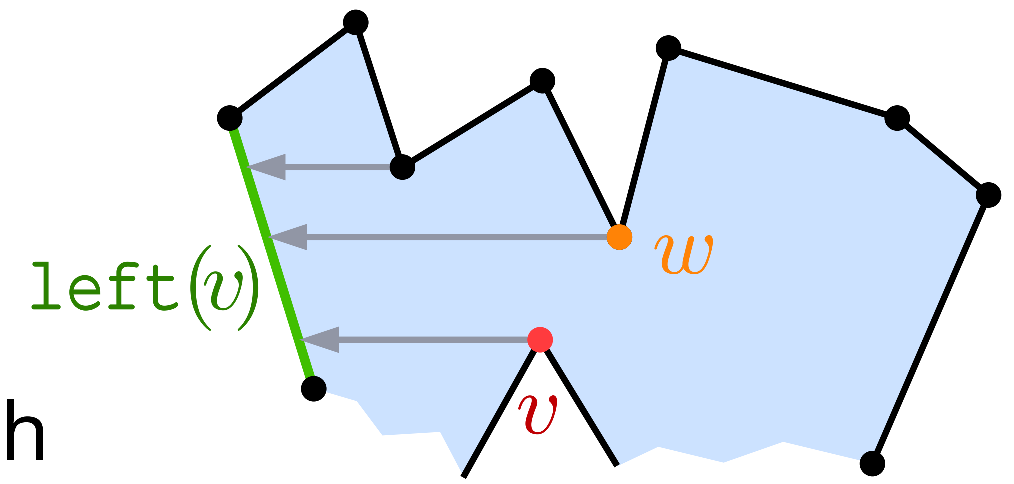
- connect split vertex v to lowest vertex w above v with $\texttt{left}(w) = \texttt{left}(v)$



Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$
- connect split vertex v to lowest vertex w above v with $\texttt{left}(w) = \texttt{left}(v)$

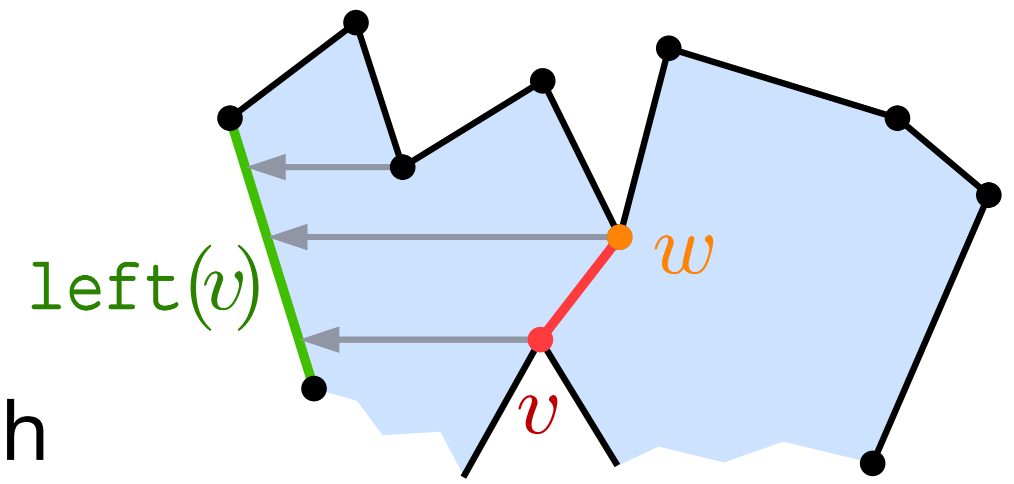


Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$

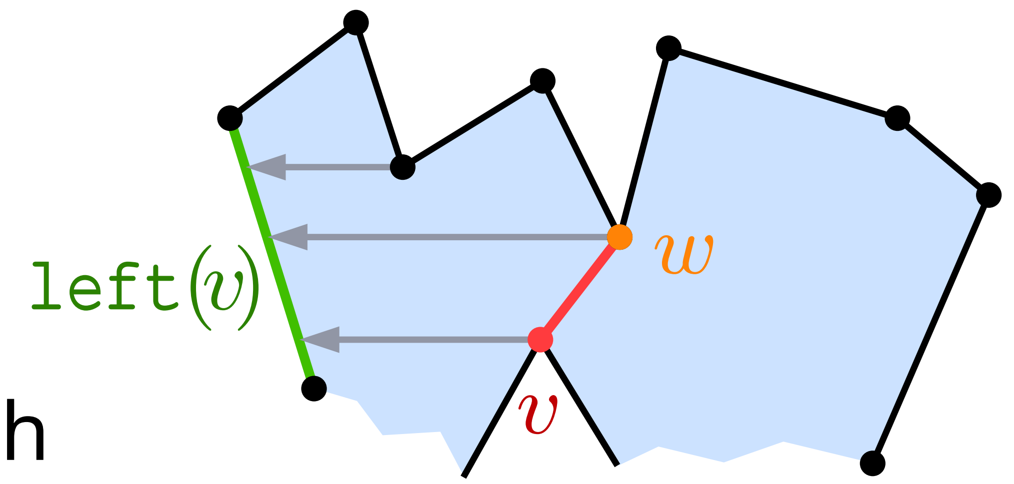
- connect split vertex v to lowest vertex w above v with $\texttt{left}(w) = \texttt{left}(v)$

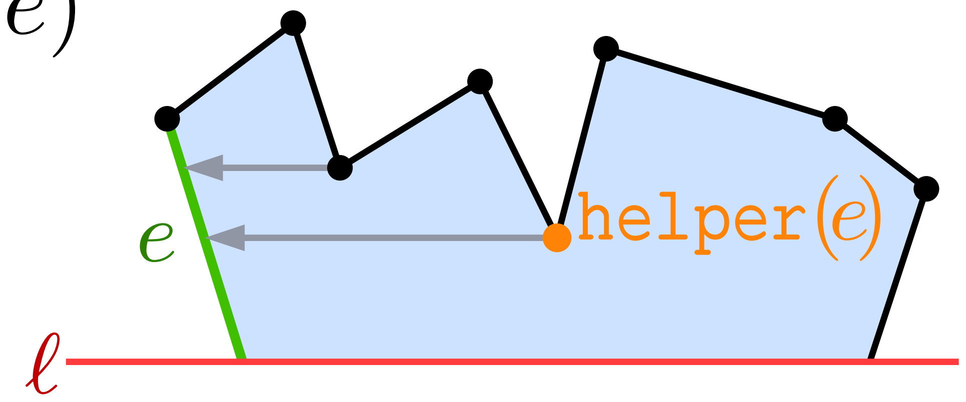


Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$
- connect split vertex v to lowest vertex w above v with $\texttt{left}(w) = \texttt{left}(v)$
- store for every edge e the lowest vertex w as $\texttt{helper}(e)$

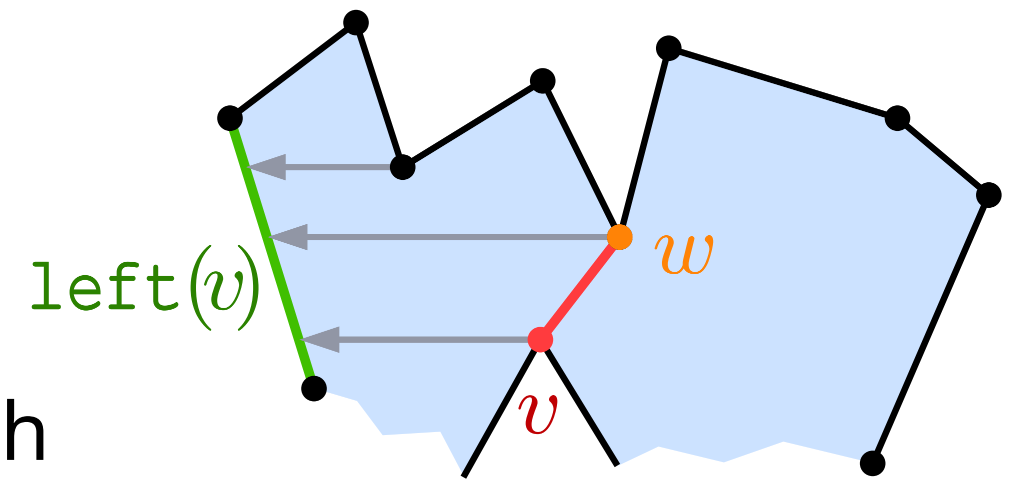


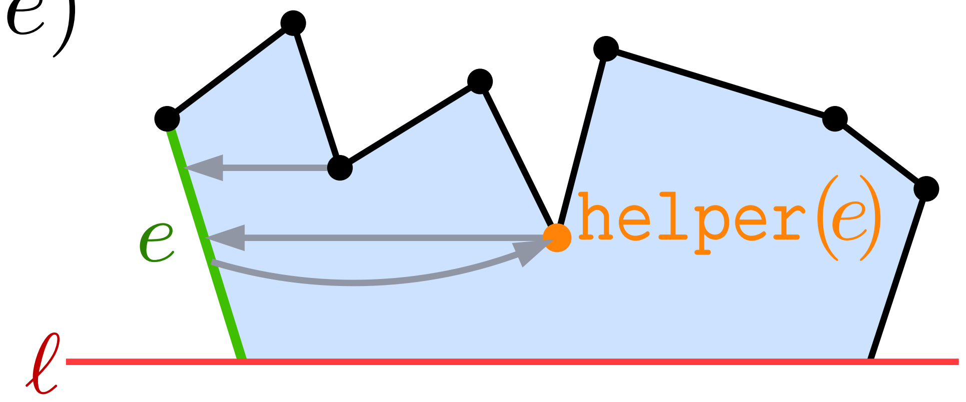


Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$
- connect split vertex v to lowest vertex w above v with $\texttt{left}(w) = \texttt{left}(v)$
- store for every edge e the lowest vertex w as $\texttt{helper}(e)$

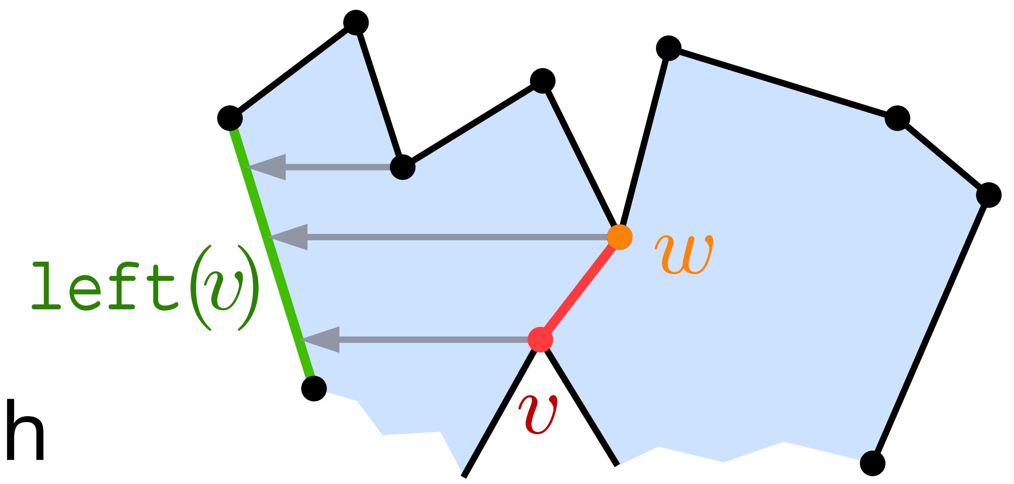


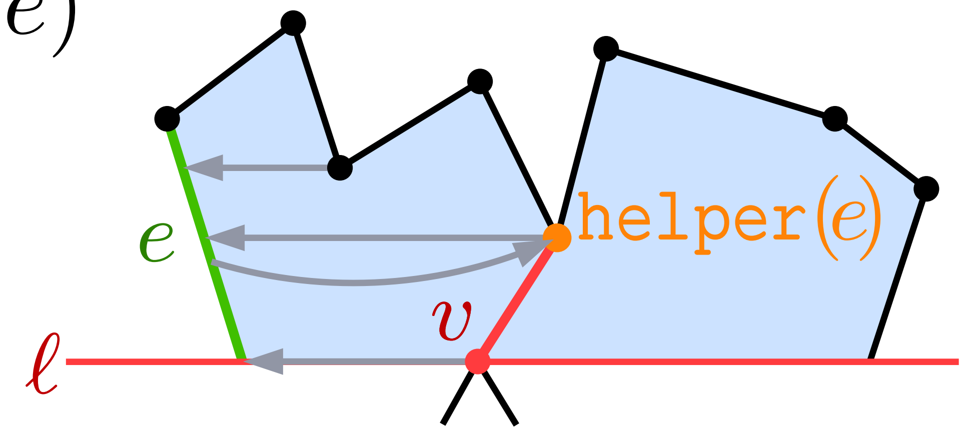


Towards a Sweepline Algorithm

1) Diagonals for split vertices

- for every vertex v: compute left edge $\texttt{left}(v)$
- connect split vertex v to lowest vertex w above v with $\texttt{left}(w) = \texttt{left}(v)$
- store for every edge e the lowest vertex w as $\texttt{helper}(e)$
- when ℓ reaches split node v: connect v to $\texttt{helper}(\texttt{left}(v))$

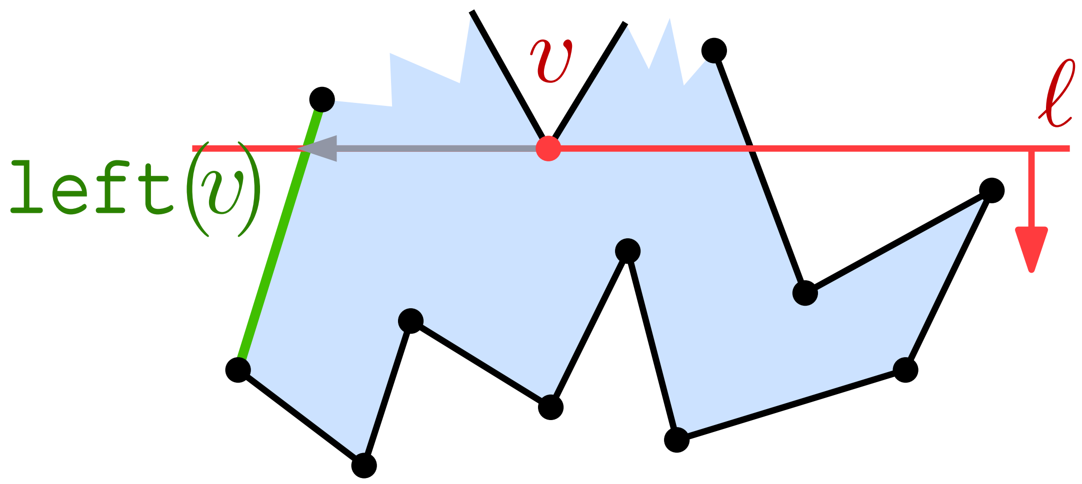




Towards a Sweepline Algorithm

2) Diagonals for merge vertices

- merge vertex v reached:
  update $\texttt{helper}(\texttt{left}(v)) = v$

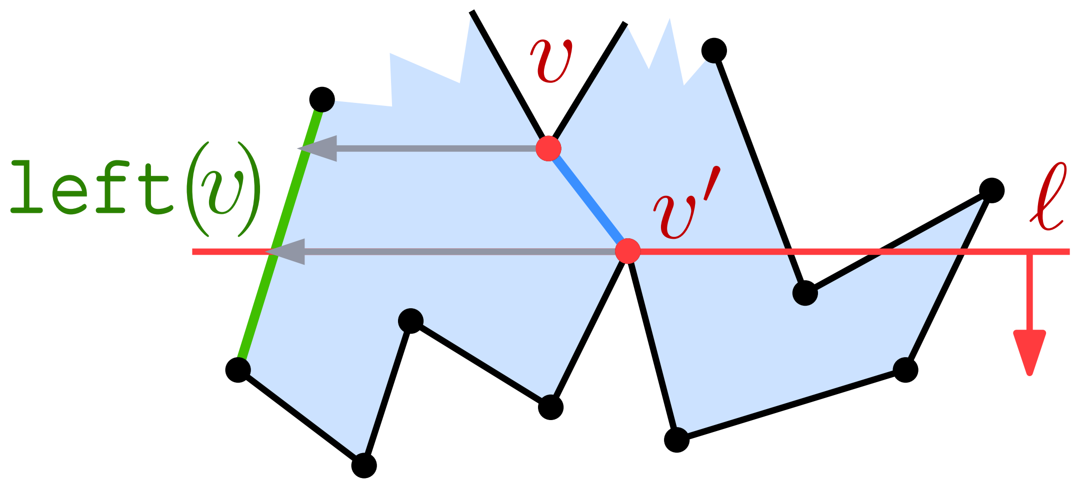


Towards a Sweepline Algorithm

2) Diagonals for merge vertices

- merge vertex v reached:
  update $\texttt{helper}(\texttt{left}(v)) = v$

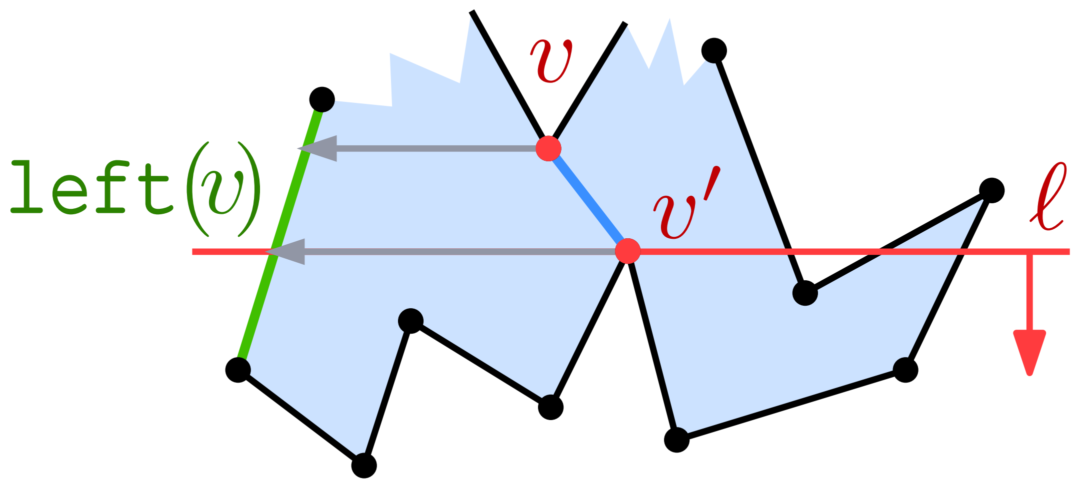
- if split vertex v' with $\texttt{left}(v') = \texttt{left}(v)$ reached next:
  add diagonal (v, v')



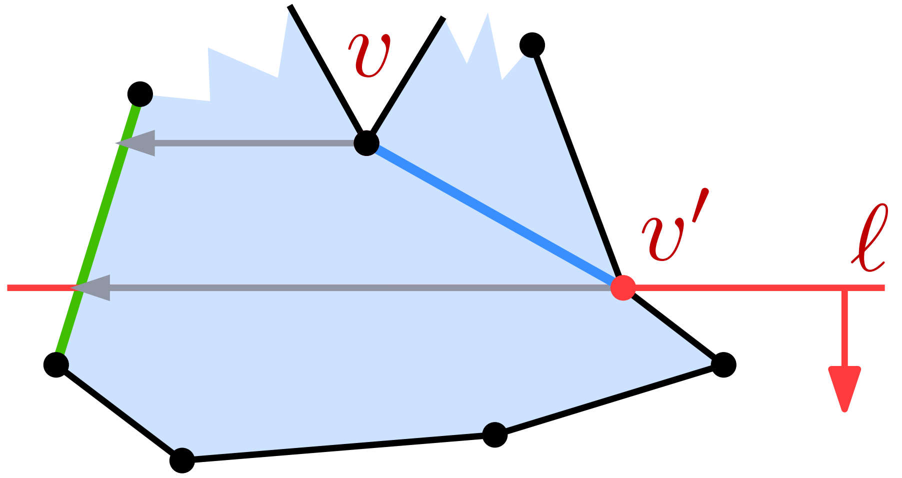
Towards a Sweepline Algorithm

2) Diagonals for merge vertices

- merge vertex v reached:
  update $\texttt{helper}(\texttt{left}(v)) = v$



- if split vertex v' with $\texttt{left}(v') = \texttt{left}(v)$ reached next:
  add diagonal (v, v')

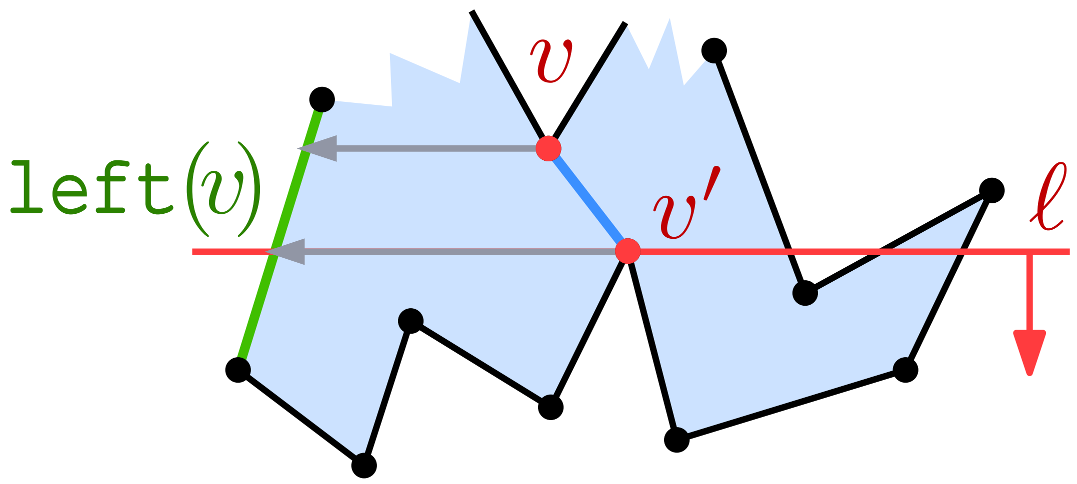


- if $\texttt{helper}(\texttt{left}(v))$ is updated to v':
  add diagonal (v, v')

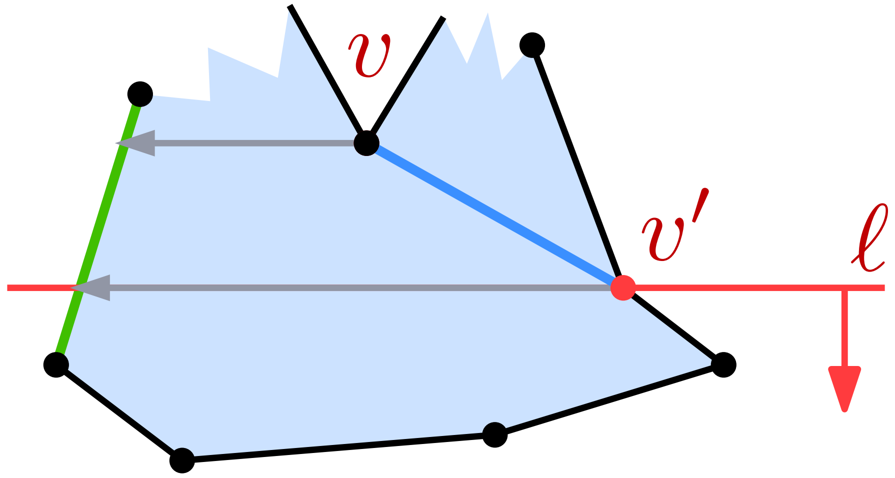
Towards a Sweepline Algorithm

2) Diagonals for merge vertices

- merge vertex v reached:
  update $\texttt{helper}(\texttt{left}(v)) = v$

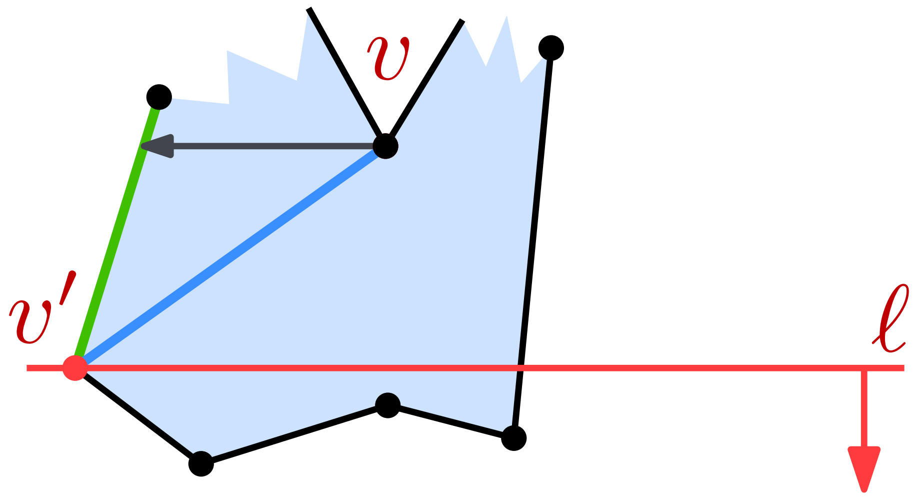


- if split vertex v' with $\texttt{left}(v') = \texttt{left}(v)$ reached next:
  add diagonal (v, v')



- if $\texttt{helper}(\texttt{left}(v))$ is updated to v':
  add diagonal (v, v')

- if end v' of $\texttt{left}(v)$ is reached:
  add diagonal (v, v')



Towards a Sweepline Algorithm

2) Diagonals for merge vertices

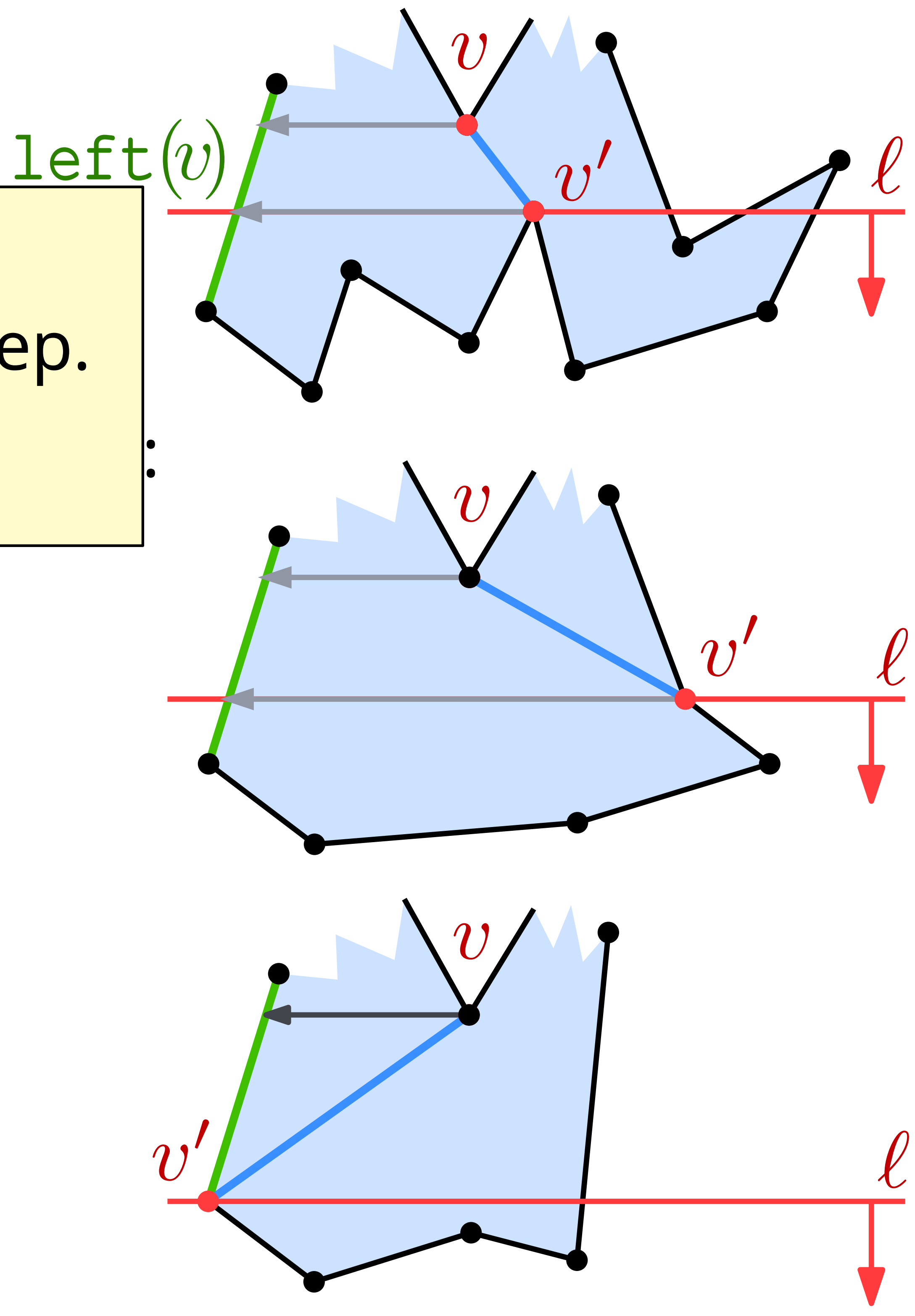
- merge ver
  update he
- if split vert :
  add diagonal (v, v')

Alternative:

Handle merge vertices in separate sweep.

merge = upside-down split

- if $\texttt{helper}(\texttt{left}(v))$ is updated to v':
  add diagonal (v, v')
- if end v' of $\texttt{left}(v)$ is reached:
  add diagonal (v, v')



Towards a Sweepline Algorithm

Events:

Status:

Towards a Sweepline Algorithm

Events: Vertices of P in lexicographical order

Status:

Towards a Sweepline Algorithm

Events: Vertices of P in lexicographical order

Status: Components of P intersected by ℓ: for each component store left edge e and vertex $v = \texttt{helper}(e)$

Algorithm MakeMonotone(P)

Algorithm MAKEMONOTONE(polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $E(P)$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically
$\mathcal{T} \leftarrow \varnothing$ (binary search tree for status of sweepline)
while $\mathcal{Q} \neq \varnothing$ **do**
    $v \leftarrow \mathcal{Q}$.popVertex()
    HANDLEVERTEX(v)
return $\mathcal{D}$

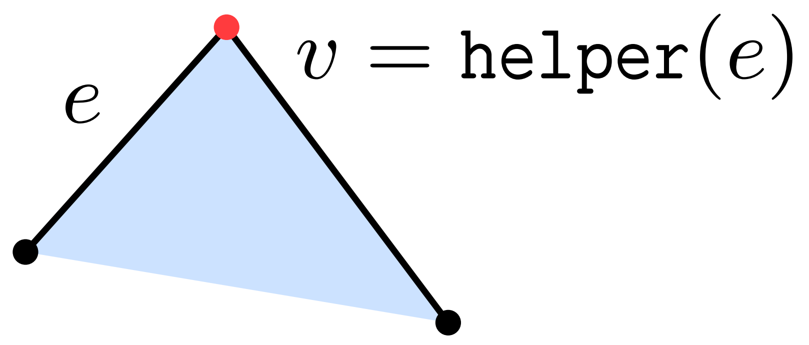
Algorithm MakeMonotone(P)

Algorithm MAKEMONOTONE(polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $E(P)$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically
$\mathcal{T} \leftarrow \emptyset$ (binary search tree for status of sweepline)
while $\mathcal{Q} \neq \emptyset$ **do**
    $v \leftarrow \mathcal{Q}$.popVertex()
    HANDLEVERTEX(v)
return $\mathcal{D}$

HANDLESTARTVERTEX(vertex v)

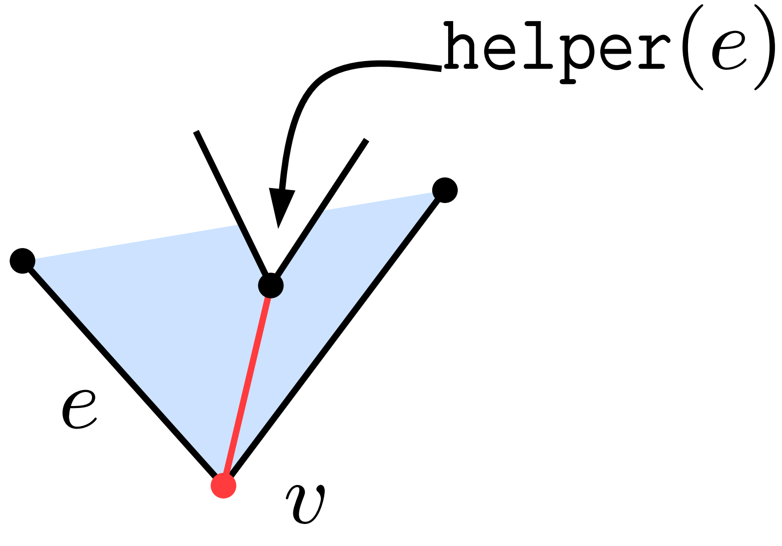
$\mathcal{T} \leftarrow$ insert left edge e
$\texttt{helper}(e) \leftarrow v$



Algorithm MakeMonotone(P)

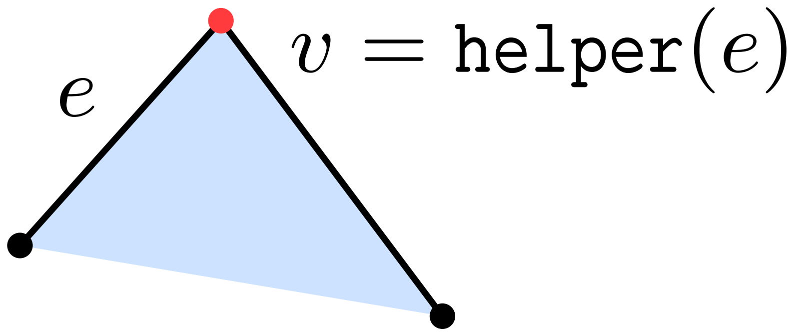
Algorithm MAKEMONOTONE(polygon P)

1: $\mathcal{D} \leftarrow$ doubly-connected edge list for $E(P)$
2: $\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically
3: $\mathcal{T} \leftarrow \varnothing$ (binary search tree for status of sweepline)
4: **while** $\mathcal{Q} \neq \varnothing$ **do**
5: $\quad v \leftarrow \mathcal{Q}$.popVertex()
6: $\quad$ HANDLEVERTEX(v)
7: **return** $\mathcal{D}$



HANDLESTARTVERTEX(vertex v)

1: $\mathcal{T} \leftarrow$ insert left edge e
2: $\texttt{helper}(e) \leftarrow v$



HANDLEENDVERTEX(vertex v)

1: $e \leftarrow$ left edge
2: **if** isMergeVertex($\texttt{helper}(e)$) **then**
3: $\quad \mathcal{D} \leftarrow$ insert $(\texttt{helper}(e), v)$
4: delete e from $\mathcal{T}$

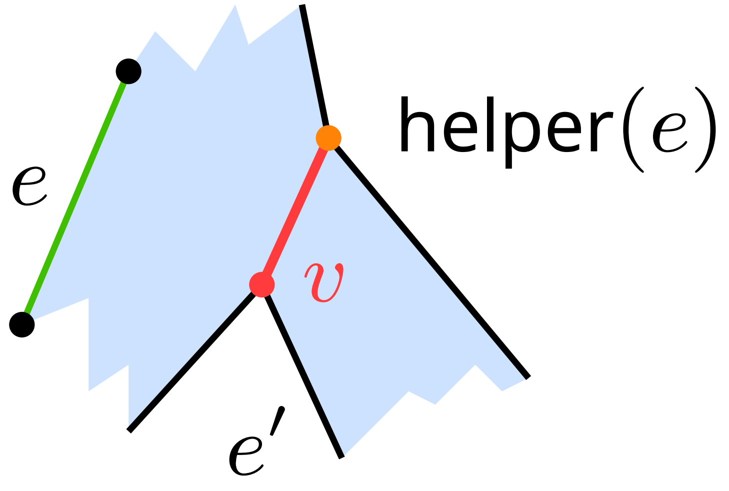
Algorithm MakeMonotone(P)

Algorithm MAKEMONOTONE(polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $E(P)$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically
$\mathcal{T} \leftarrow \varnothing$ (binary search tree for status of sweepline)
while $\mathcal{Q} \neq \varnothing$ **do**
    $v \leftarrow \mathcal{Q}$.popVertex()
    HANDLEVERTEX(v)
return $\mathcal{D}$

HANDLESPLITVERTEX(vertex v)

$e \leftarrow$ edge left of v in $\mathcal{T}$
$\mathcal{D} \leftarrow$ insert $(\texttt{helper}(e), v)$
$\texttt{helper}(e) \leftarrow v$
$\mathcal{T} \leftarrow$ insert right edge e' of v
$\texttt{helper}(e') \leftarrow v$



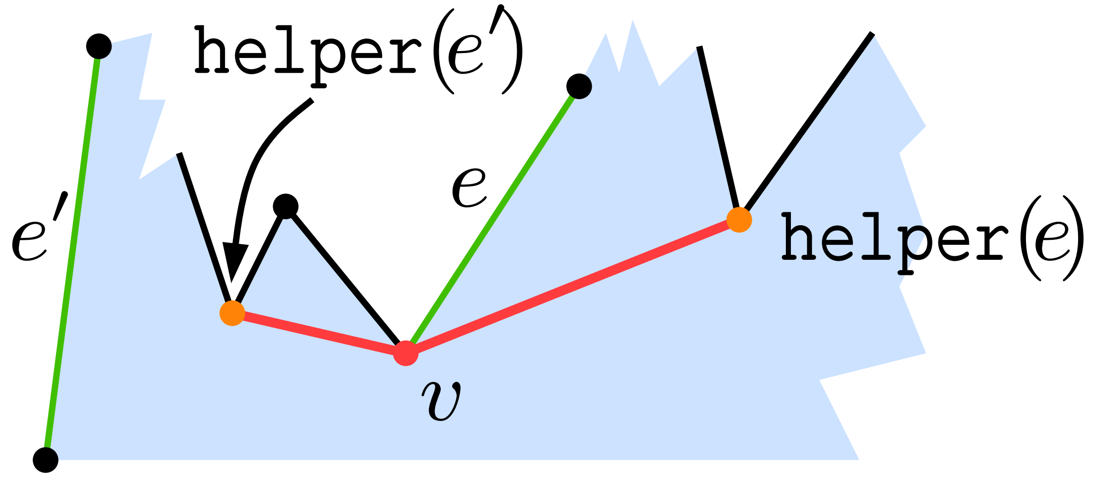
Algorithm MakeMonotone(P)

Algorithm MAKEMONOTONE(polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $E(P)$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically
$\mathcal{T} \leftarrow \varnothing$ (binary search tree for status of sweepline)
while $\mathcal{Q} \neq \varnothing$ **do**
    $v \leftarrow \mathcal{Q}$.popVertex()
    HANDLEVERTEX(v)
return $\mathcal{D}$

HANDLEMERGEVERTEX(vertex v)

$e \leftarrow$ right edge
if isMergeVertex($\texttt{helper}(e)$) **then**
    $\mathcal{D} \leftarrow$ insert $(\texttt{helper}(e), v)$
delete e from $\mathcal{T}$
$e' \leftarrow$ edge left of v in $\mathcal{T}$
if isMergeVertex($\texttt{helper}(e')$) **then**
    $\mathcal{D} \leftarrow$ insert $(\texttt{helper}(e'), v)$
$\texttt{helper}(e') \leftarrow v$



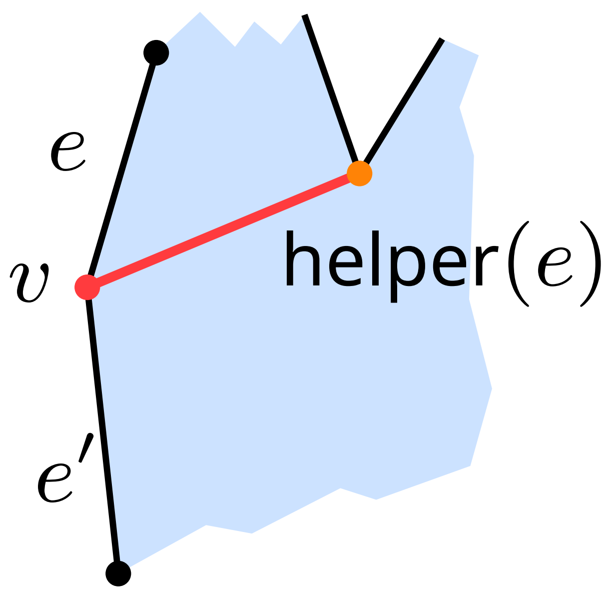
Algorithm MakeMonotone(P)

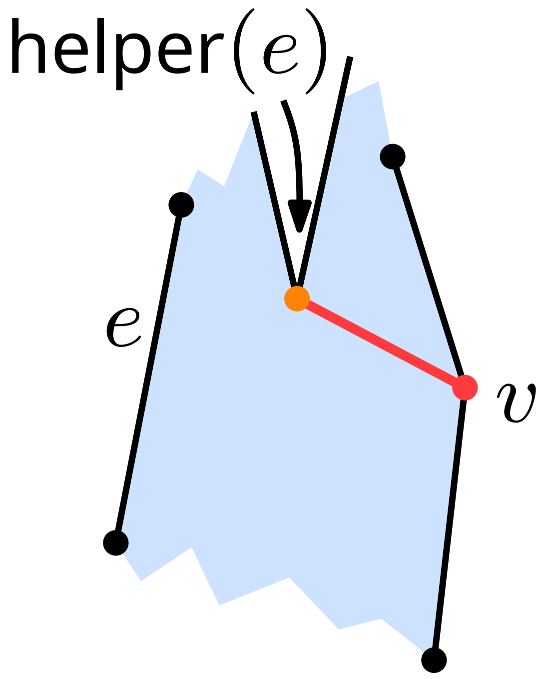
Algorithm MAKEMONOTONE(polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $E(P)$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically
$\mathcal{T} \leftarrow \varnothing$ (binary search tree for status of sweepline)
while $\mathcal{Q} \neq \varnothing$ **do**
    $v \leftarrow \mathcal{Q}$.popVertex()
    HANDLEVERTEX(v)
return $\mathcal{D}$

HANDLEREGULARVERTEX(vertex v)

if P lies locally right of v **then**
    $e, e' \leftarrow$ upper, lower edge
    if isMergeVertex(helper(e)) **then**
        $\mathcal{D} \leftarrow$ insert (helper$(e), v$)
    delete e from $\mathcal{T}$
    $\mathcal{T} \leftarrow$ insert e'; helper$(e') \leftarrow v$
else
    $e \leftarrow$ edge left of v in $\mathcal{T}$
    if isMergeVertex(helper(e)) **then**
        $\mathcal{D} \leftarrow$ insert (helper$(e), v$)
    helper$(e) \leftarrow v$





Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Proof:

- all split and merge vertices are removed

Analysis

Lemma 2: Algorithm MakeMonotone inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Proof:

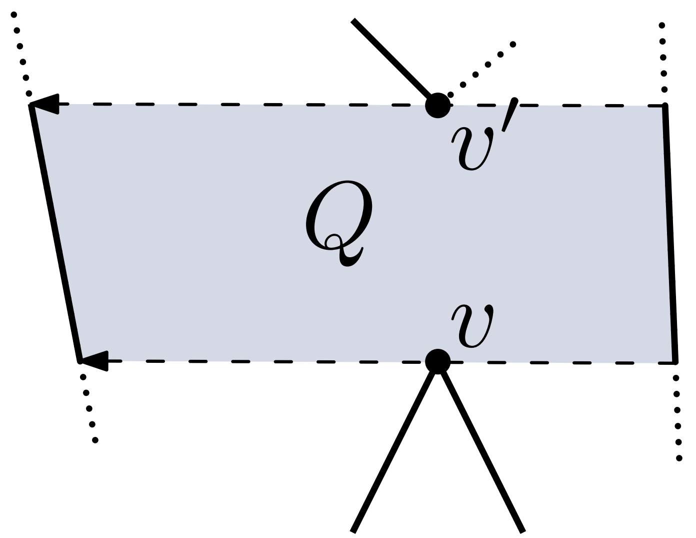
- all split and merge vertices are removed
- diagonals are crossing free

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Proof:

- all split and merge vertices are removed
- diagonals are crossing free
  - no edges and diagonals cross $Q \Rightarrow$ safe to add (v, v')

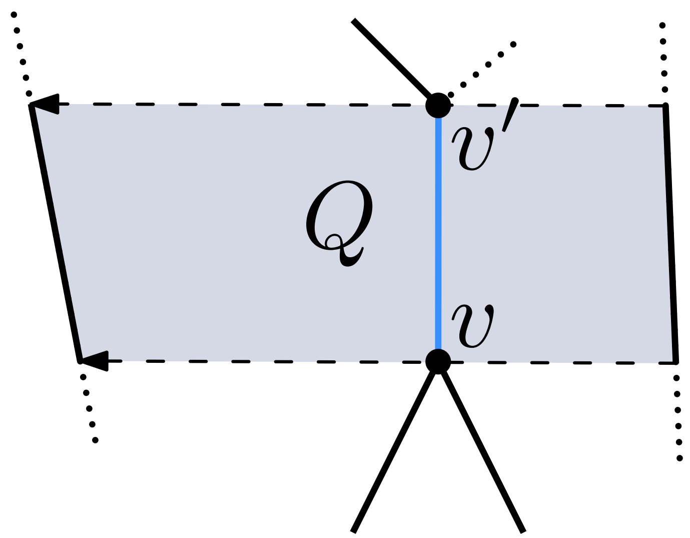


Analysis

Lemma 2: Algorithm MakeMonotone inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Proof:

- all split and merge vertices are removed
- diagonals are crossing free
  - no edges and diagonals cross $Q \Rightarrow$ safe to add (v, v')



Analysis

Lemma 2: Algorithm MakeMonotone inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(\quad ? \quad)$ time using $O(?)$ space into y-monotone polygons.

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(\quad ? \quad)$ time using $O(?)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$
- initialize status $\mathcal{T}$
- time per event
  - $\mathcal{Q}$.deleteMax
  - search, delete, insert elements of $\mathcal{T}$
  - add ≤ 2 diagonals to $\mathcal{D}$
- space:

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(\quad ? \quad)$ time using $O(?)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ $O(n)$ time
- initialize status $\mathcal{T}$
- time per event
  - $\mathcal{Q}$.deleteMax
  - search, delete, insert elements of $\mathcal{T}$
  - add ≤ 2 diagonals to $\mathcal{D}$
- space:

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O($? $)$ time using $O($?$)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event
  - $\mathcal{Q}$.deleteMax
  - search, delete, insert elements of $\mathcal{T}$
  - add ≤ 2 diagonals to $\mathcal{D}$
- space:

Analysis

Lemma 2: Algorithm MakeMonotone inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O($? $)$ time using $O($?$)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event
  - $\mathcal{Q}$.deleteMax — $O(\log n)$ time
  - search, delete, insert elements of $\mathcal{T}$
  - add ≤ 2 diagonals to $\mathcal{D}$
- space:

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(\quad ? \quad)$ time using $O(?)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event
  - $\mathcal{Q}$.deleteMax — $O(\log n)$ time
  - search, delete, insert elements of $\mathcal{T}$ — $O(\log n)$ time
  - add ≤ 2 diagonals to $\mathcal{D}$
- space:

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O($? $)$ time using $O($? $)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event
  - $\mathcal{Q}$.deleteMax — $O(\log n)$ time
  - search, delete, insert elements of $\mathcal{T}$ — $O(\log n)$ time
  - add ≤ 2 diagonals to $\mathcal{D}$ — $O(1)$ time
- space:

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(\quad ? \quad)$ time using $O(?)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event — $O(\log n)$ time
  - $\mathcal{Q}$.deleteMax — $O(\log n)$ time
  - search, delete, insert elements of $\mathcal{T}$ — $O(\log n)$ time
  - add ≤ 2 diagonals to $\mathcal{D}$ — $O(1)$ time
- space:

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O($? $)$ time using $O($?$)$ space into y-monotone polygons.

• create priority queue $\mathcal{Q}$	$O(n)$ time
• initialize status $\mathcal{T}$	$O(1)$ time
• time per event	$O(\log n)$ time
– $\mathcal{Q}$.deleteMax	$O(\log n)$ time
– search, delete, insert elements of $\mathcal{T}$	$O(\log n)$ time
– add ≤ 2 diagonals to $\mathcal{D}$	$O(1)$ time
• space:	$O(n)$

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(\quad ? \quad)$ time using $O(?)$ space into y-monotone polygons.

- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event — $O(\log n)$ time
  - $\mathcal{Q}$.deleteMax — $O(\log n)$ time
  - search, delete, insert elements of $\mathcal{T}$ — $O(\log n)$ time
  - add ≤ 2 diagonals to $\mathcal{D}$ — $O(1)$ time
- space: *obviously?* $O(n)$

Analysis

Lemma 2: Algorithm MAKEMONOTONE inserts a set of crossing-free diagonals in P that partition P into y-monotone polygons.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(n \log n)$ time using $O(n)$ space into y-monotone polygons.

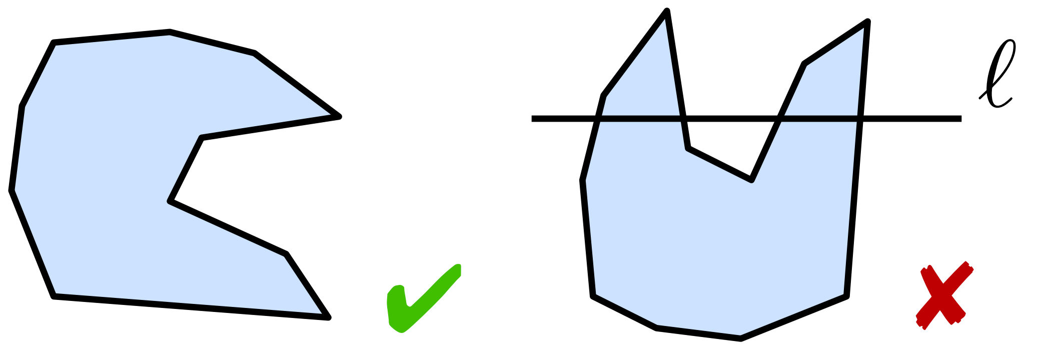
- create priority queue $\mathcal{Q}$ — $O(n)$ time
- initialize status $\mathcal{T}$ — $O(1)$ time
- time per event — $O(\log n)$ time
  - $\mathcal{Q}$.deleteMax — $O(\log n)$ time
  - search, delete, insert elements of $\mathcal{T}$ — $O(\log n)$ time
  - add ≤ 2 diagonals to $\mathcal{D}$ — $O(1)$ time
- space: — $O(n)$

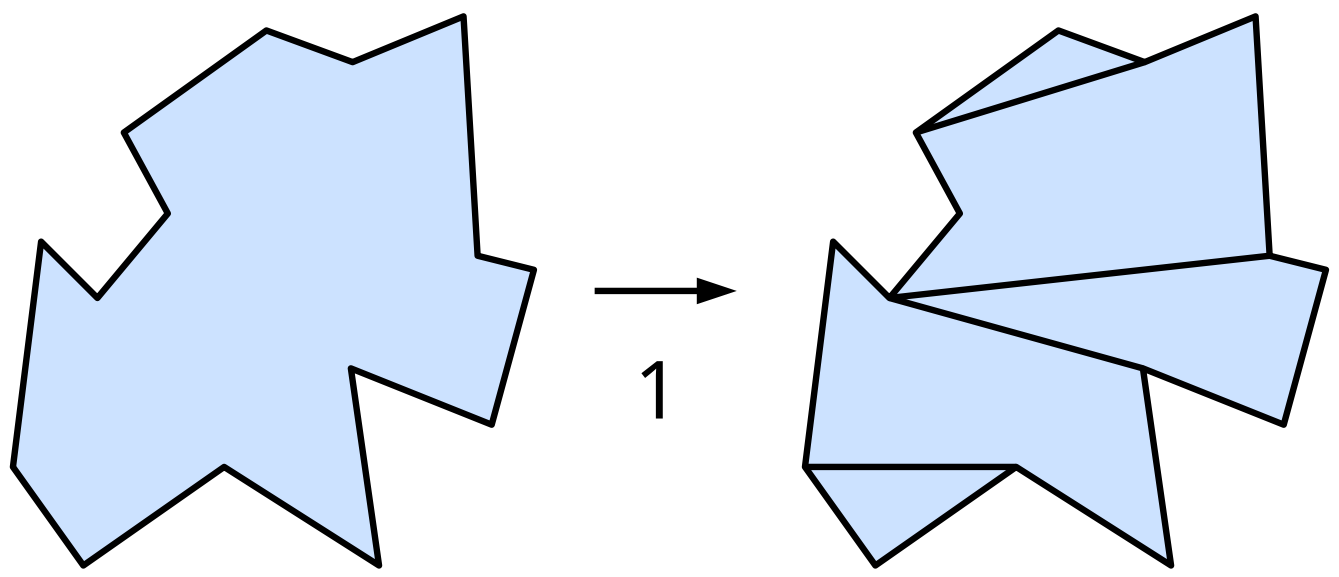
Triangulation: Overview

2-step procedure:

- step 1: partition P into y-monotone subpolygons

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.



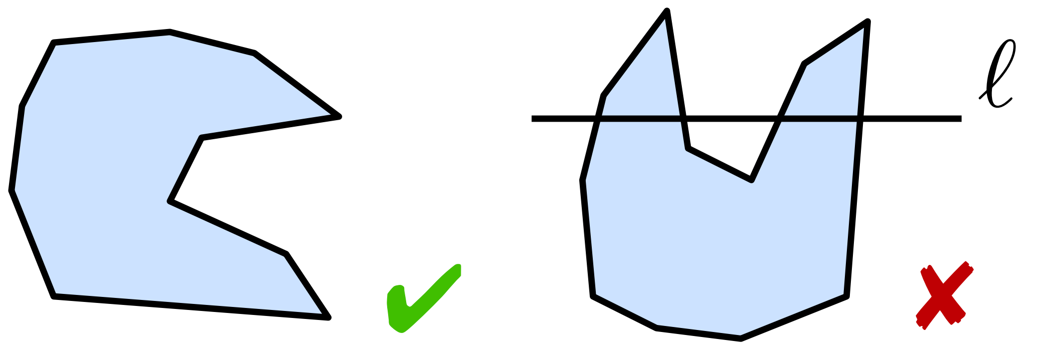


Triangulation: Overview

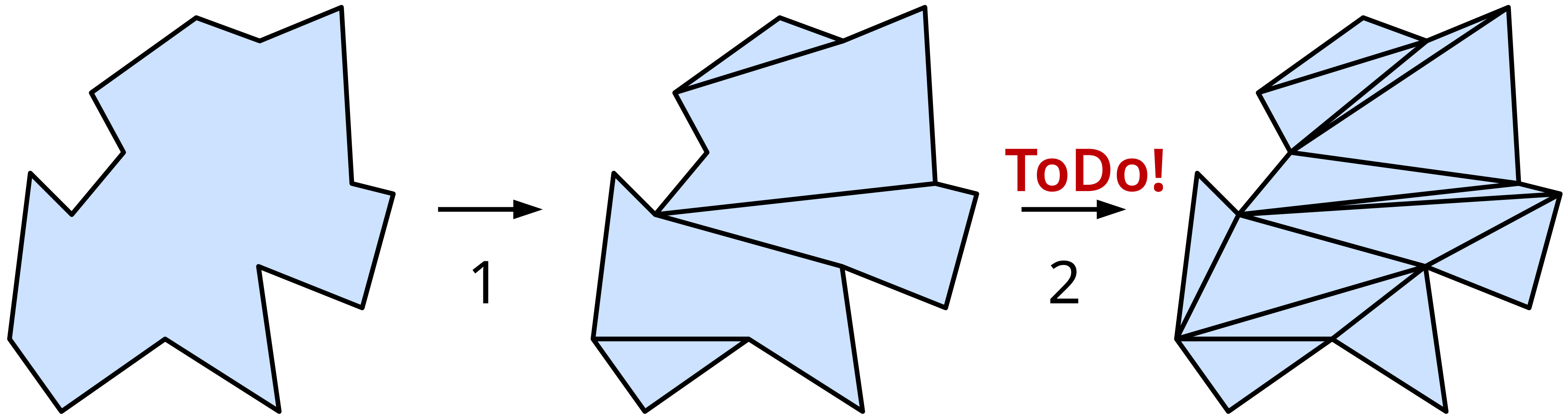
2-step procedure:

- step 1: partition P into y-monotone subpolygons

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.



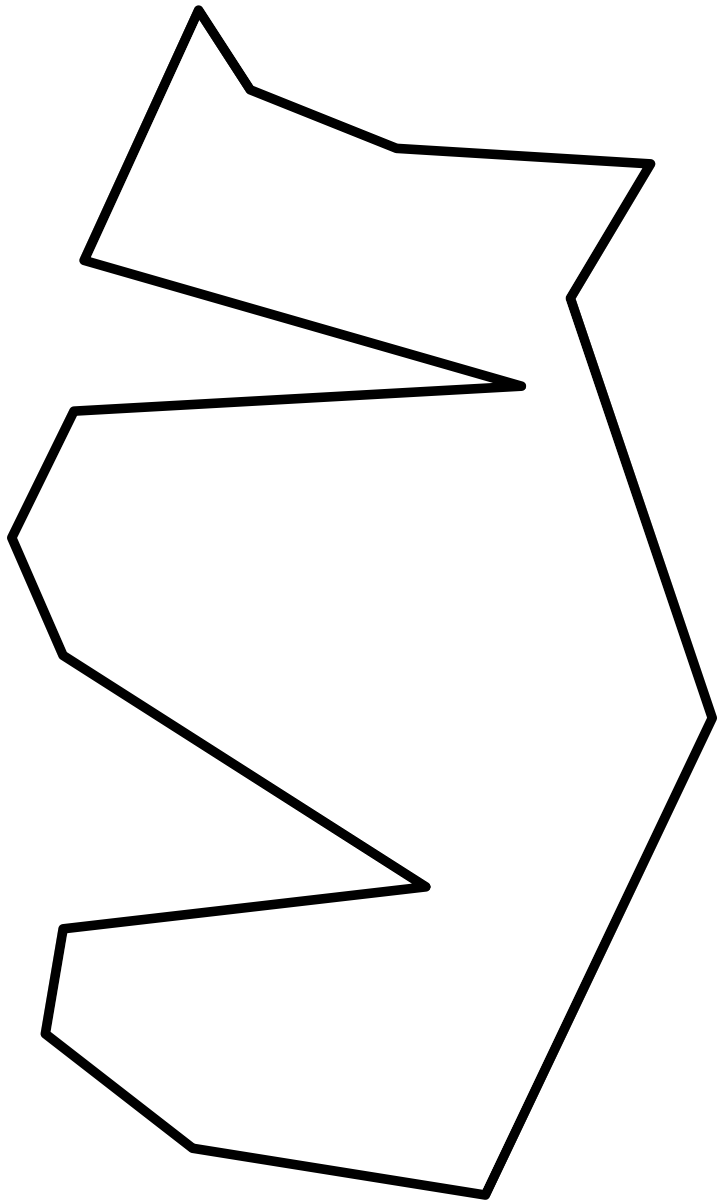
- step 2: triangulate y-monotone subpolyons



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

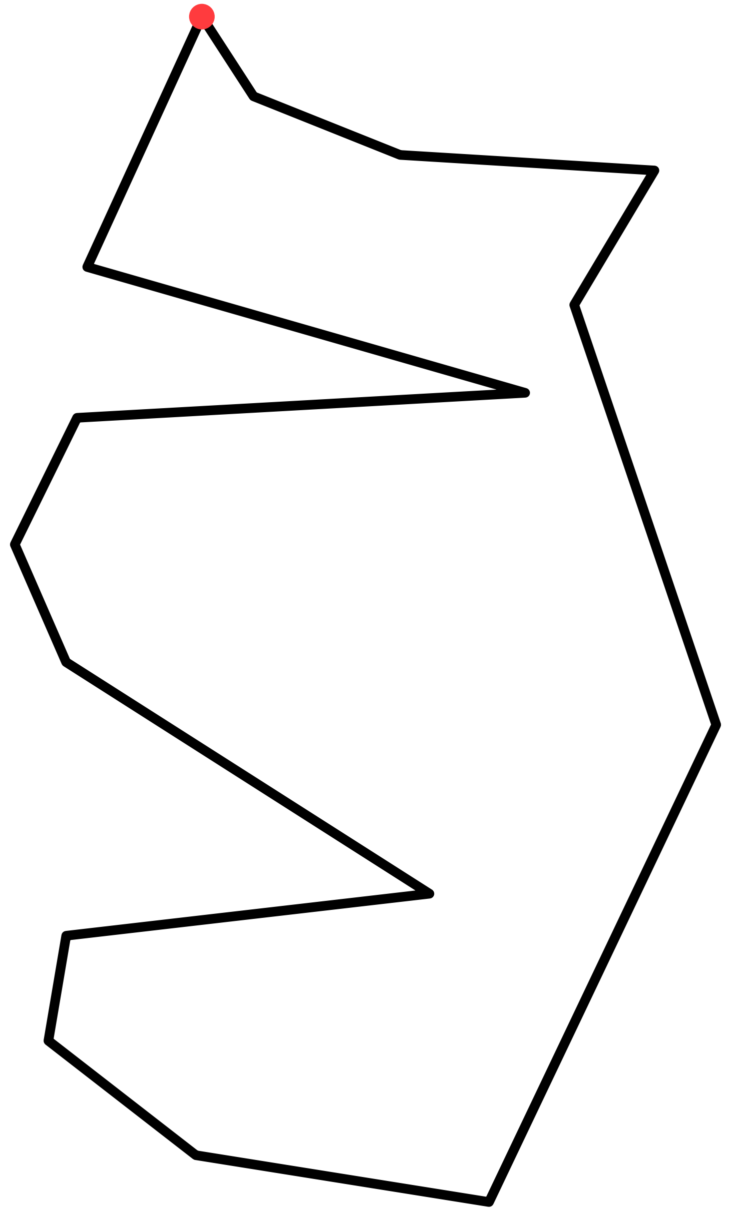
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

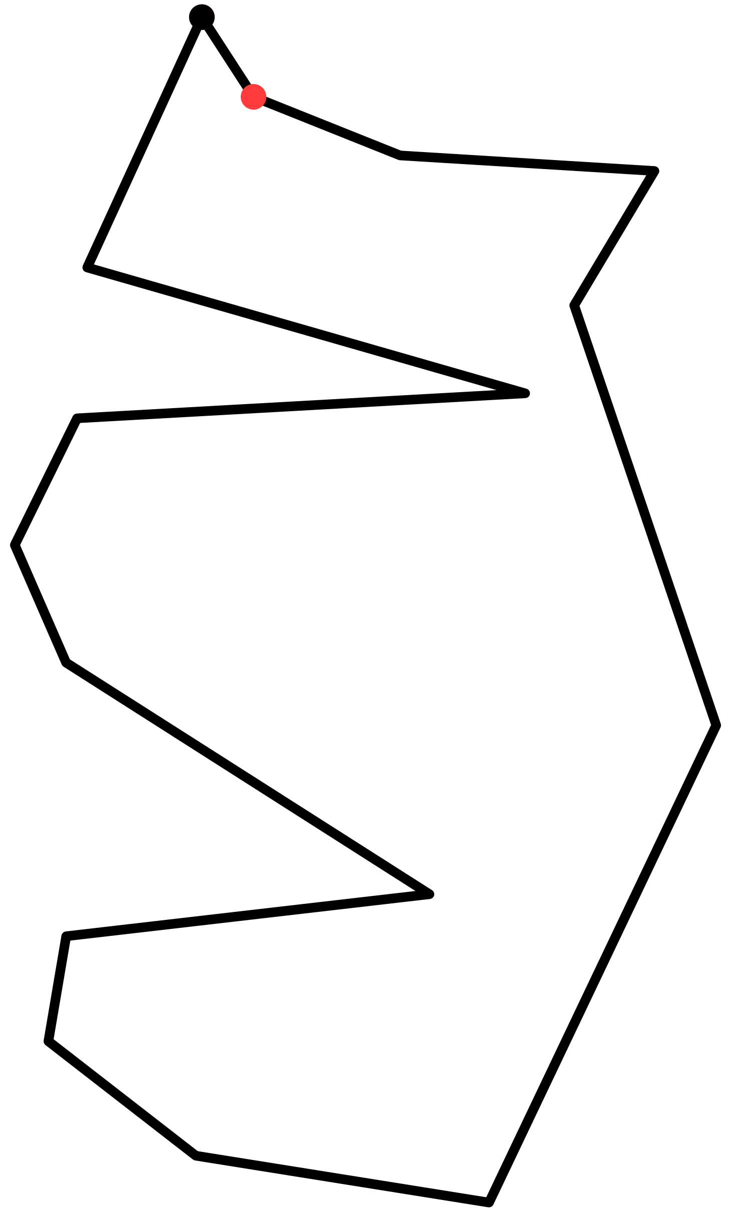
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

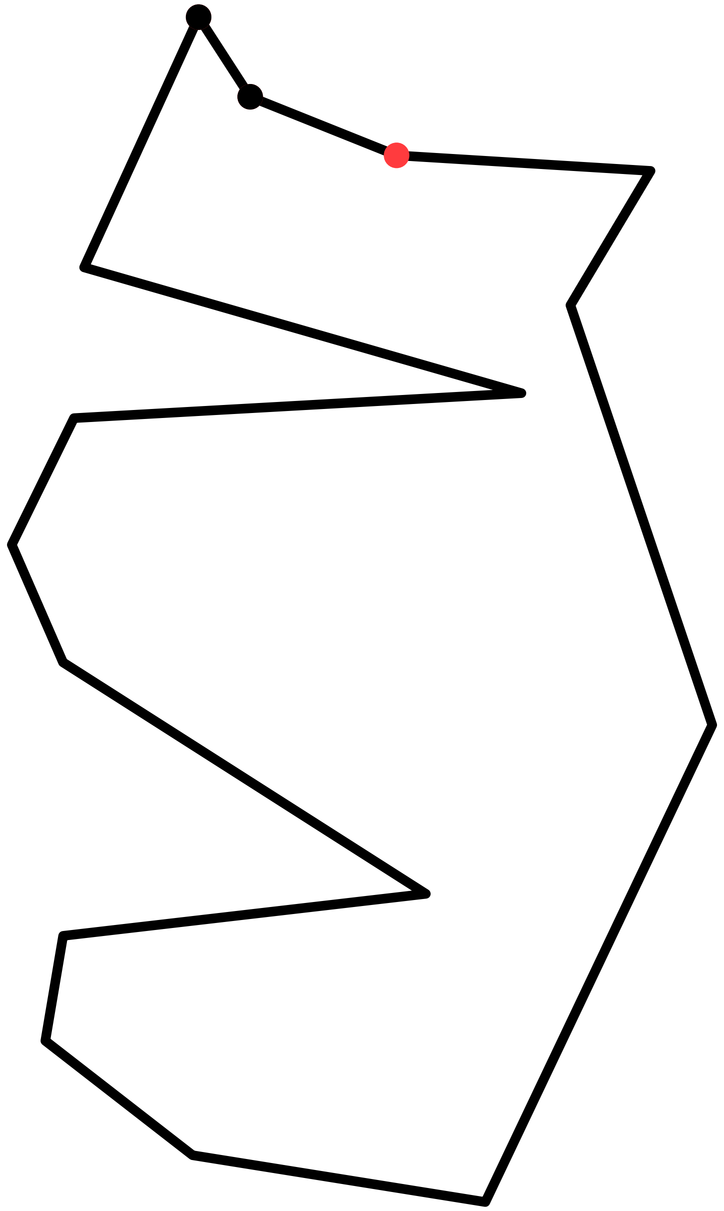
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

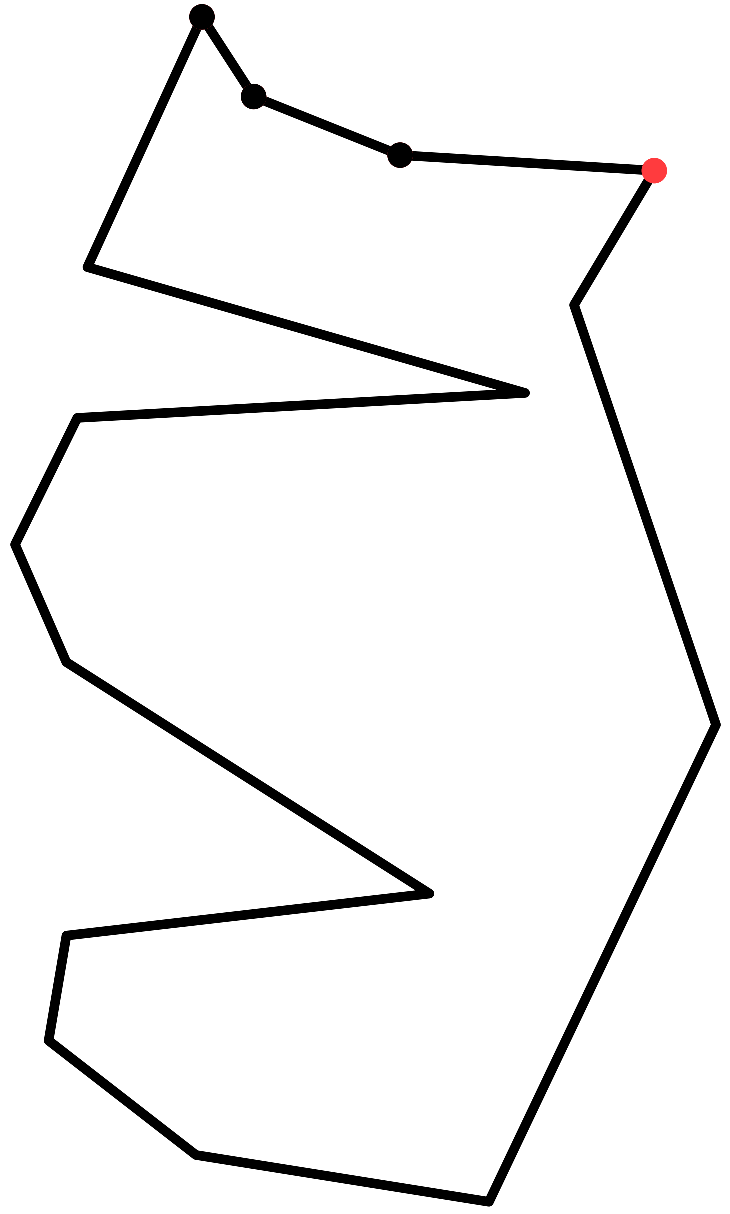
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

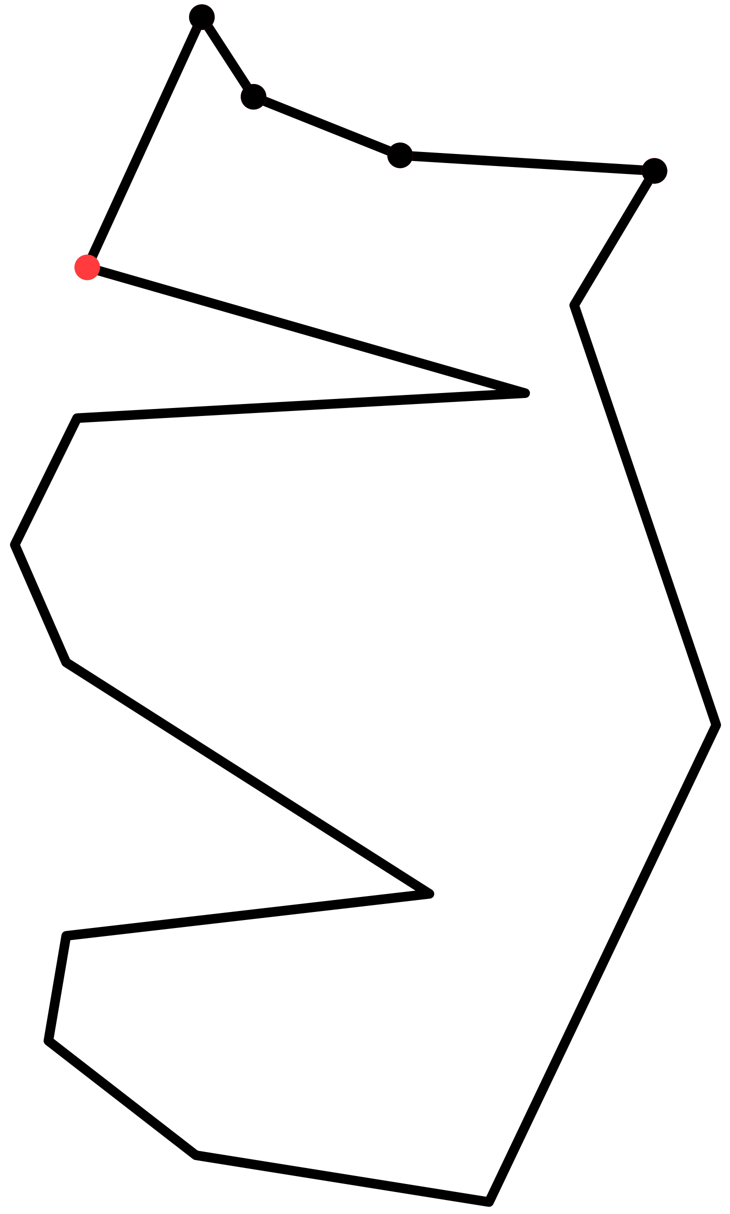
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

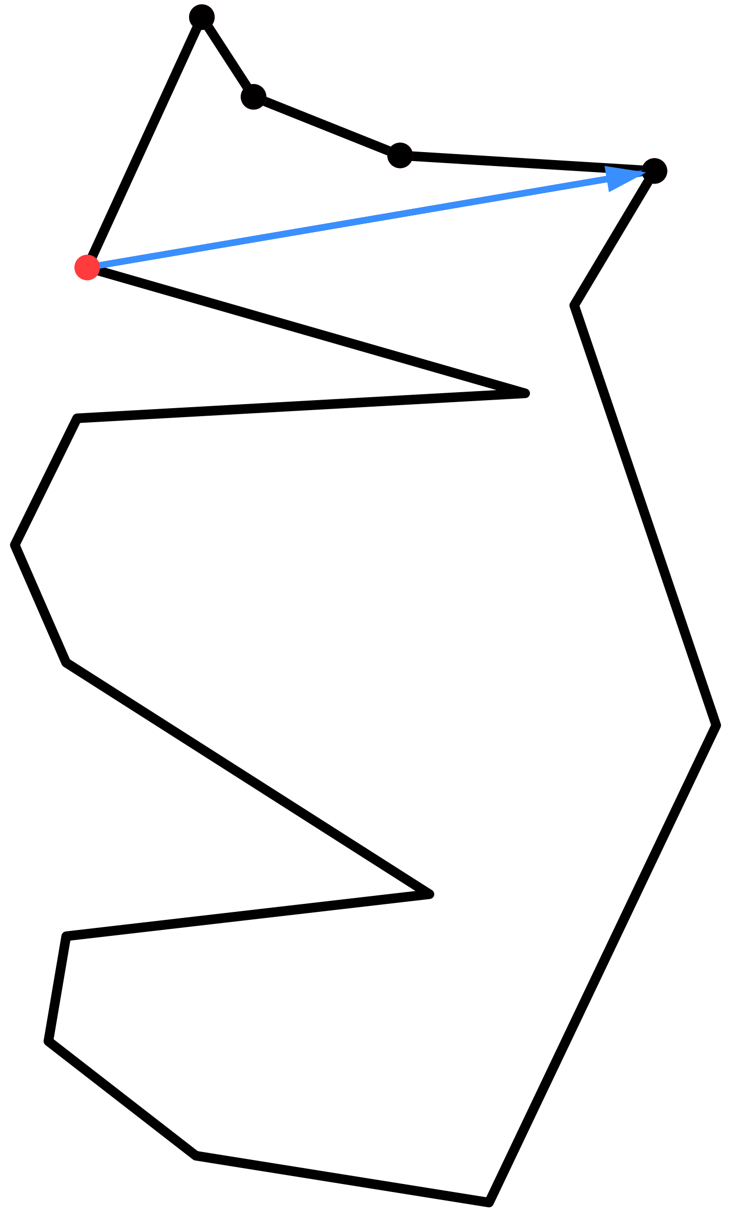
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

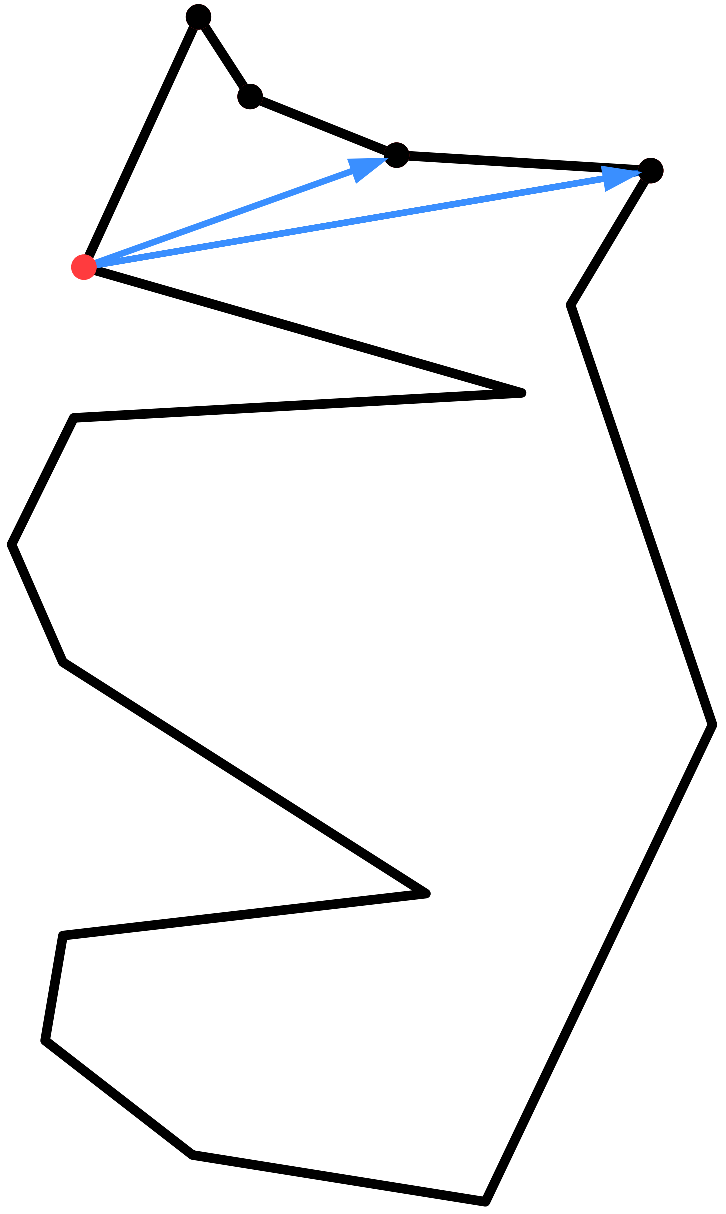
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

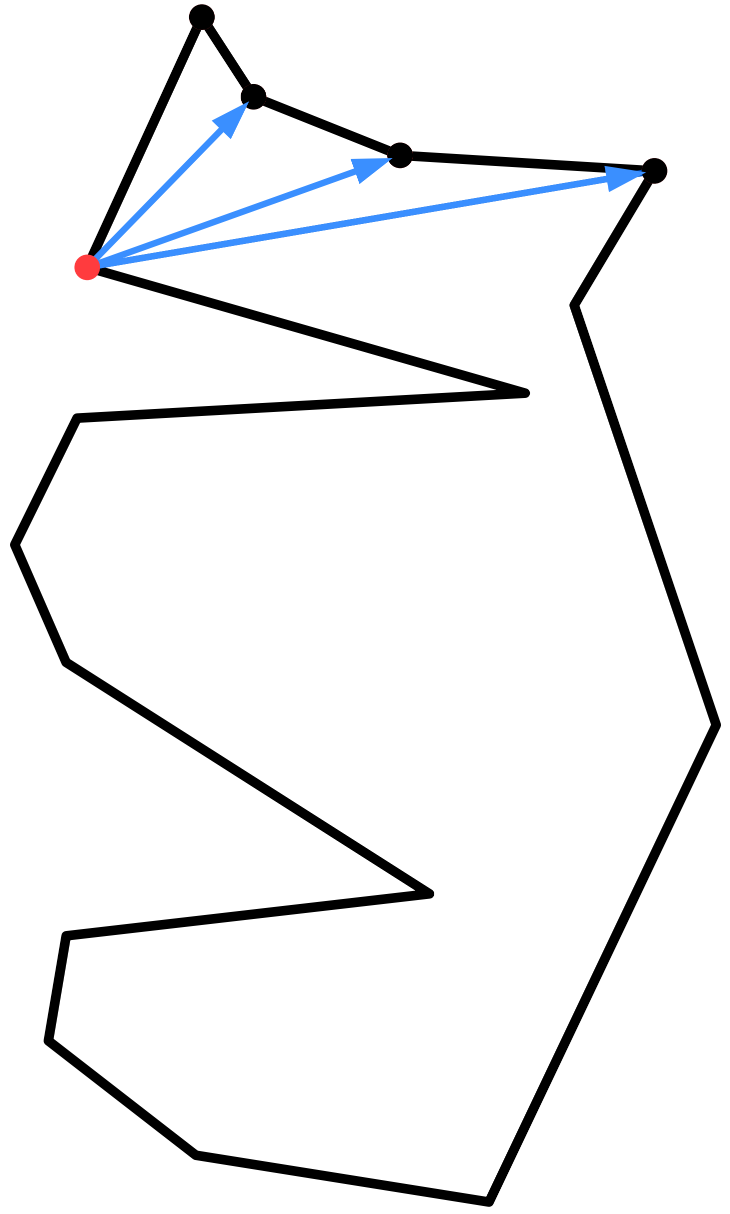
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

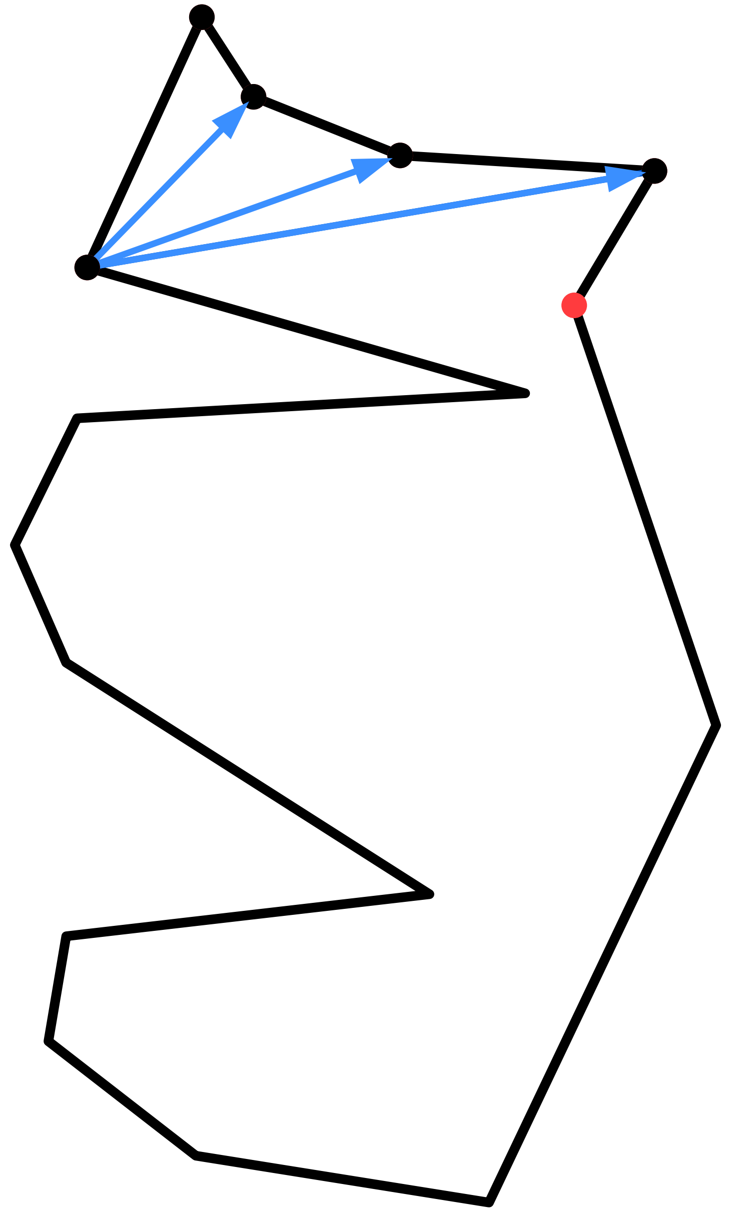
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

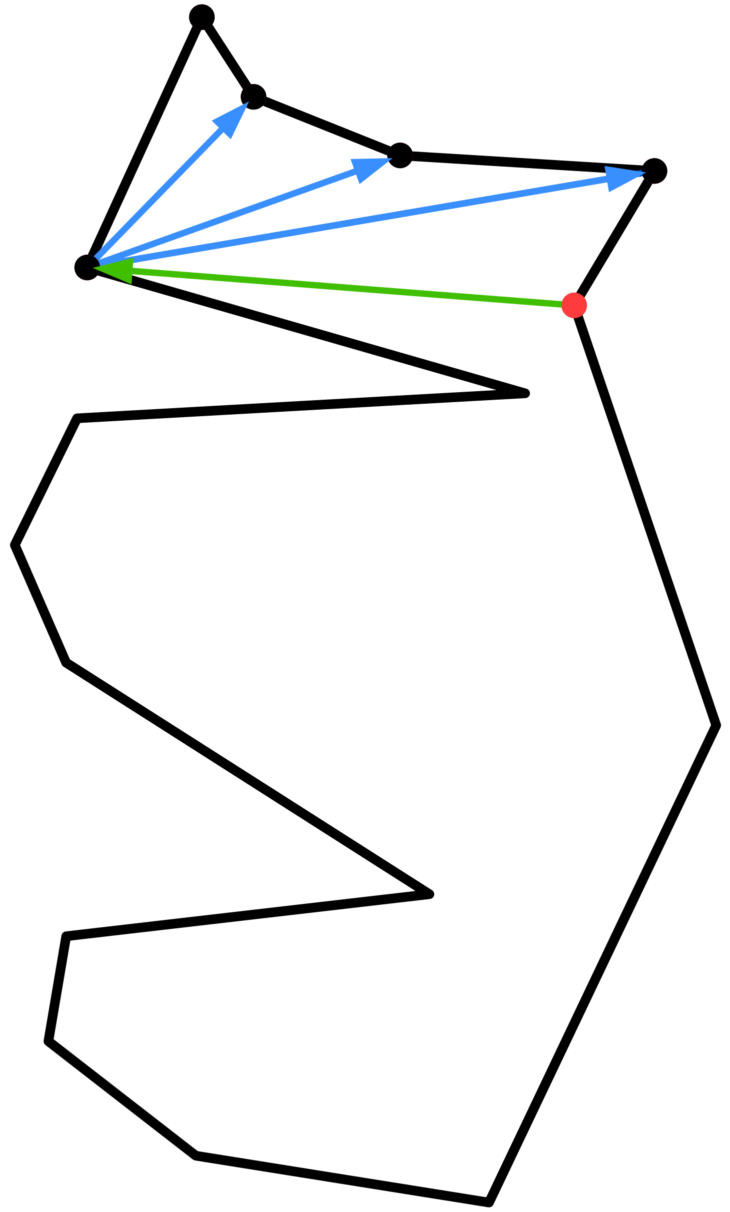
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

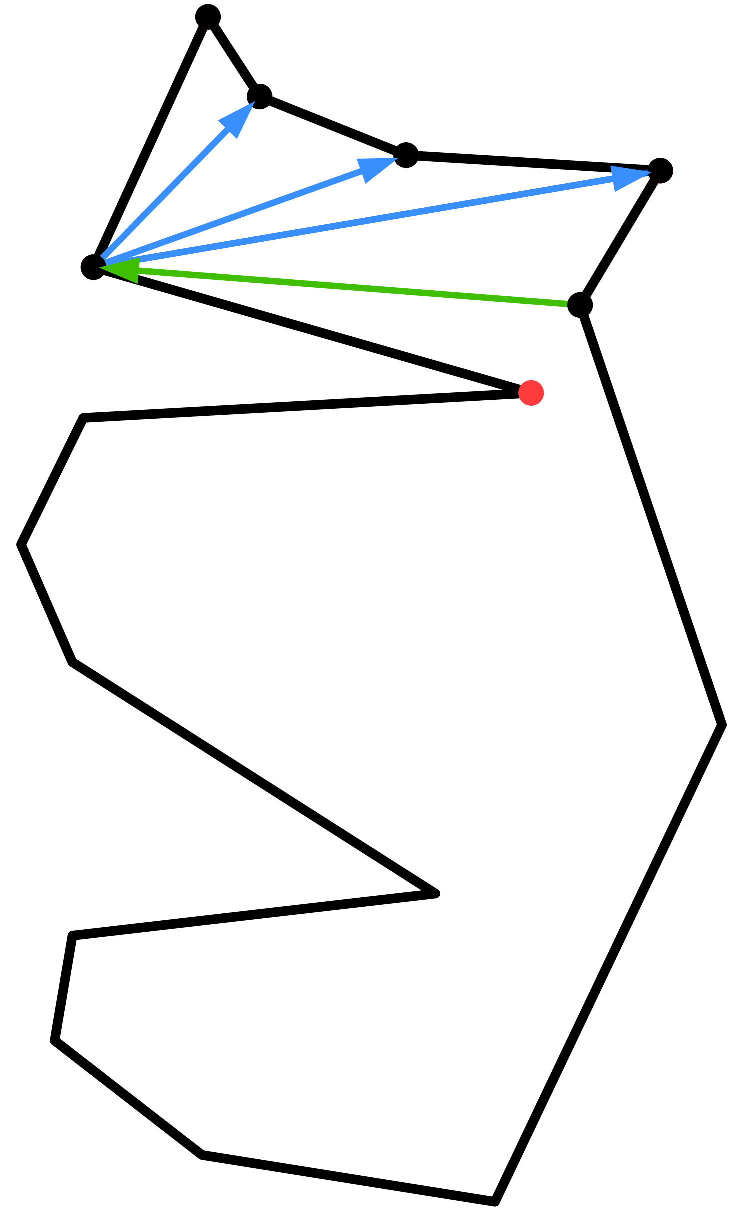
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

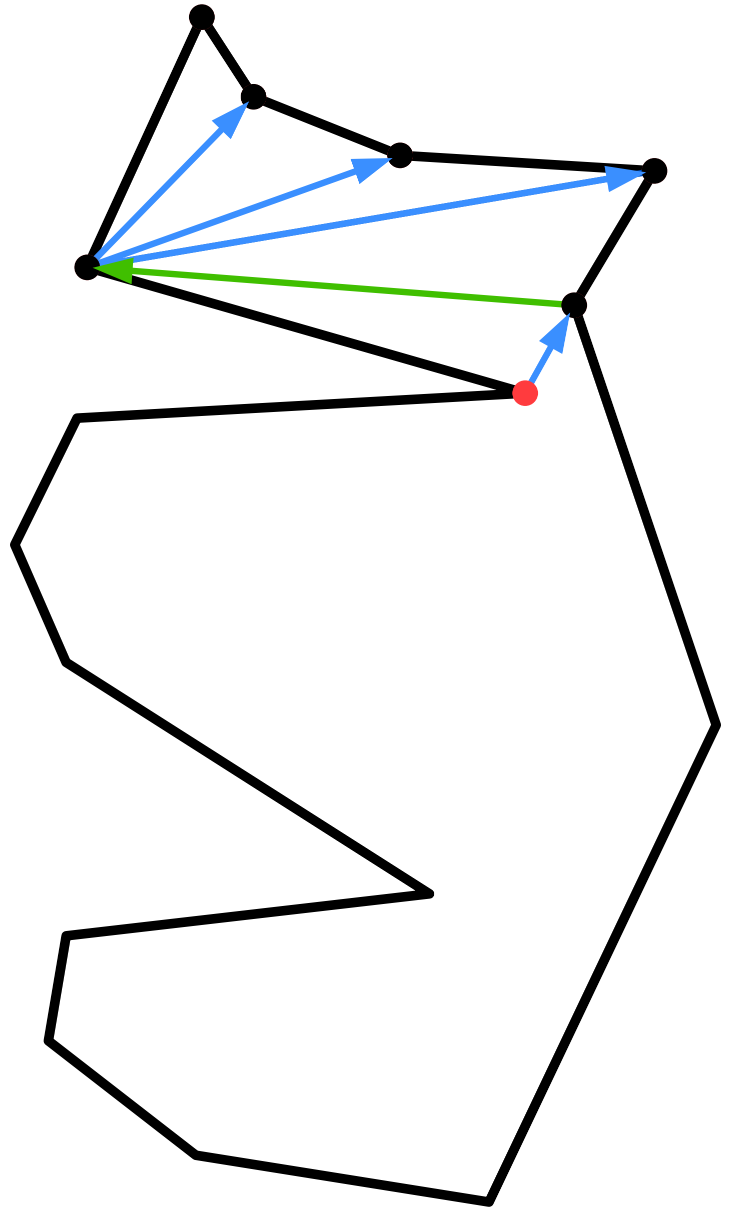
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

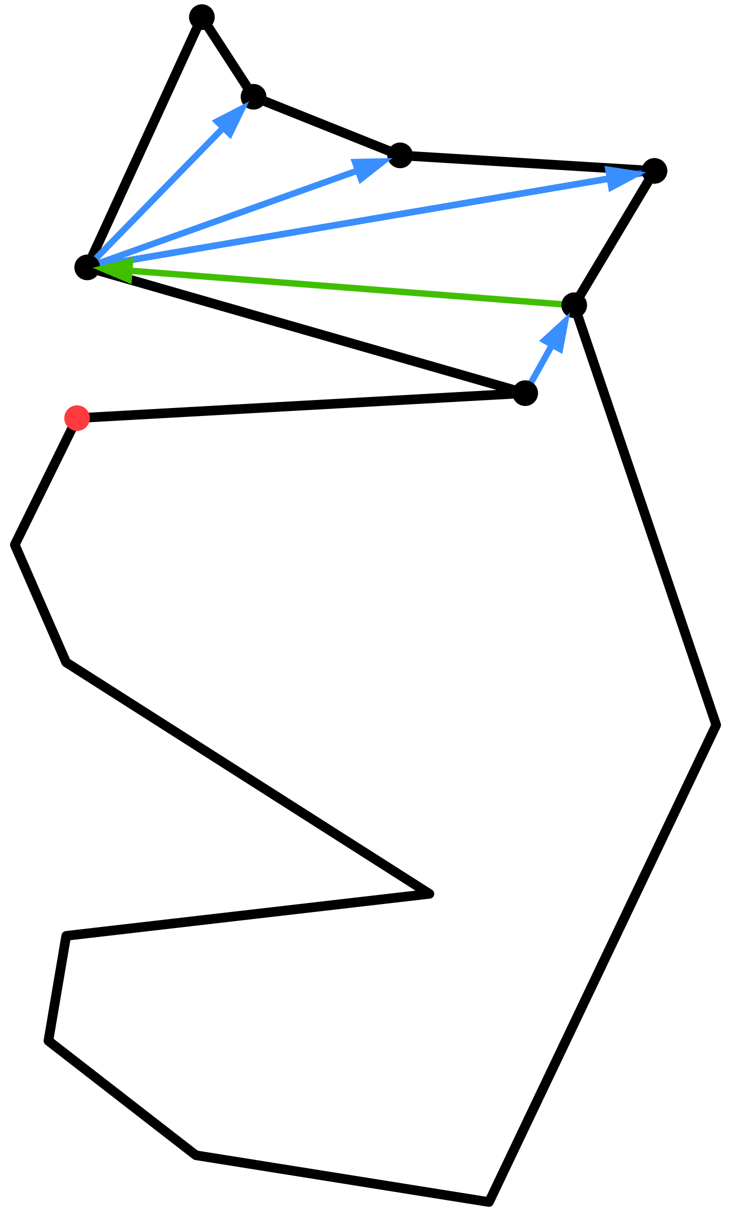
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

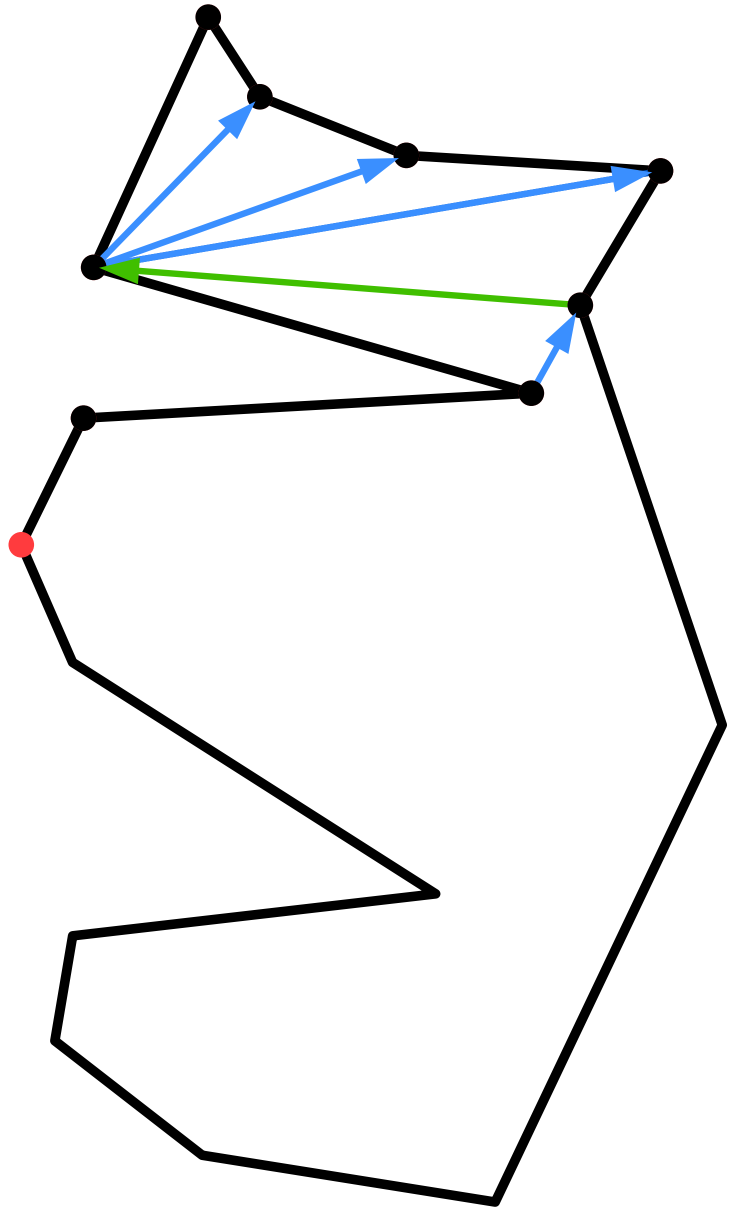
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

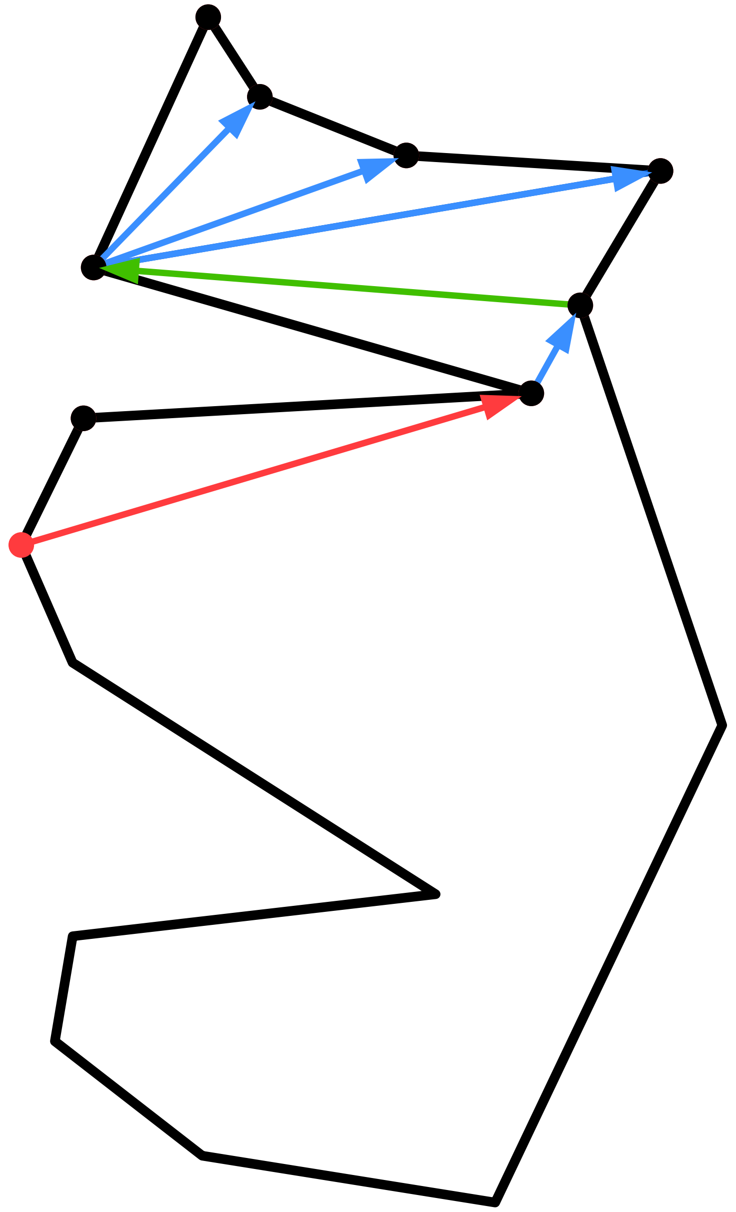
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

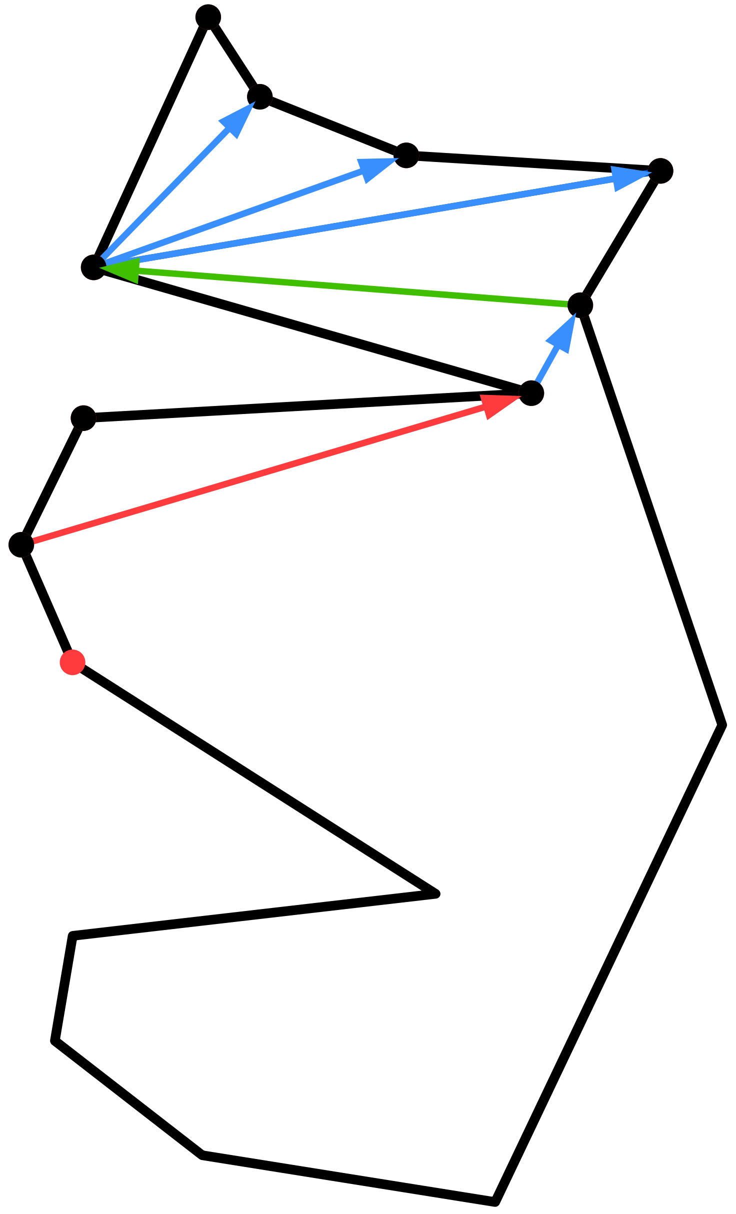
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

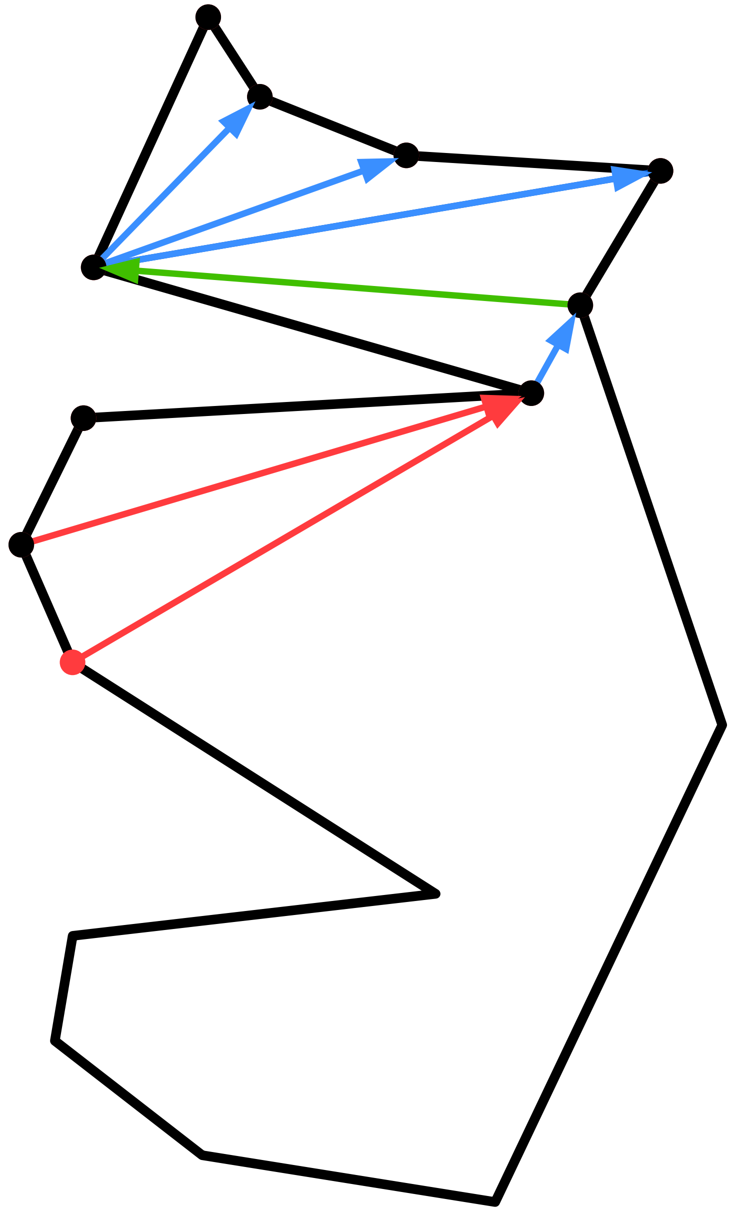
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

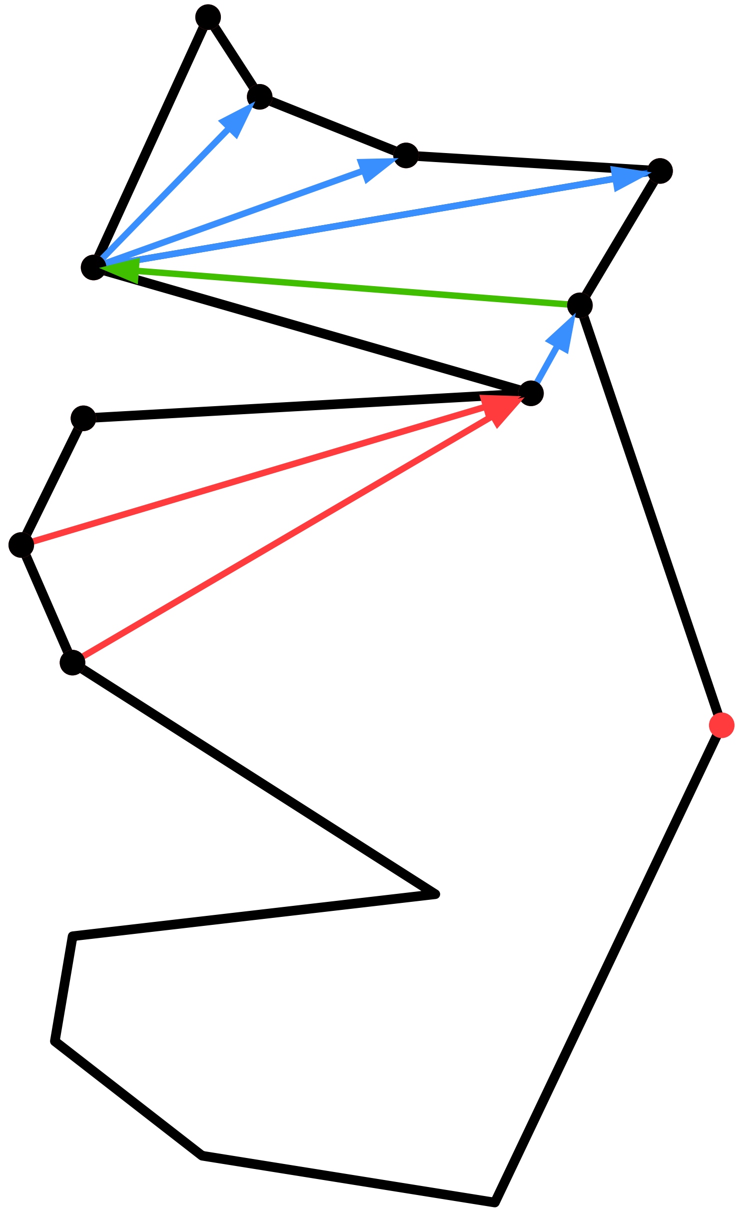
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

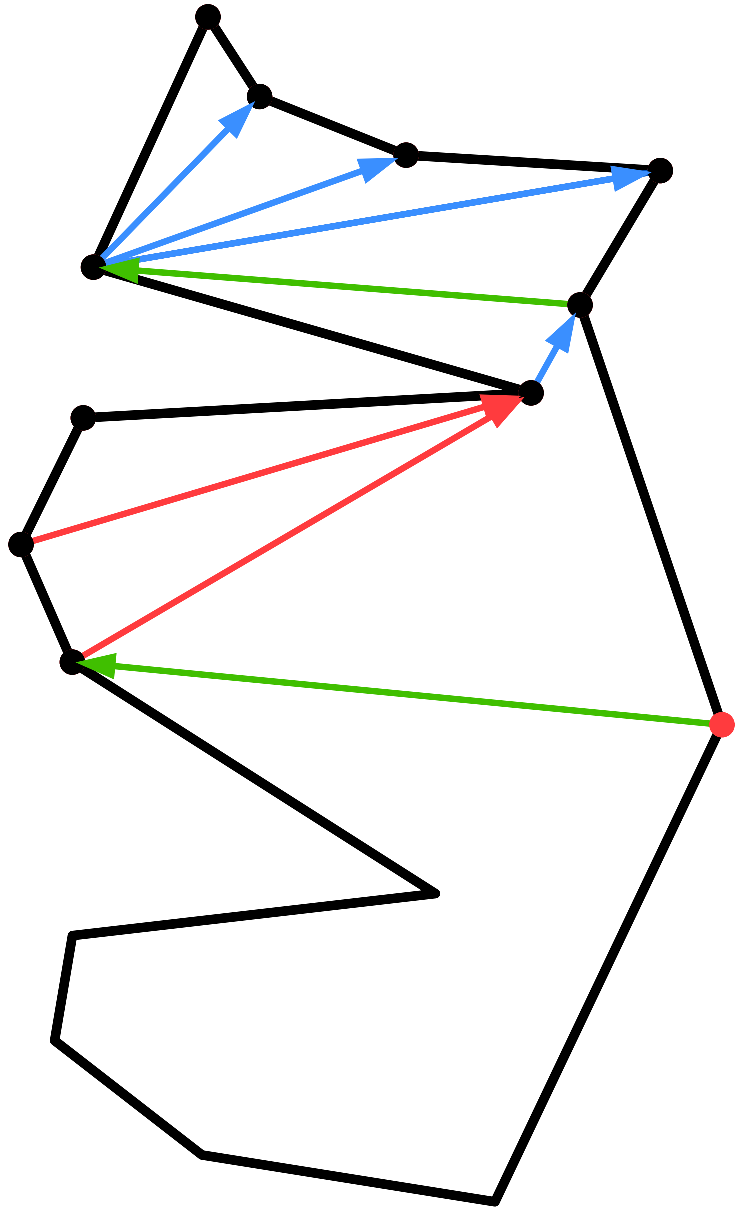
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

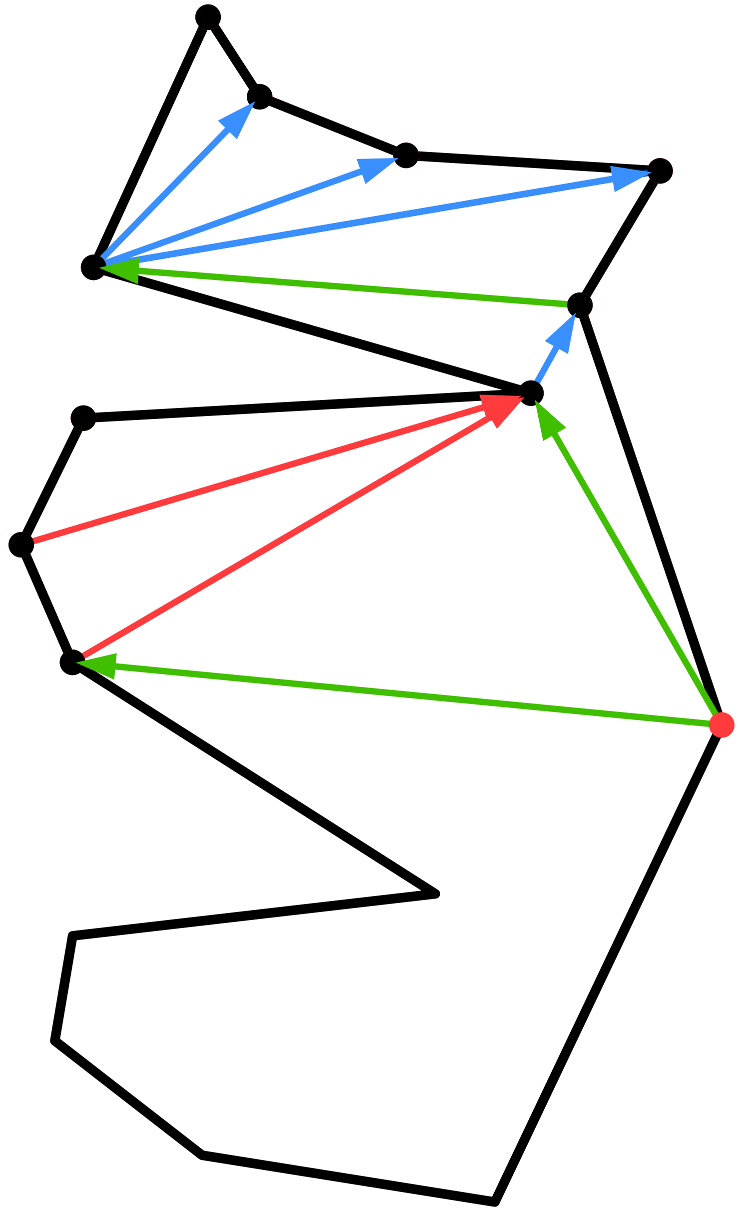
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

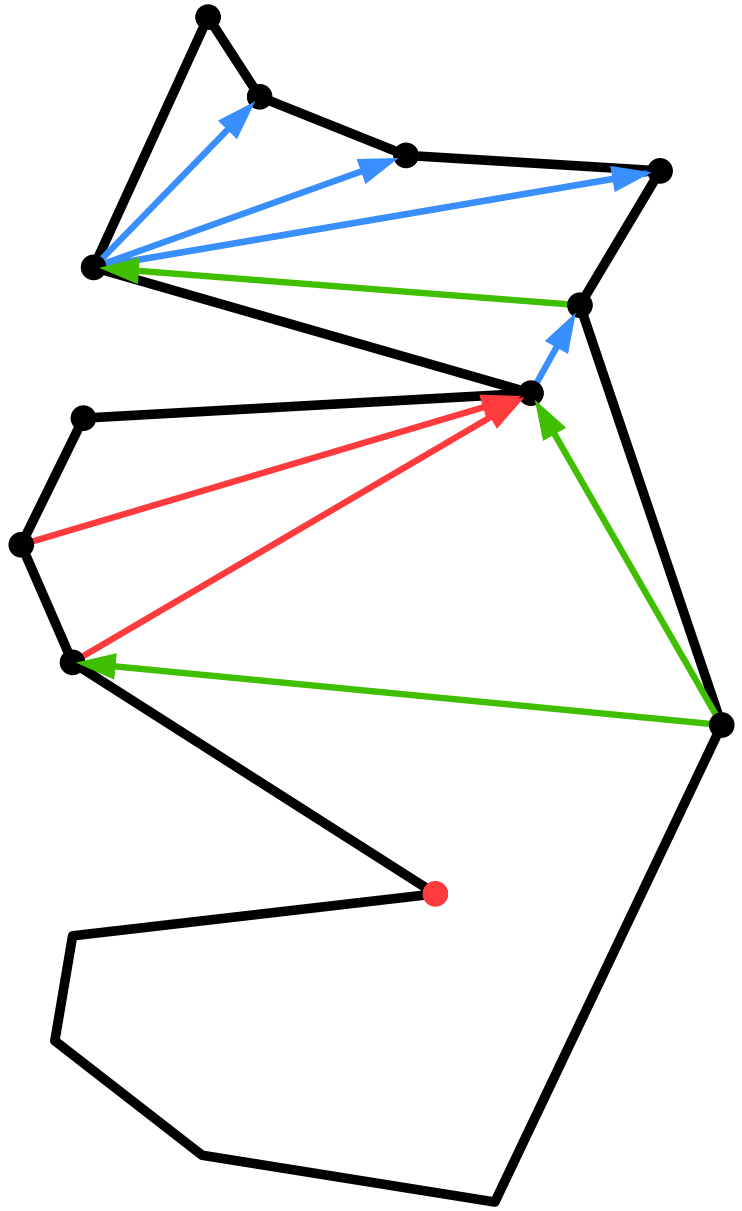
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

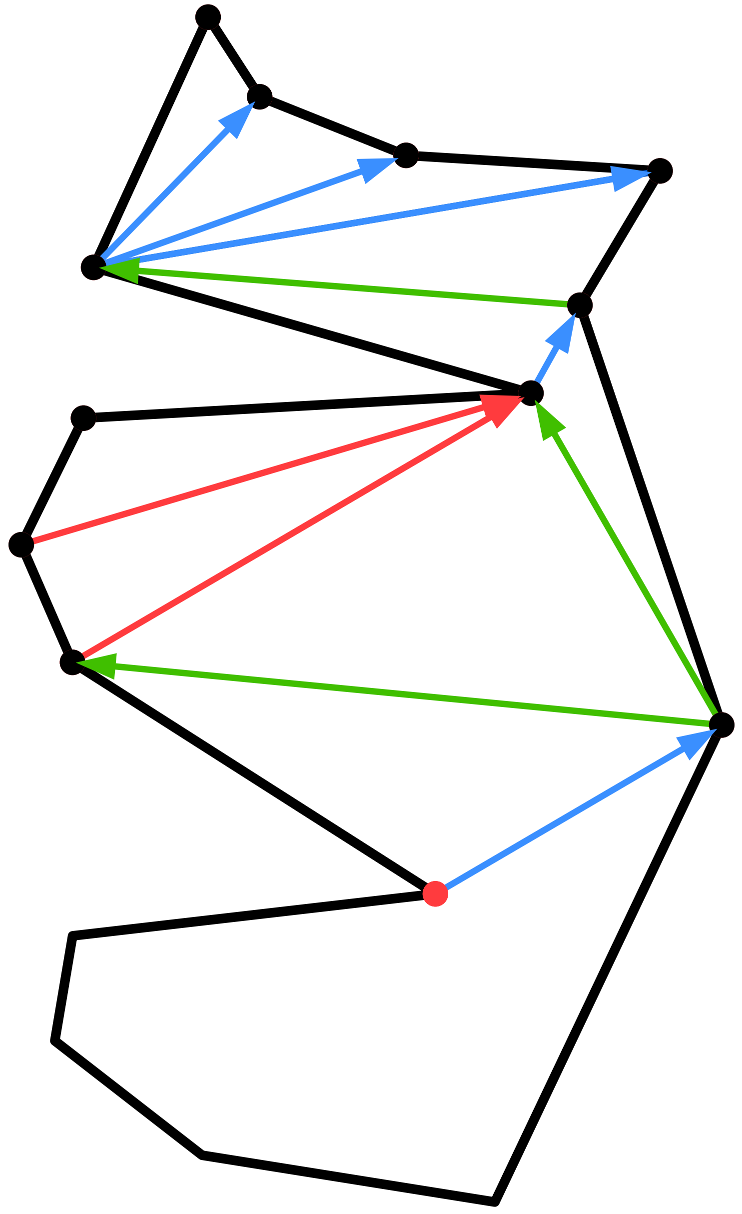
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

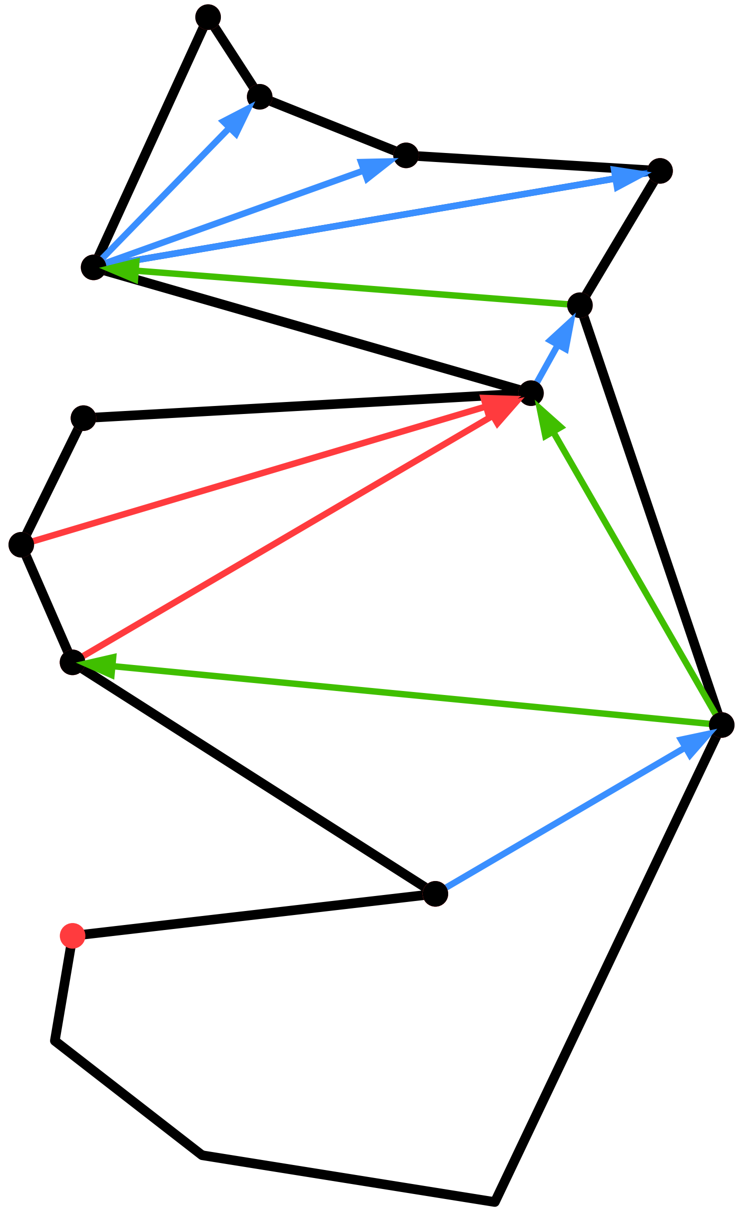
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

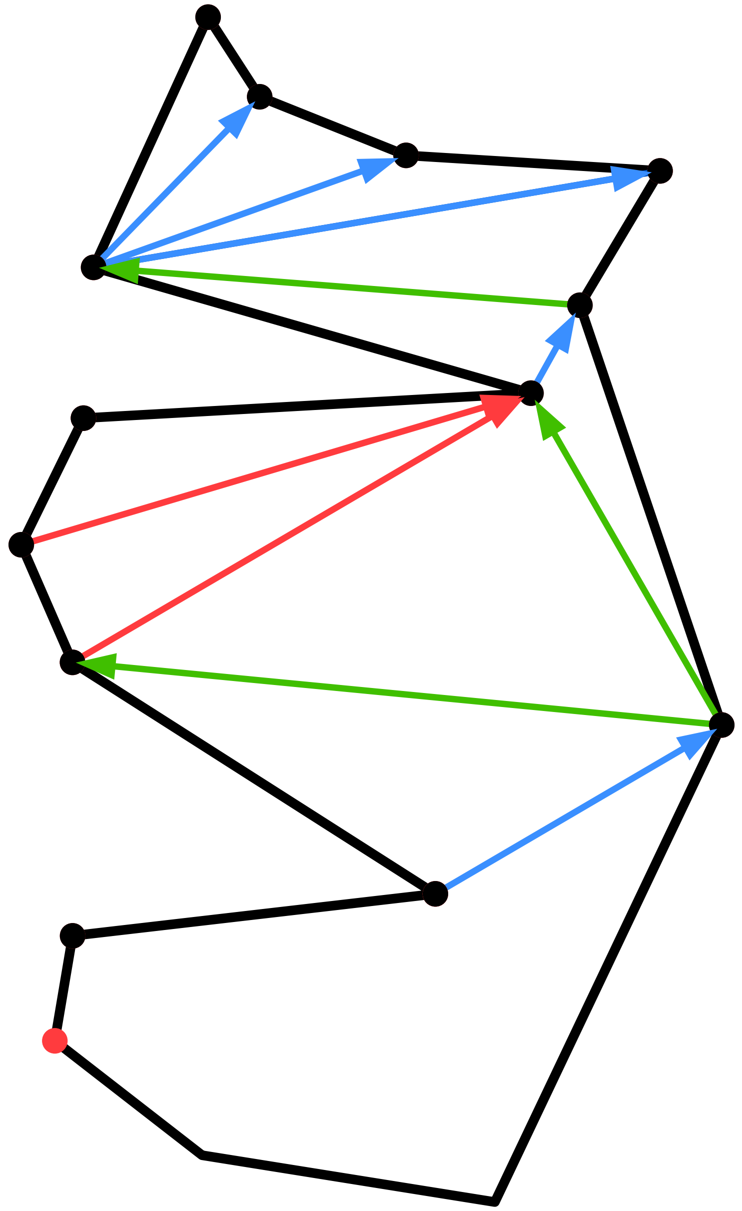
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

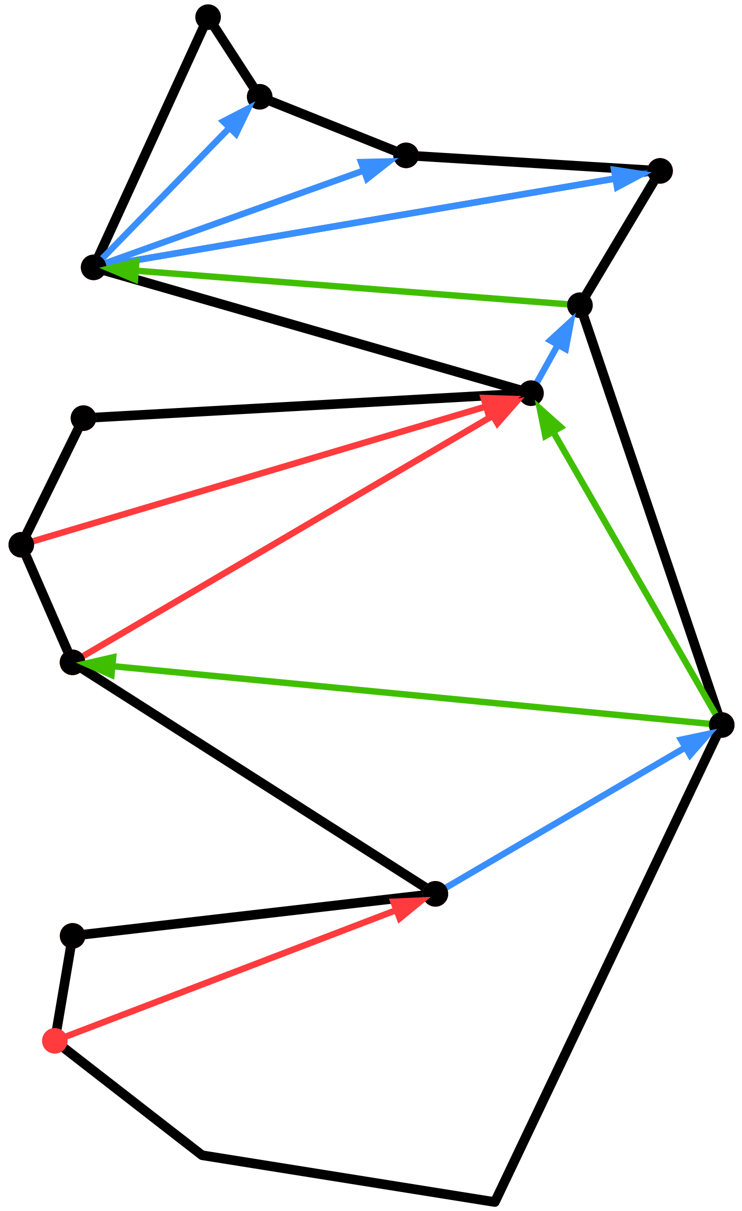
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

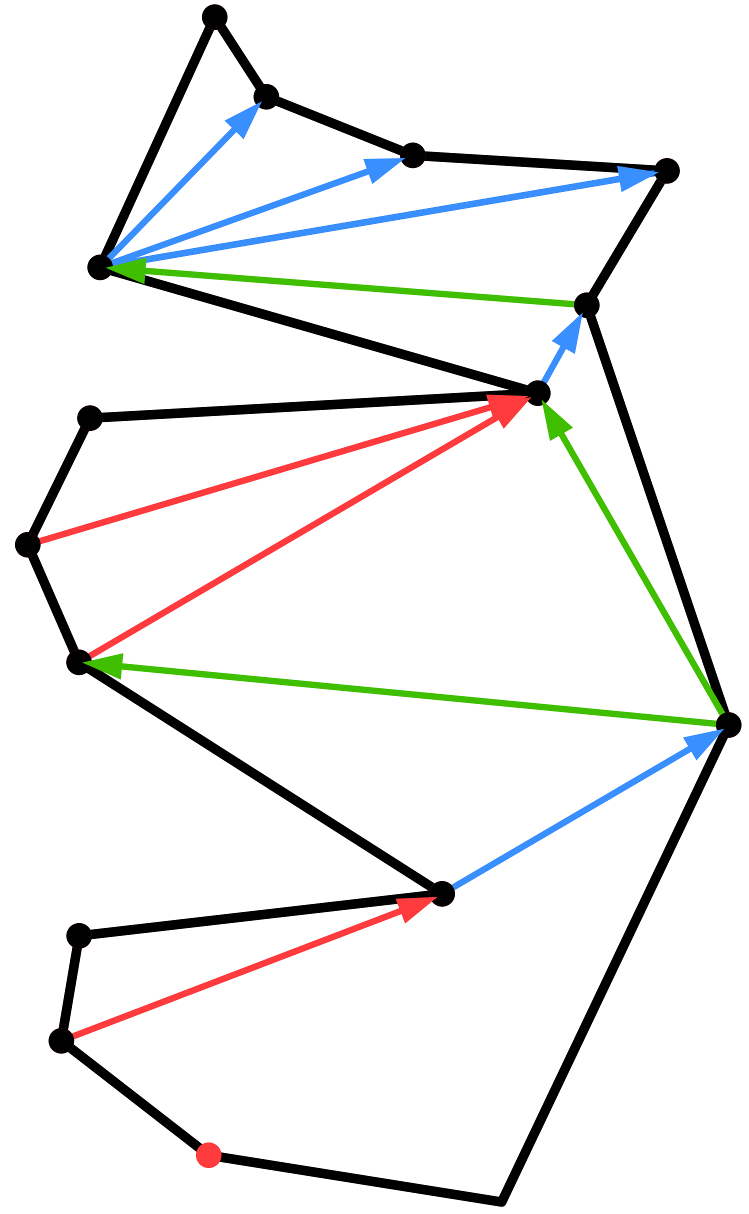
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

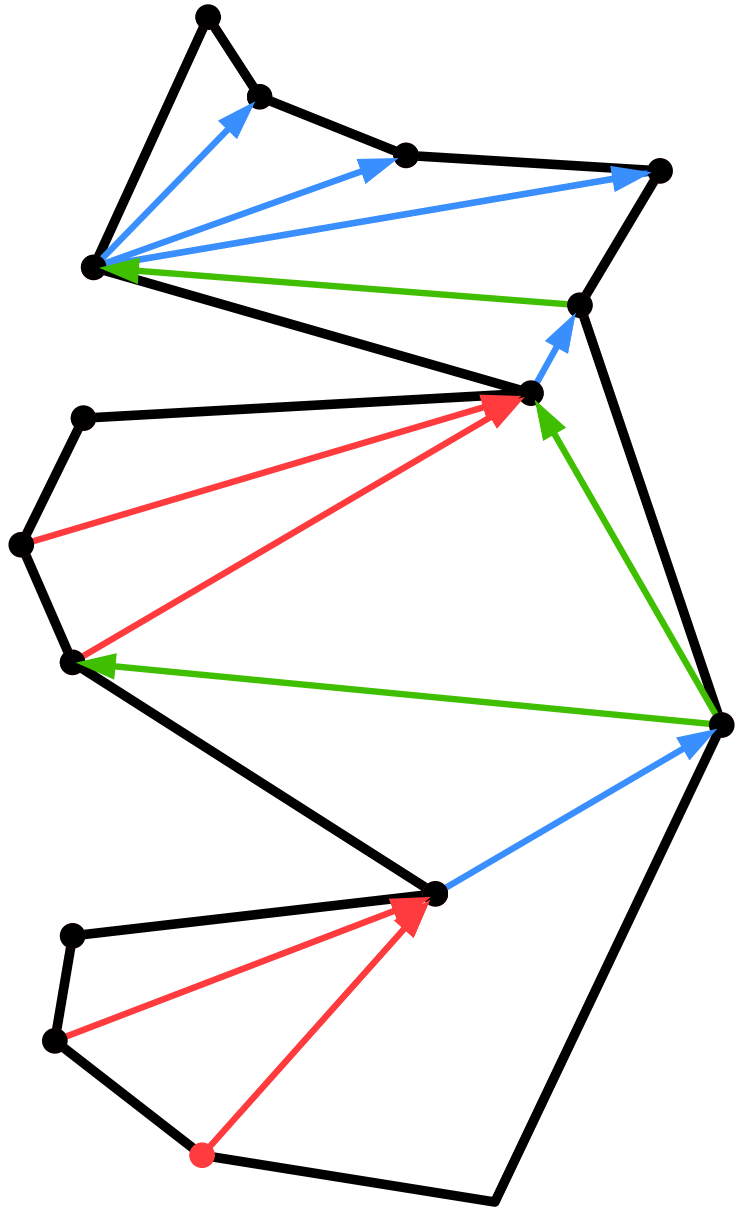
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

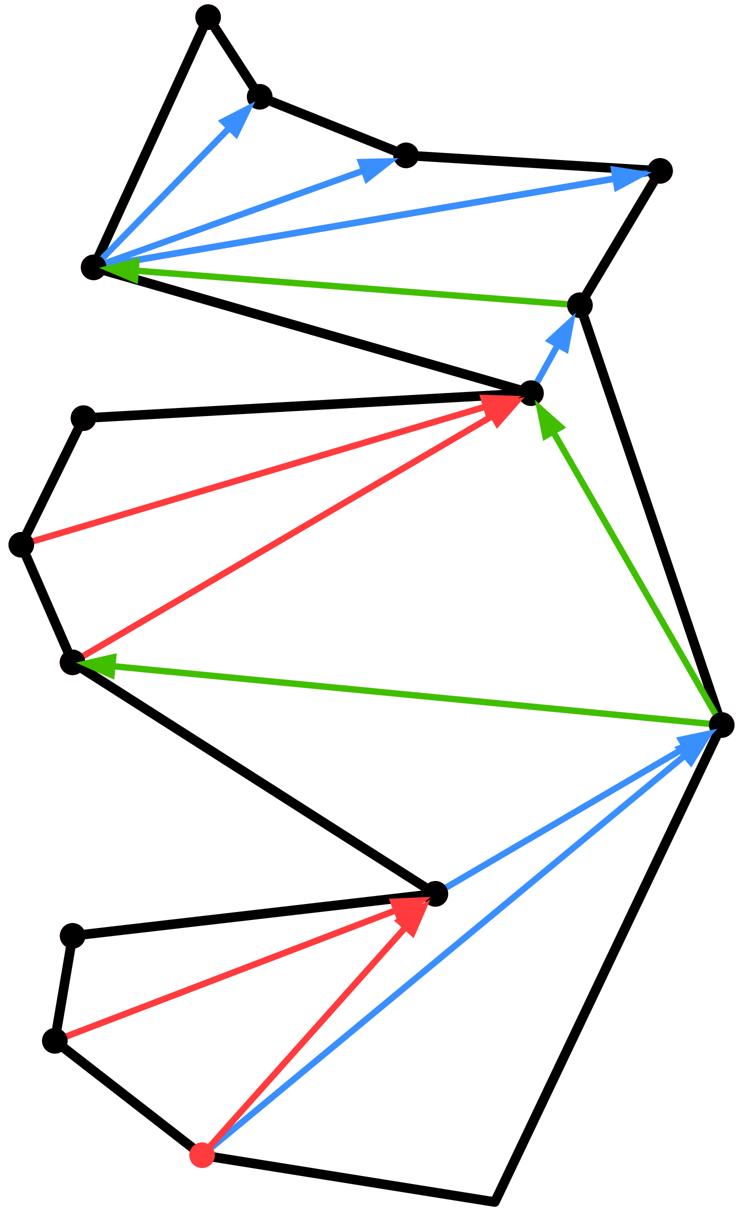
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

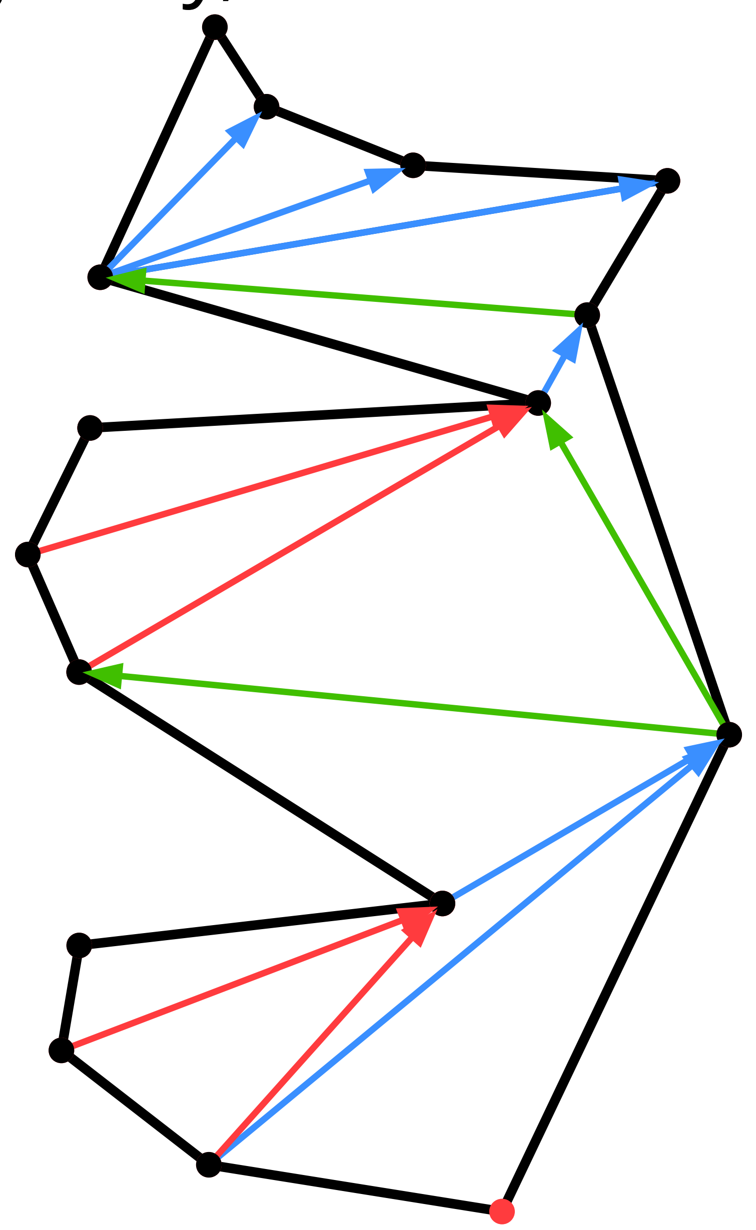
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

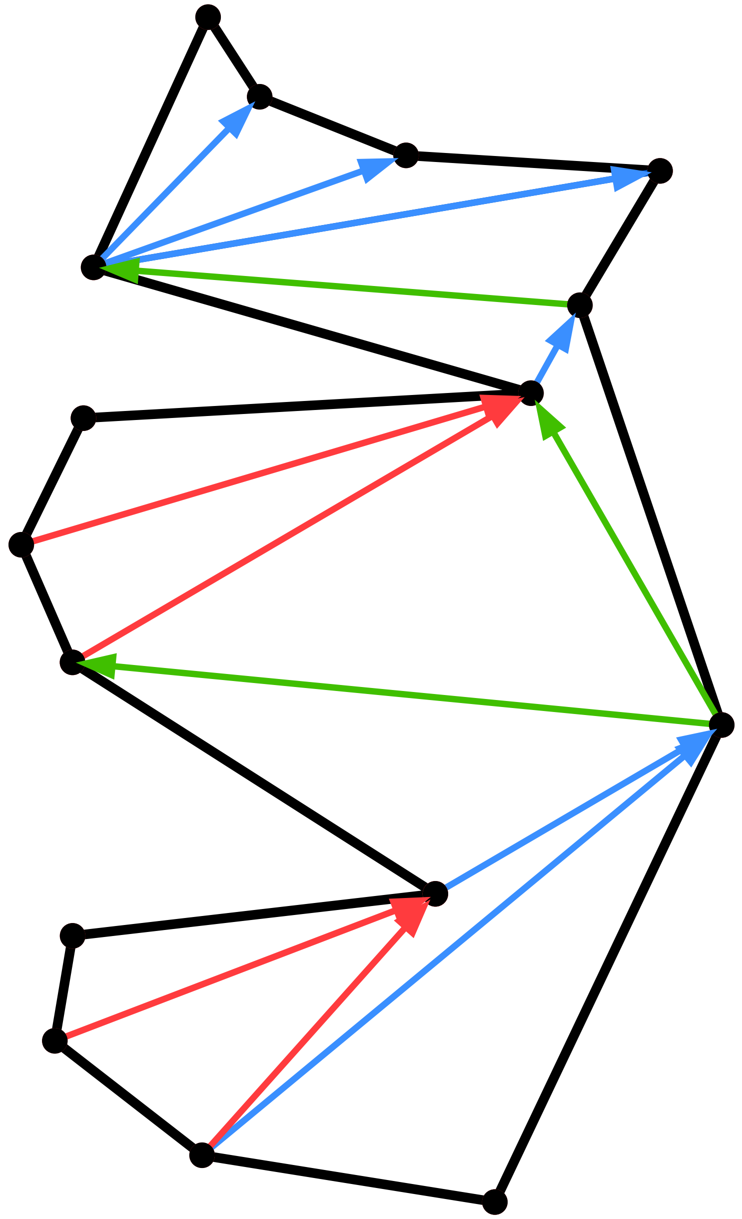
approach: greedy, on both sides top-down



Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

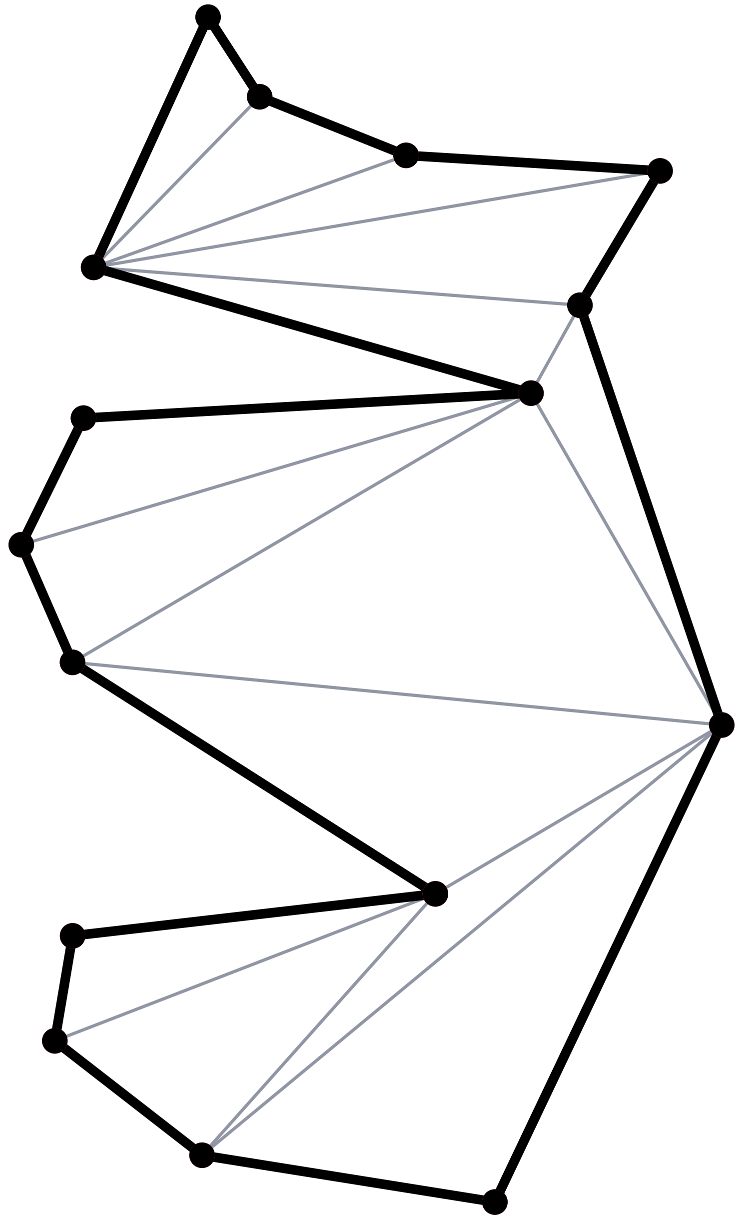


invariant?

Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

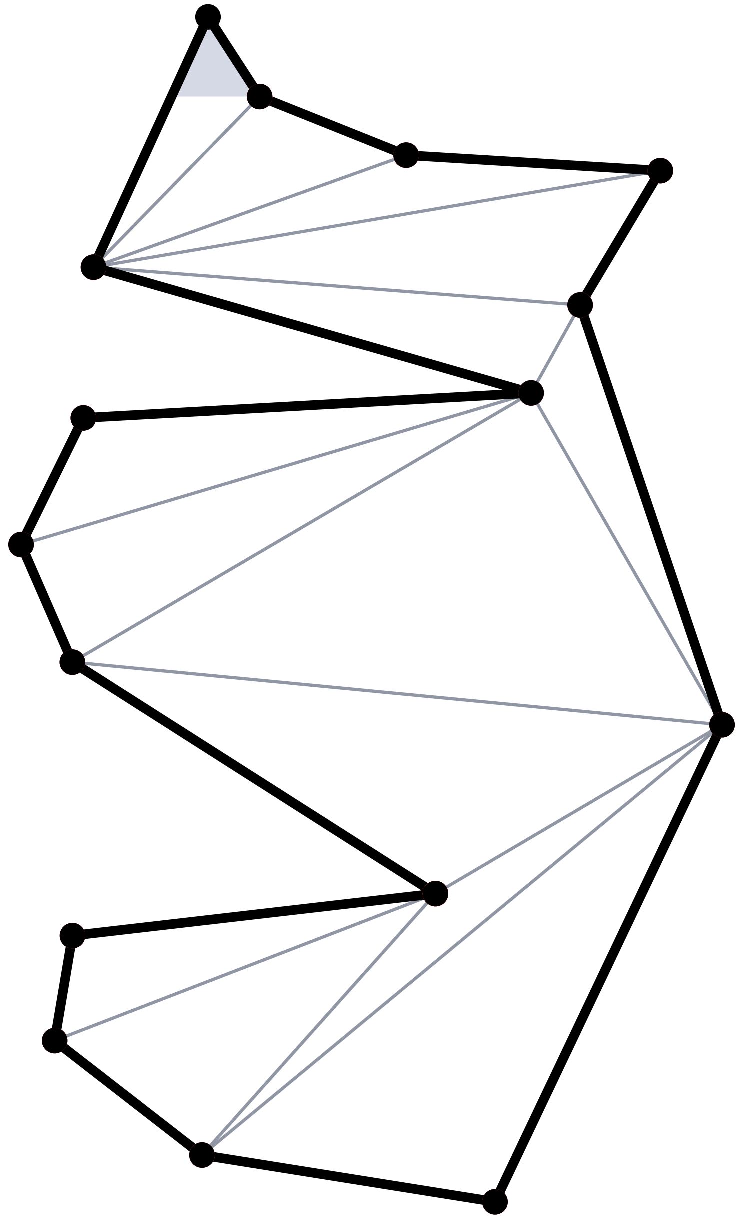


invariant?

Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

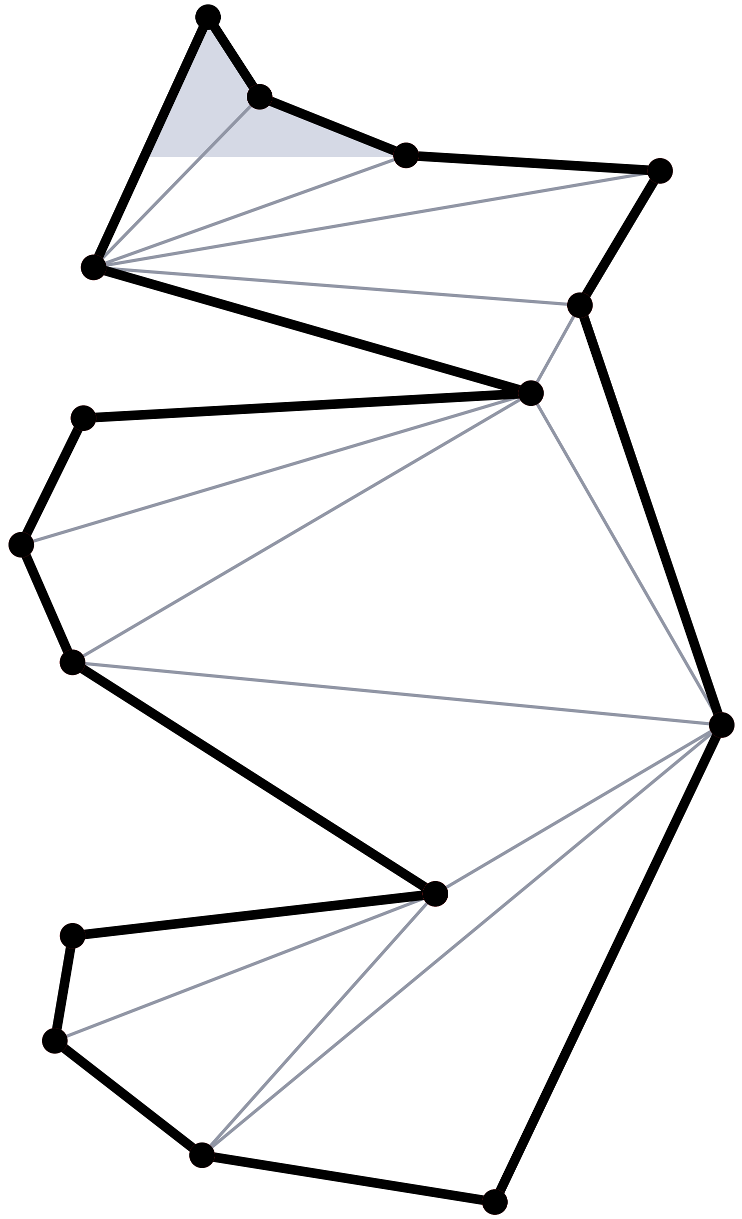


invariant?

Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

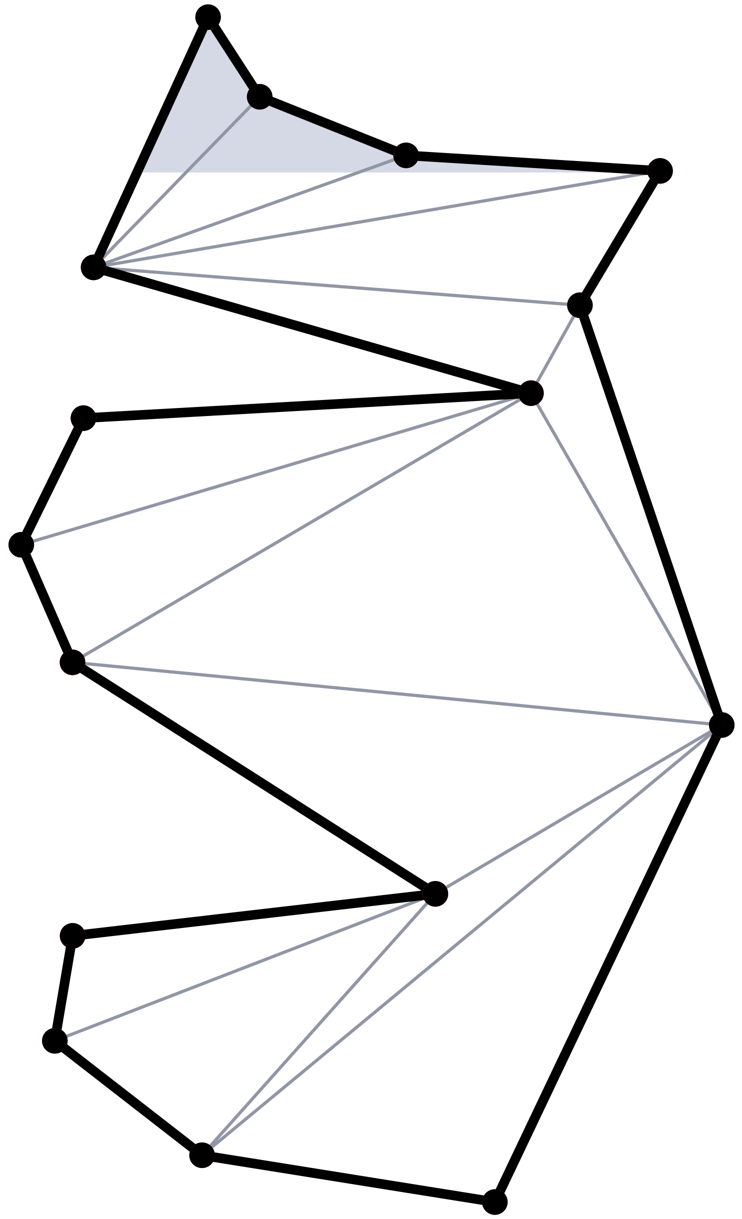


invariant?

Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

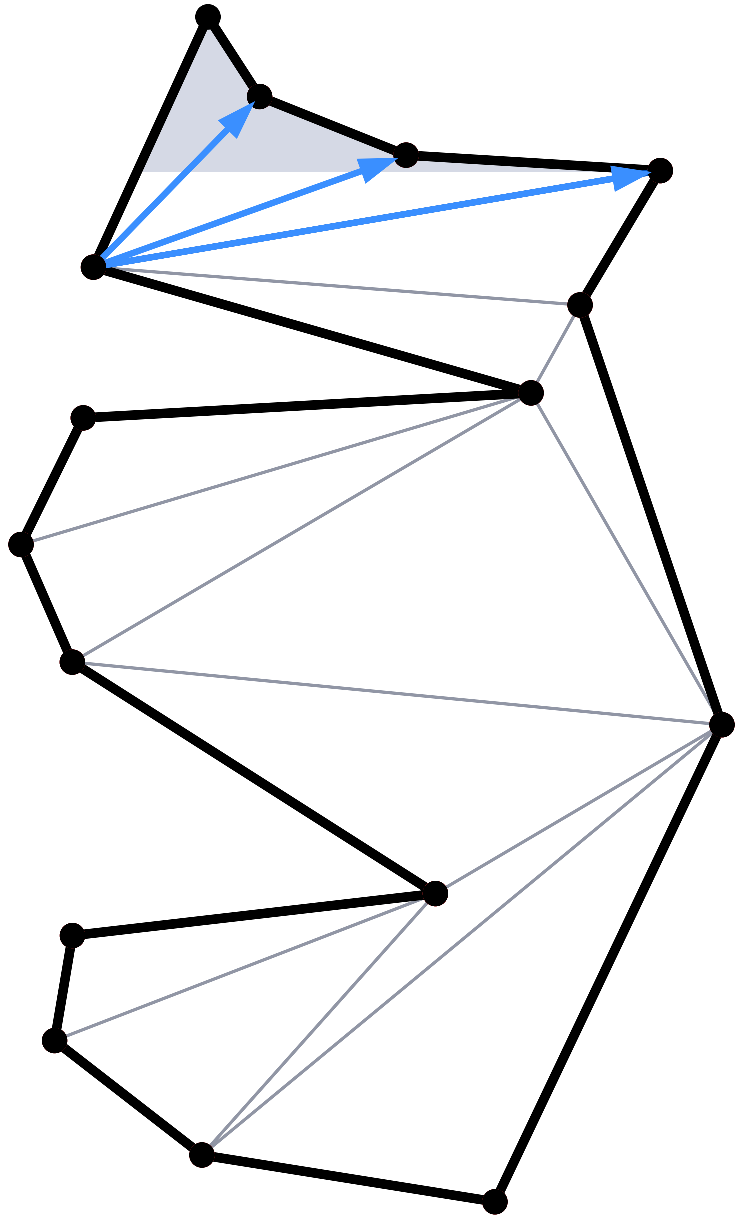


invariant?

Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

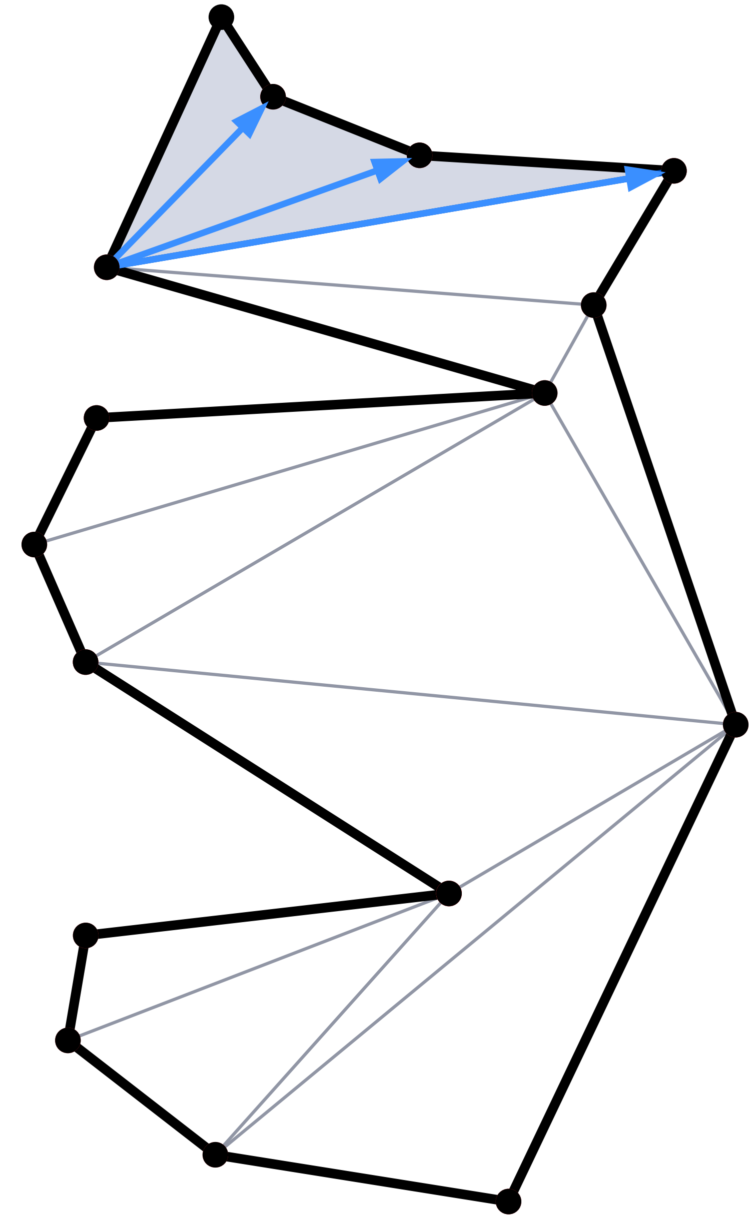


invariant?

Triangulating a y-monotone Polygon

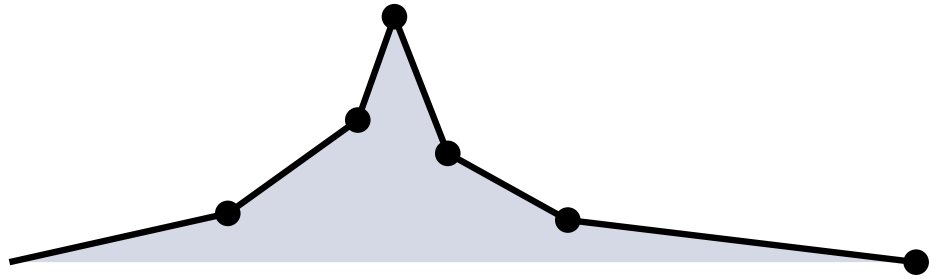
reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down



invariant?

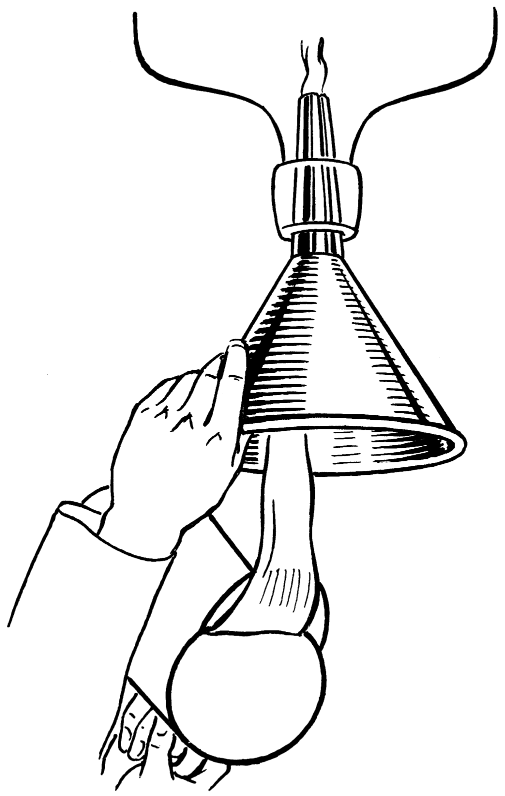
untriangulated part above current vertex is an upside-down funnel

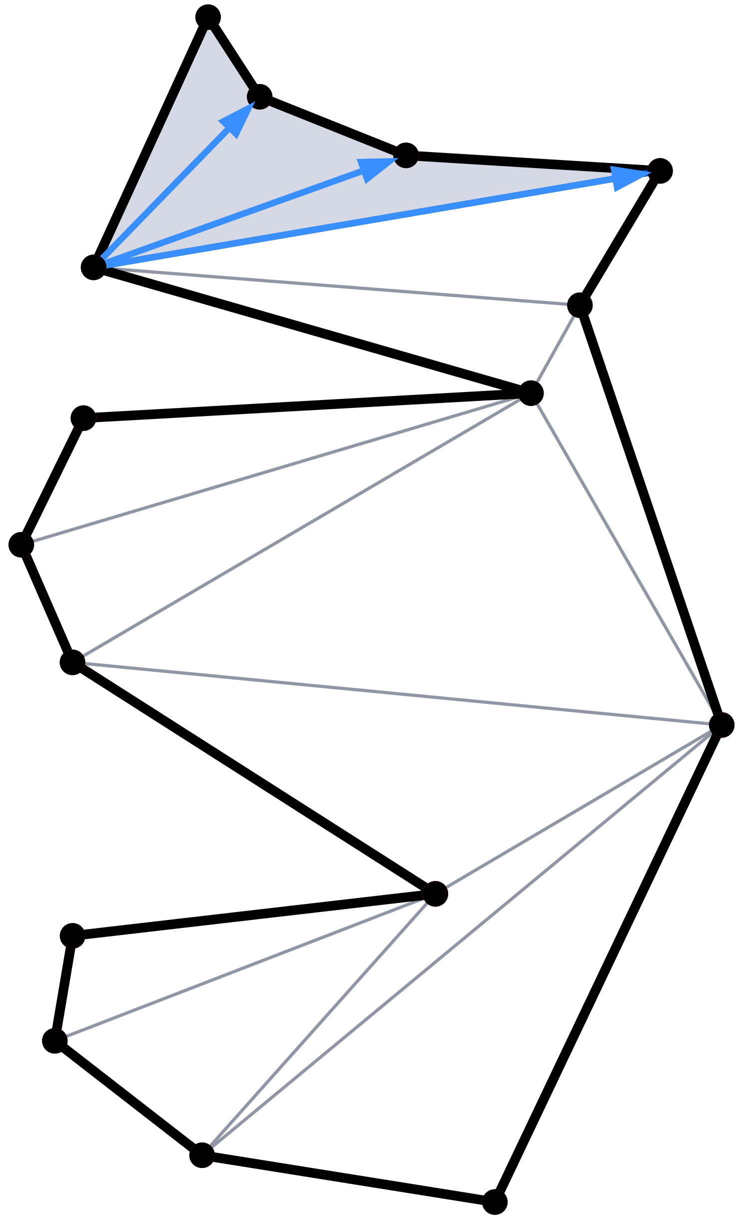


Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

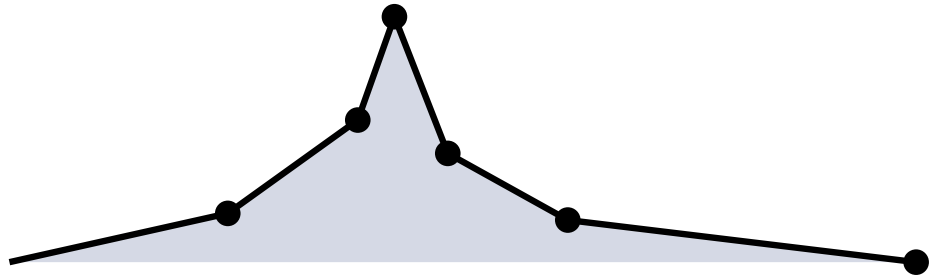
approach: greedy, on both sides top-down





invariant?

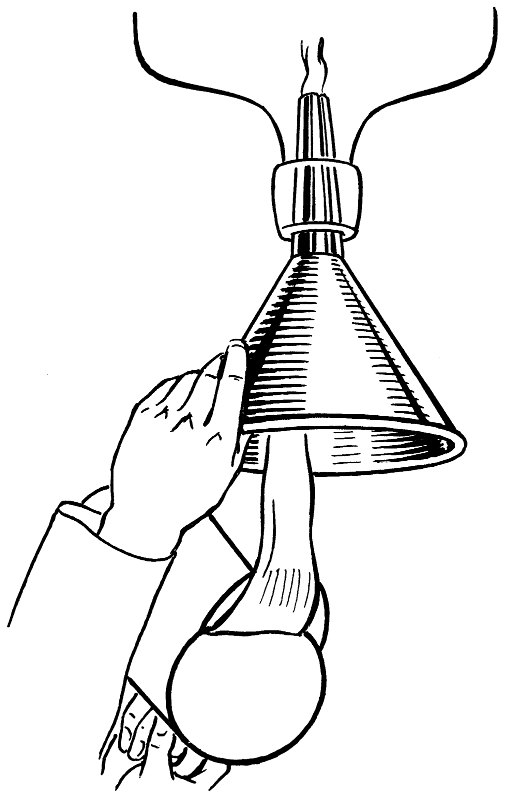
untriangulated part above current vertex is an upside-down funnel

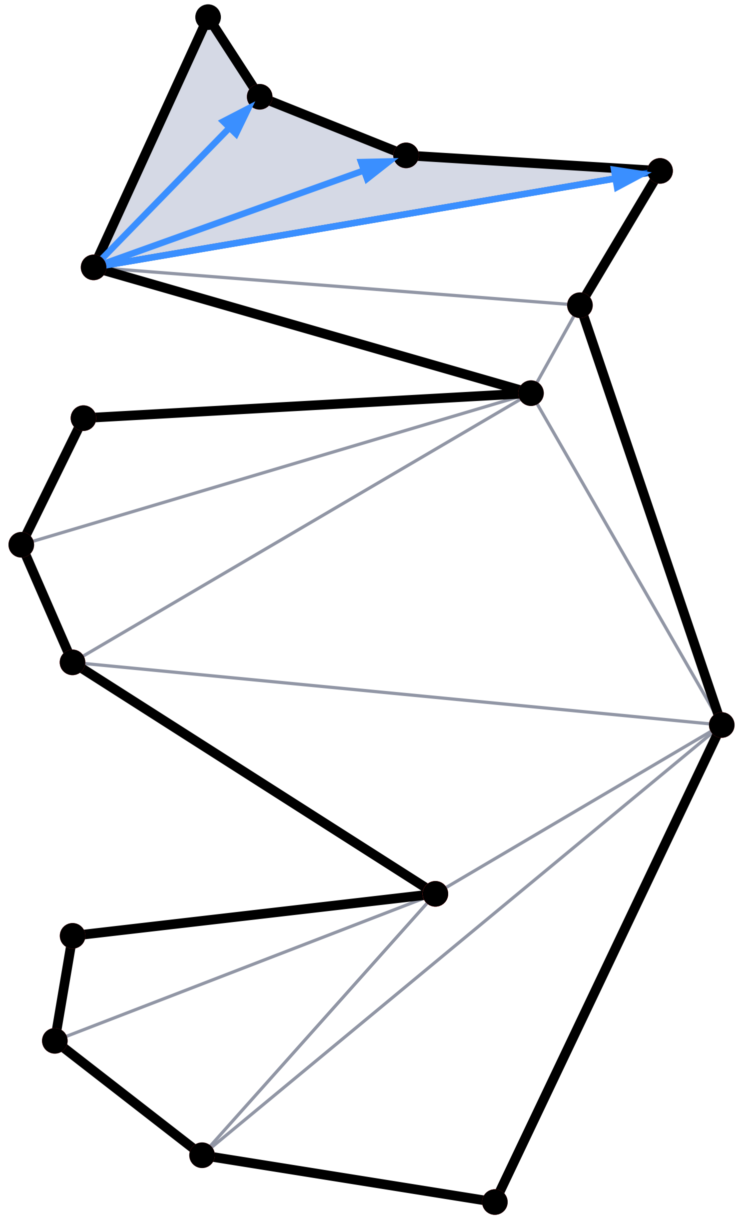


Triangulating a y-monotone Polygon

reminder: boundary chains from top to bottom only go down

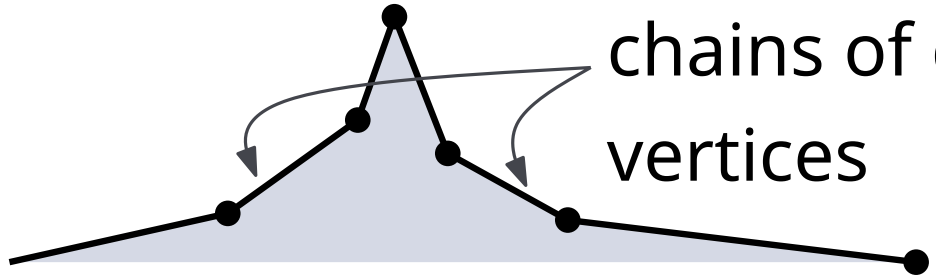
approach: greedy, on both sides top-down





invariant?

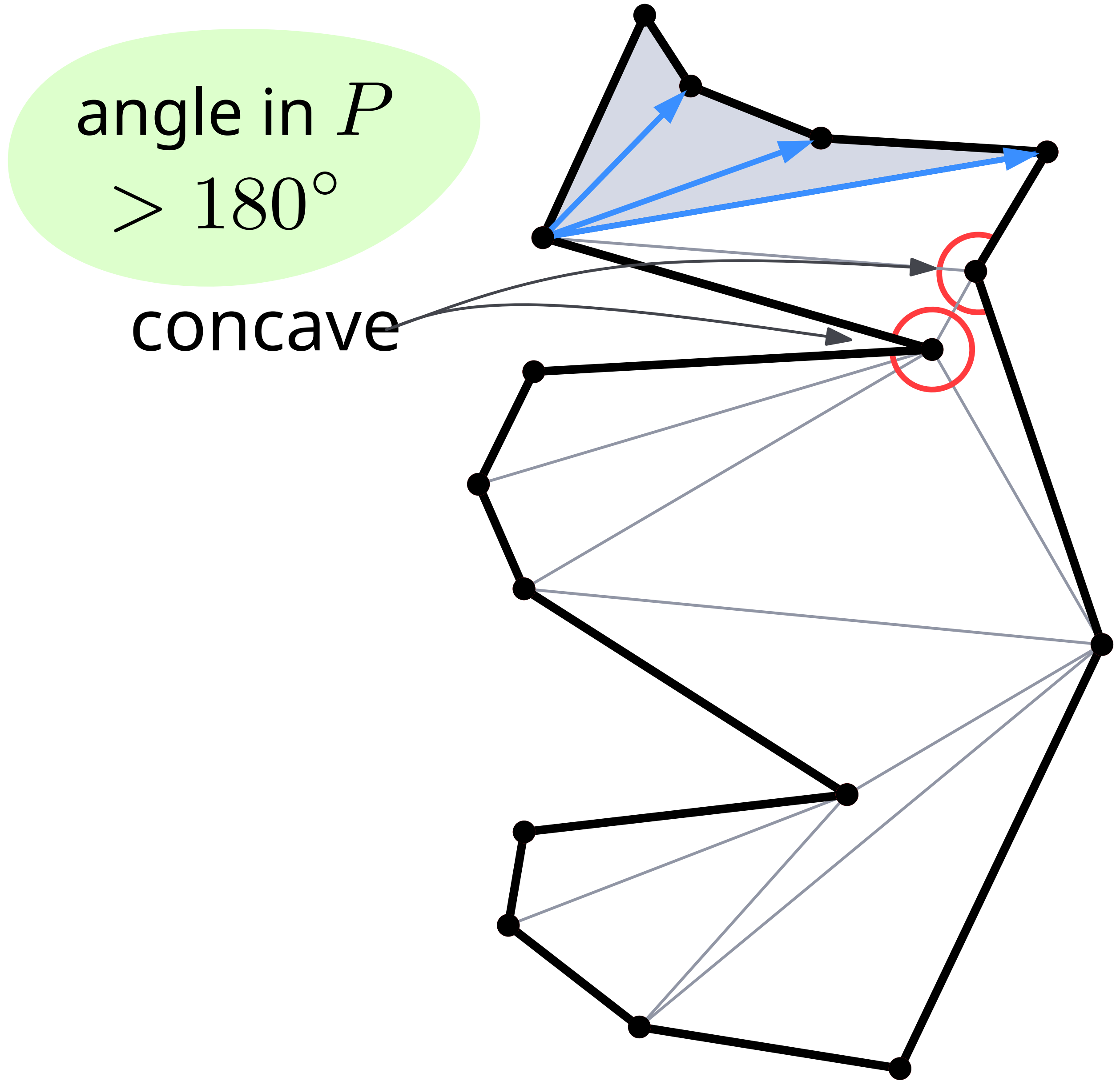
untriangulated part above current vertex is an upside-down funnel



Triangulating a y-monotone Polygon

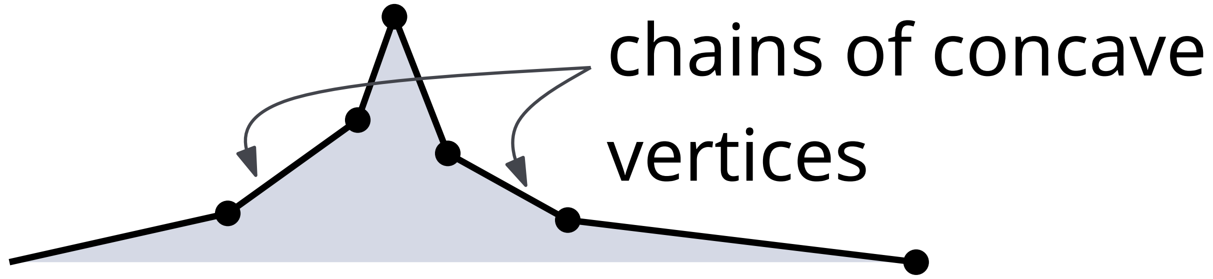
reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down



invariant?

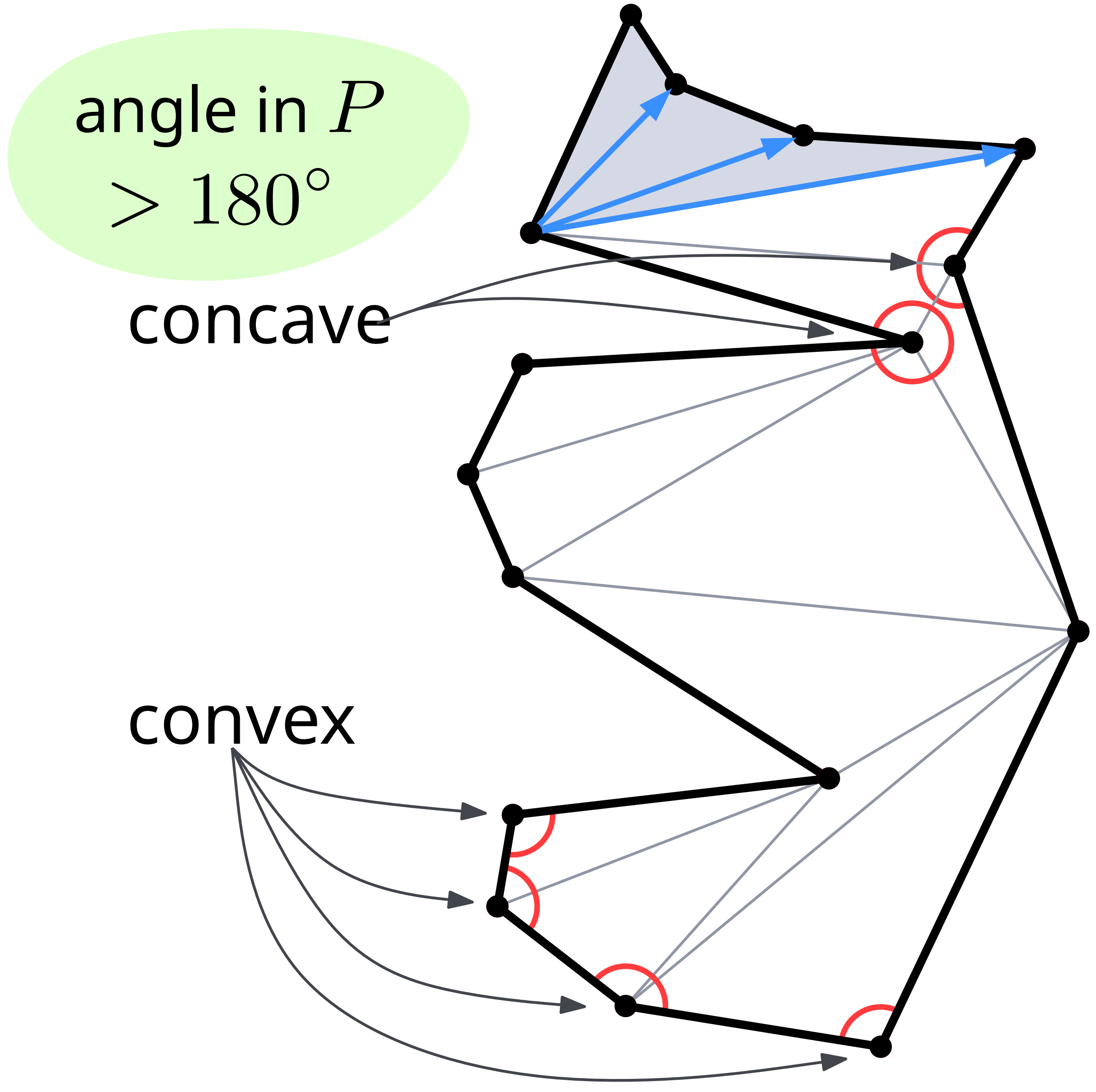
untriangulated part above current vertex is an upside-down funnel



Triangulating a y-monotone Polygon

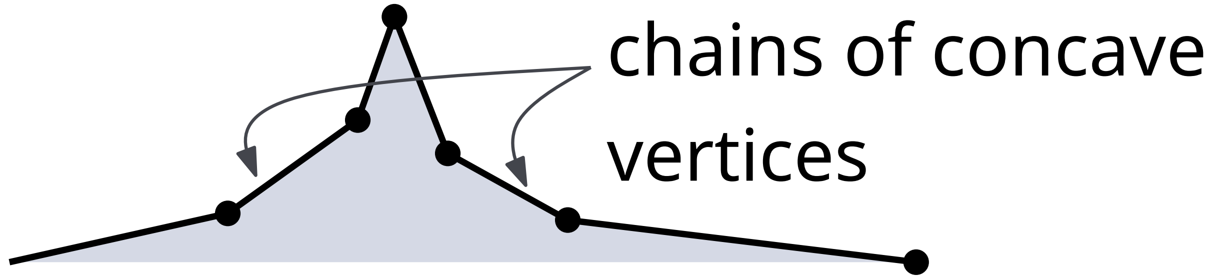
reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down



invariant?

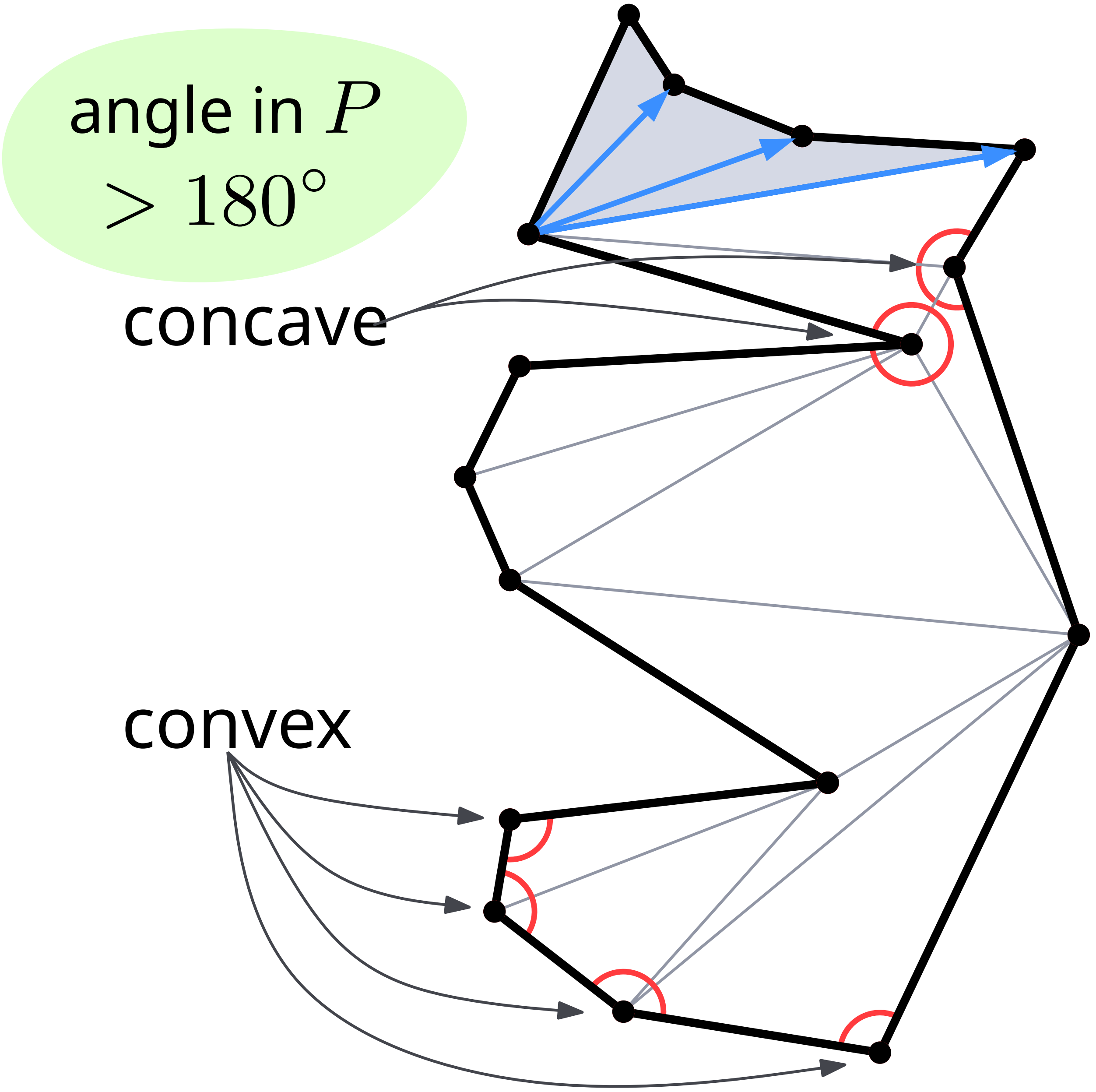
untriangulated part above current vertex is an upside-down funnel



Triangulating a y-monotone Polygon

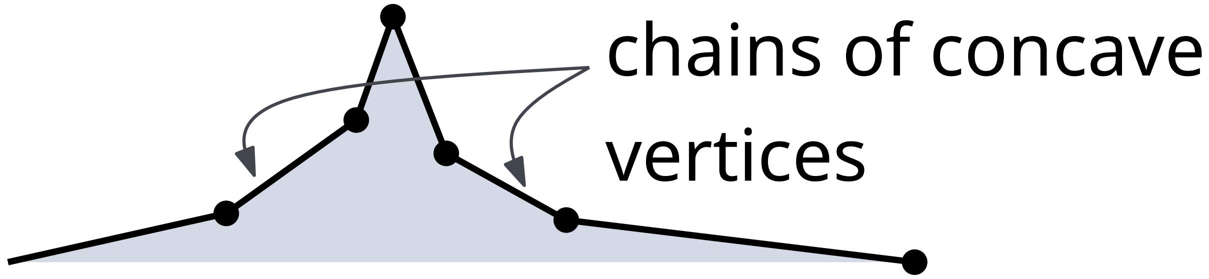
reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

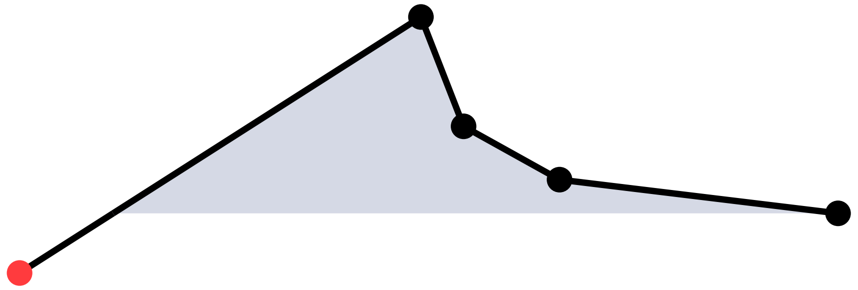


invariant?

untriangulated part above current vertex is an upside-down funnel



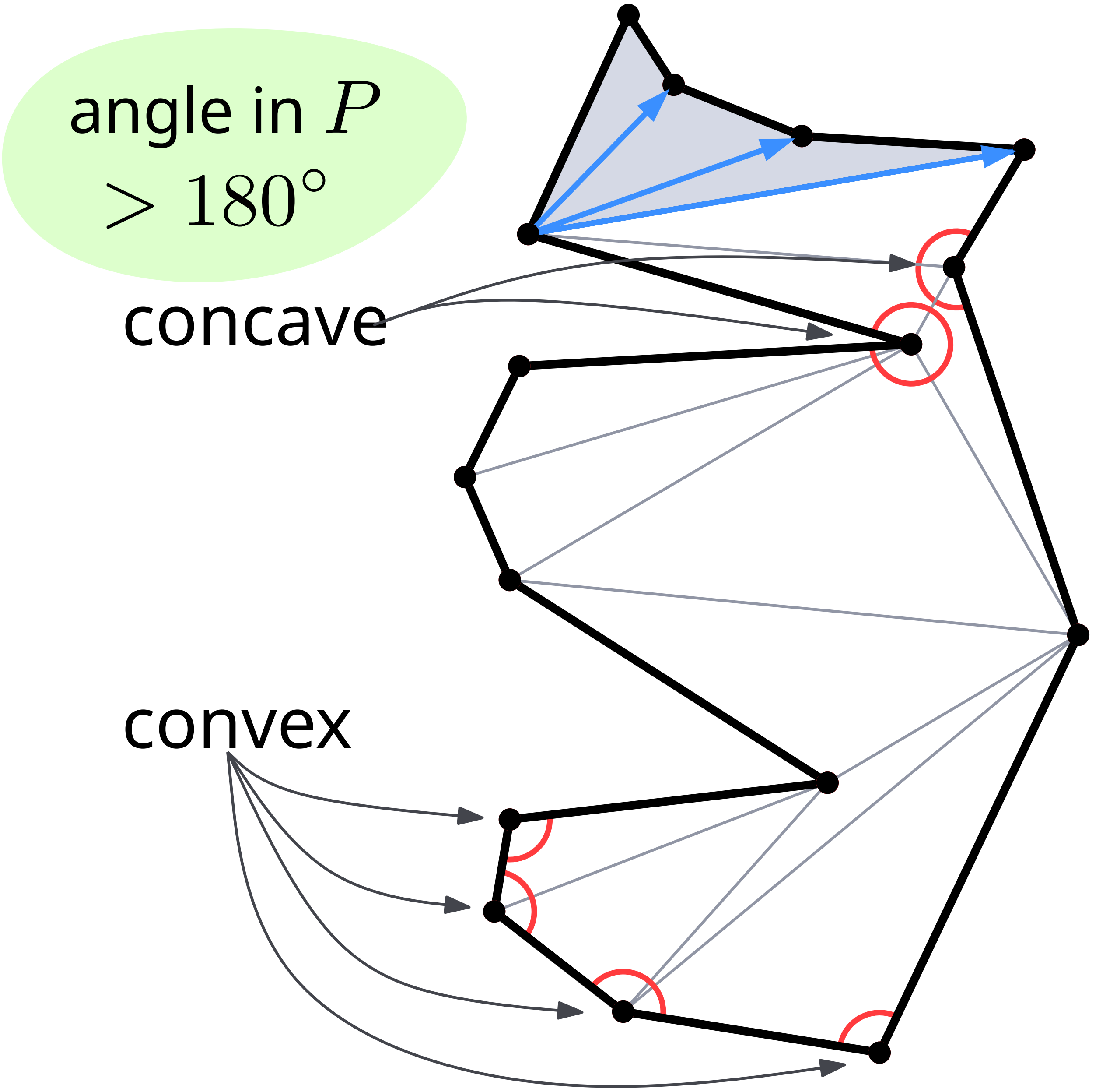
more precisely:



Triangulating a y-monotone Polygon

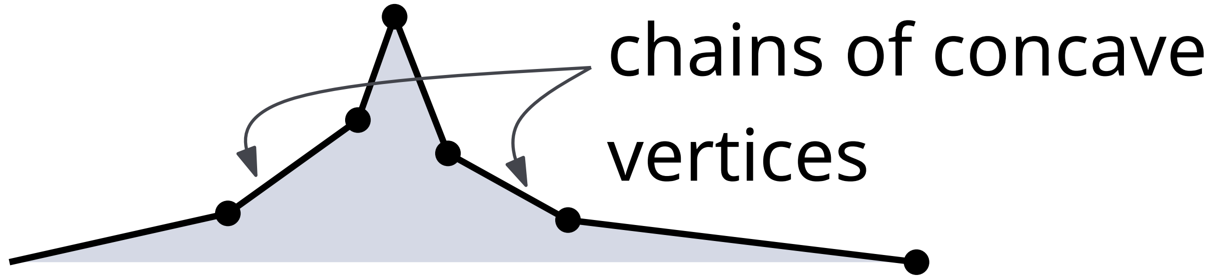
reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

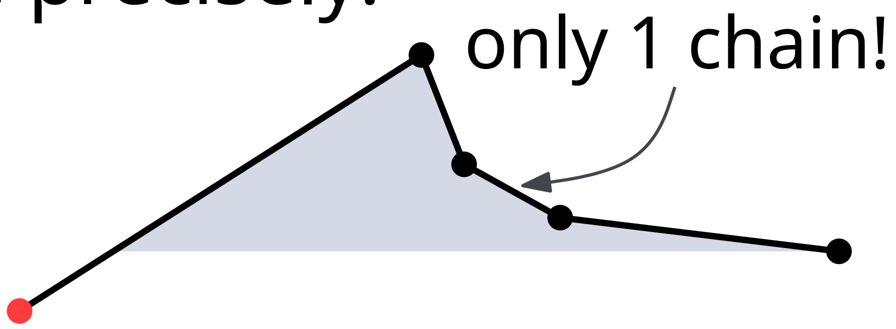


invariant?

untriangulated part above current vertex is an upside-down funnel



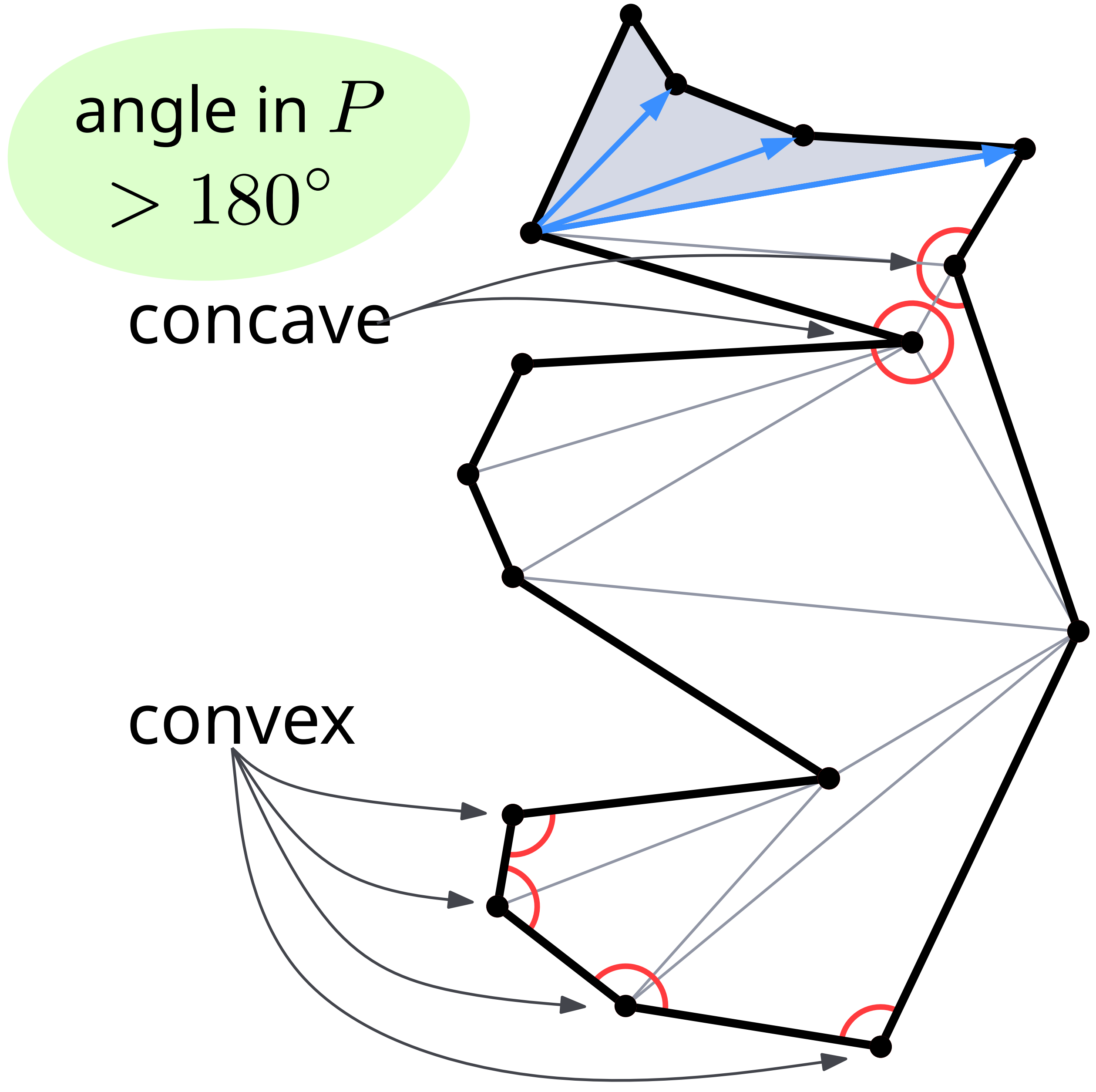
more precisely:



Triangulating a y-monotone Polygon

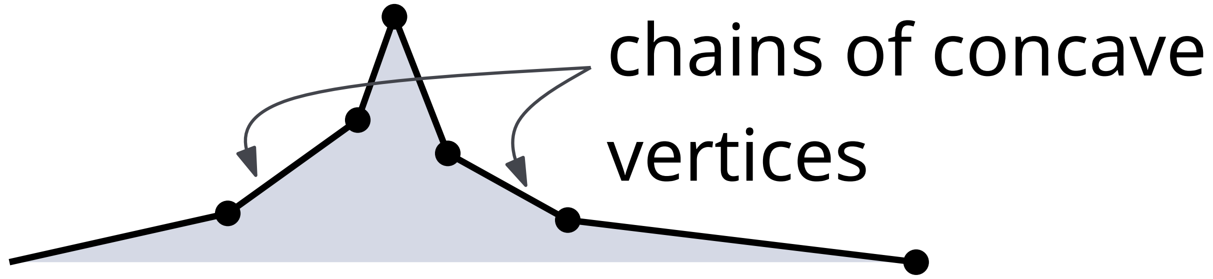
reminder: boundary chains from top to bottom only go down

approach: greedy, on both sides top-down

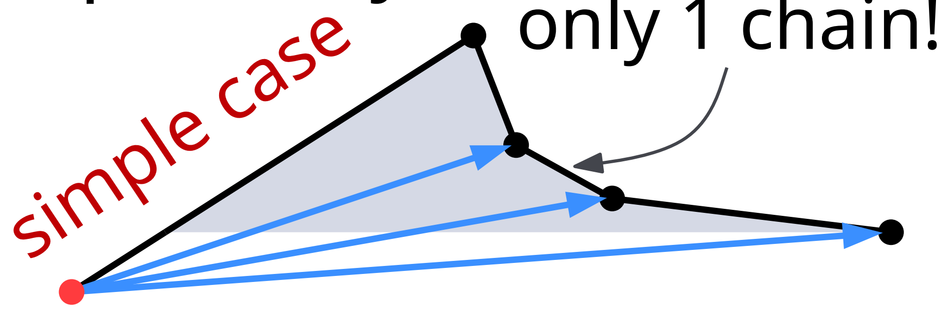


invariant?

untriangulated part above current vertex is an upside-down funnel



more precisely:



Algorithm TriangulateMonotonePolygon

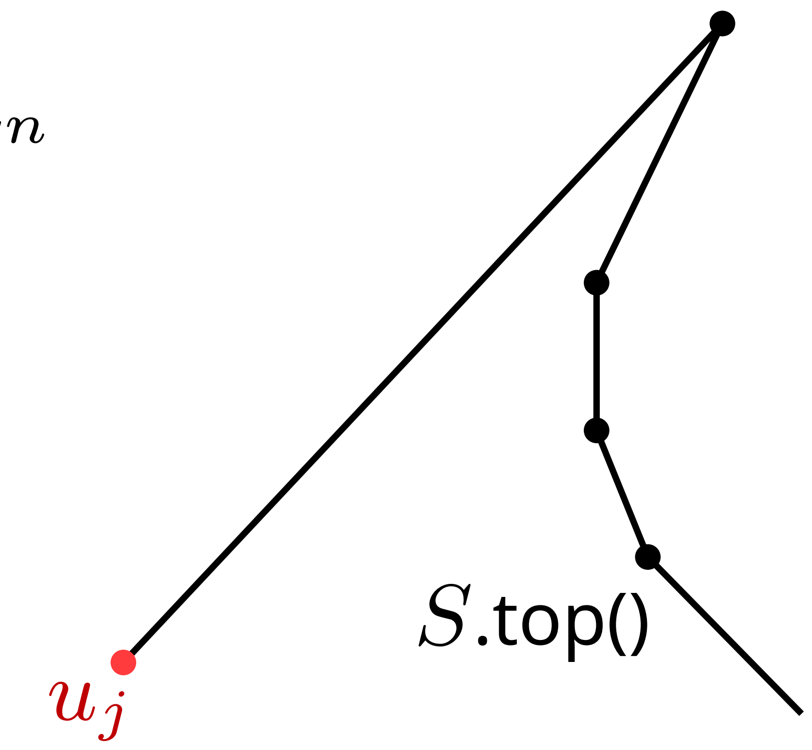
TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

1: merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
2: stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)

Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(polygon P as DCEL)

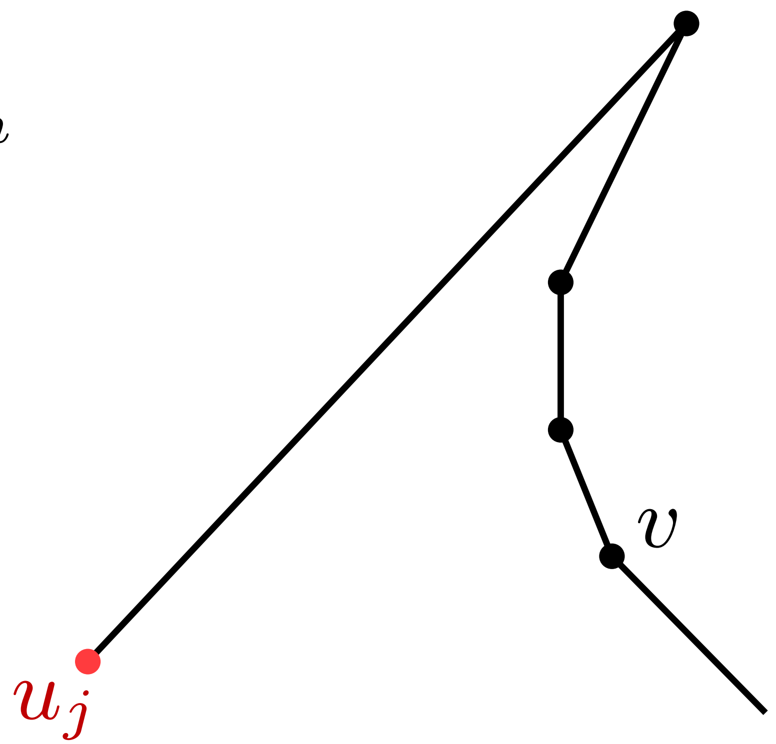
```
merge vertices of left/right boundary → decreasing seq. u_1, ..., u_n
stack S ← ∅; S.push(u_1); S.push(u_2)
for j ← 3 to n − 1 do
    if u_j and S.top() on different boundaries then
        while S is not empty do
            v ← S.pop()
            if S is not empty then
                add (u_j, v)
        S.push(u_{j−1}); S.push(u_j)
    else
```



Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

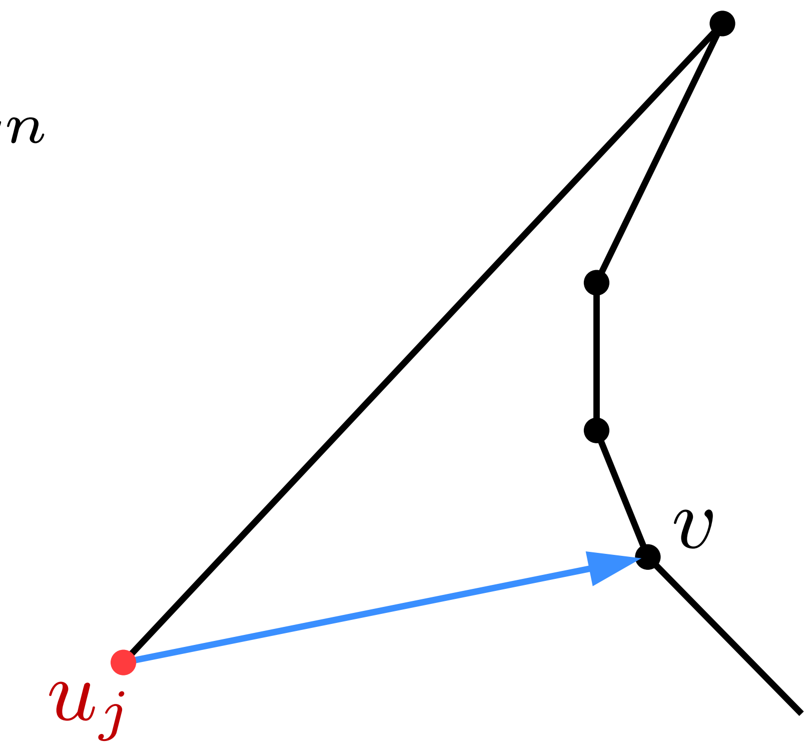
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else



Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(polygon P as DCEL)

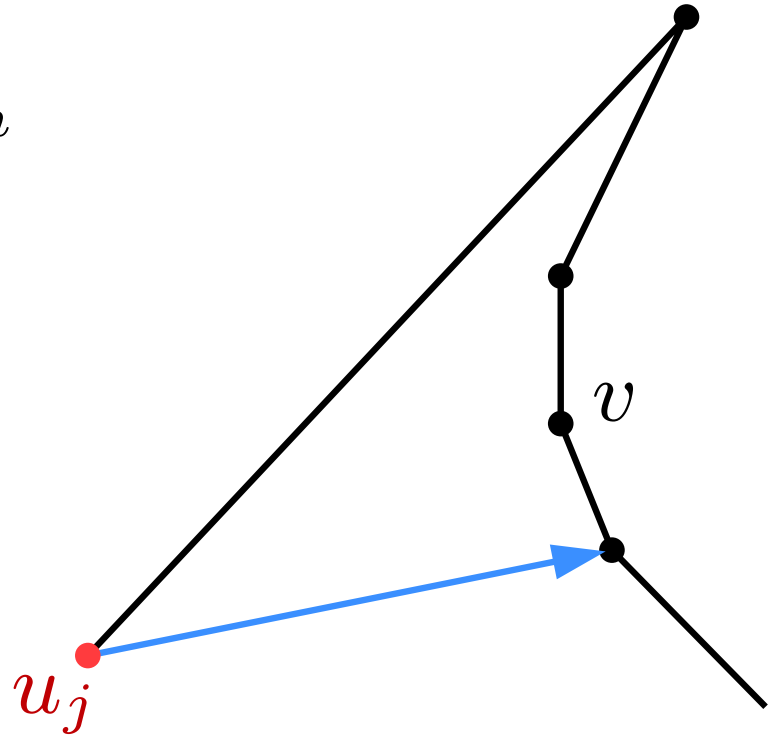
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else



Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

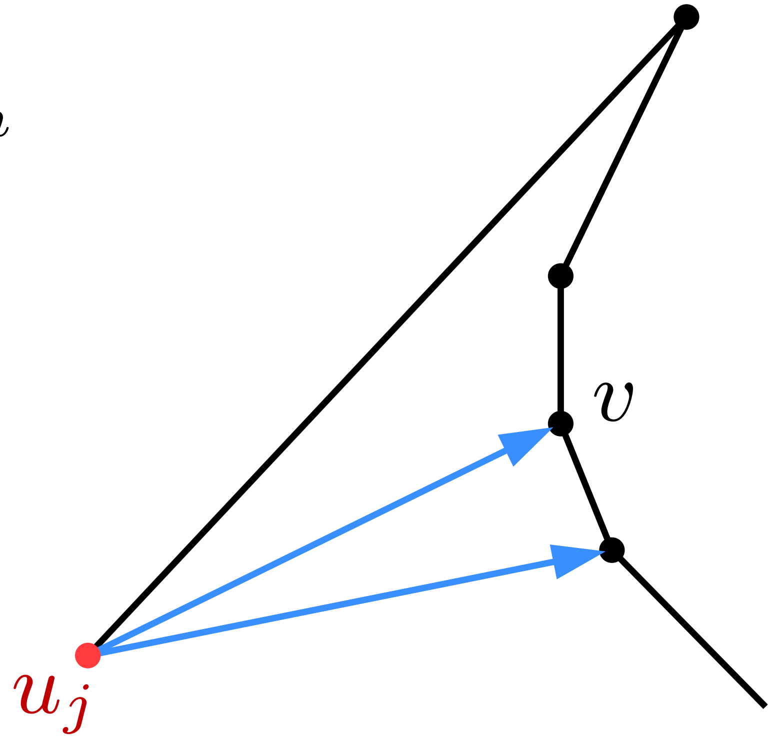
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else



Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(polygon P as DCEL)

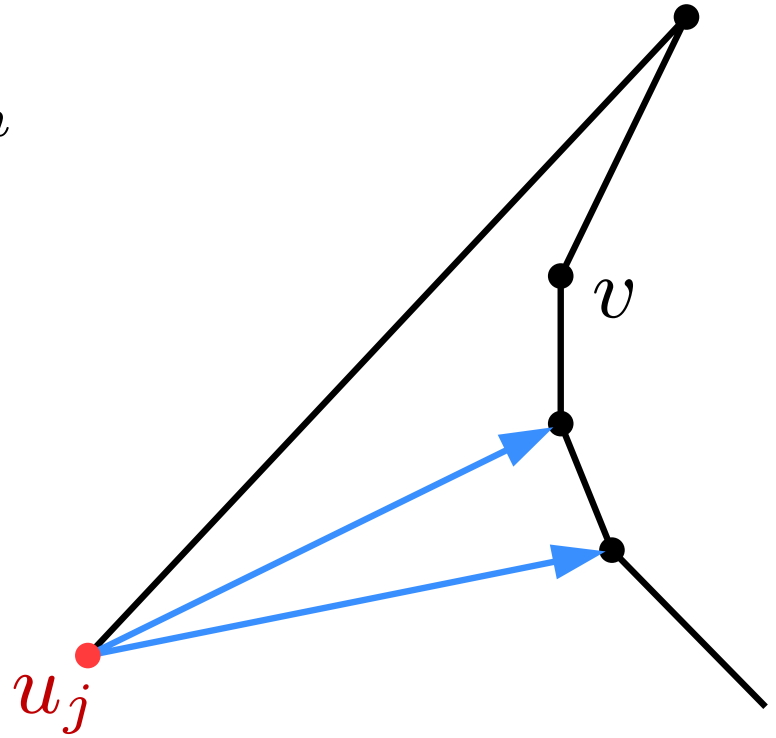
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else



Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(polygon P as DCEL)

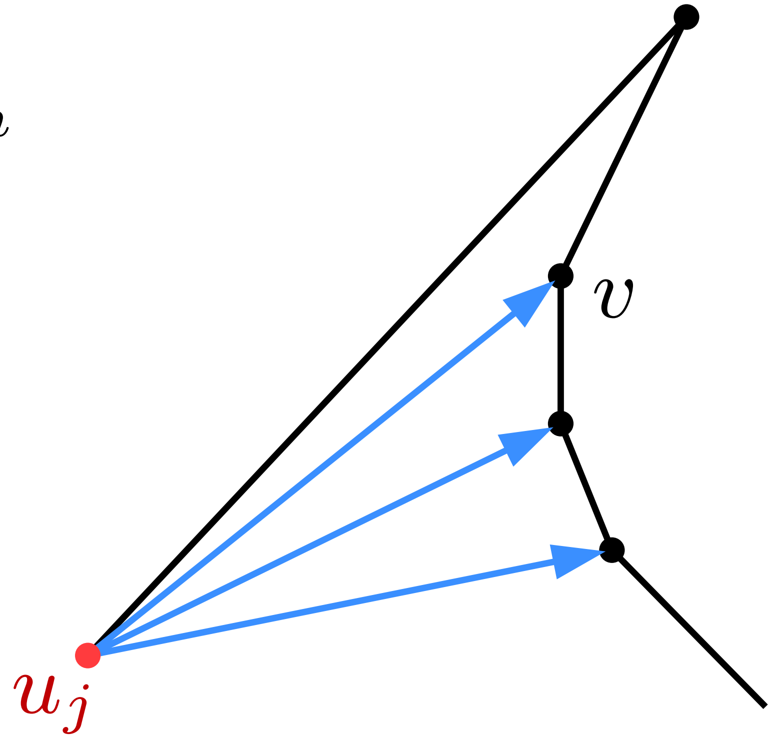
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else



Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

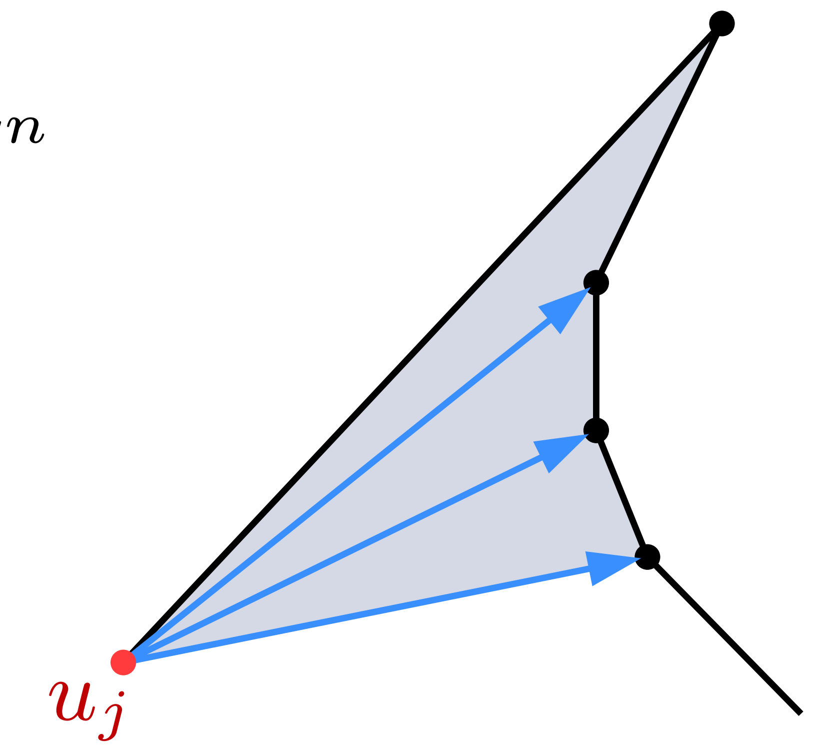
```
merge vertices of left/right boundary → decreasing seq. u_1, ..., u_n
stack S ← ∅; S.push(u_1); S.push(u_2)
for j ← 3 to n − 1 do
    if u_j and S.top() on different boundaries then
        while S is not empty do
            v ← S.pop()
            if S is not empty then
                add (u_j, v)
        S.push(u_{j−1}); S.push(u_j)
    else
```



Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

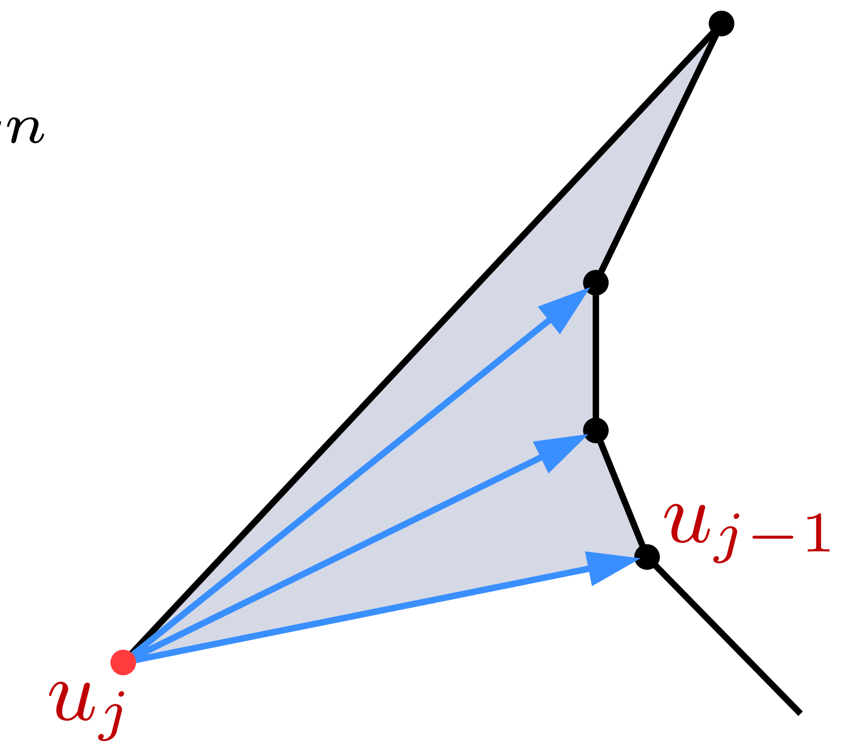
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else



Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

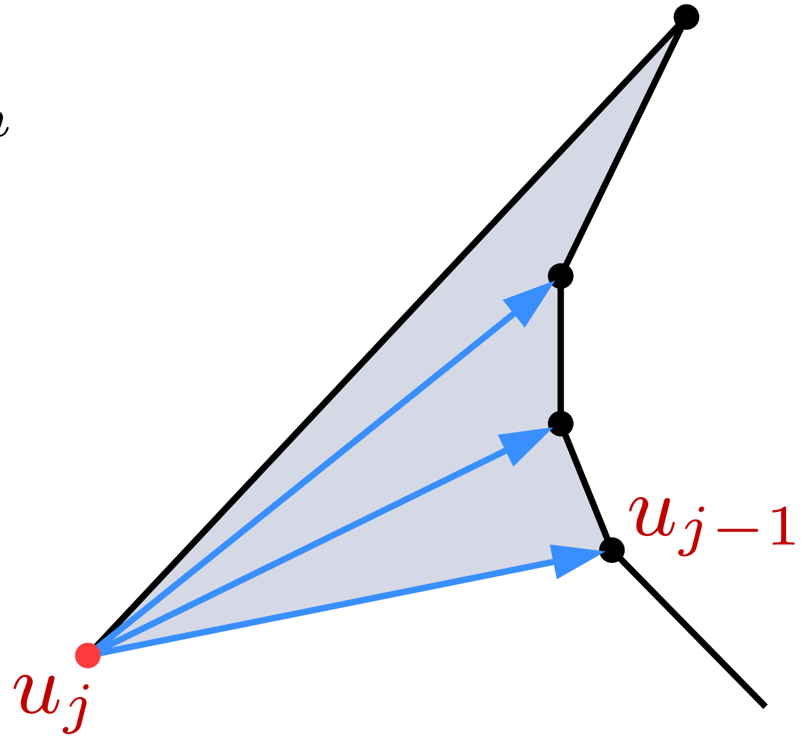
merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \varnothing$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else

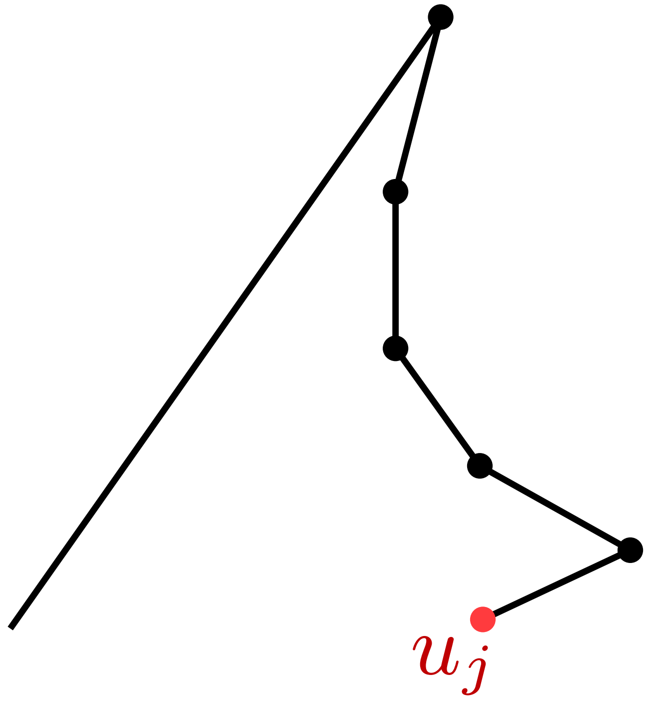


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)

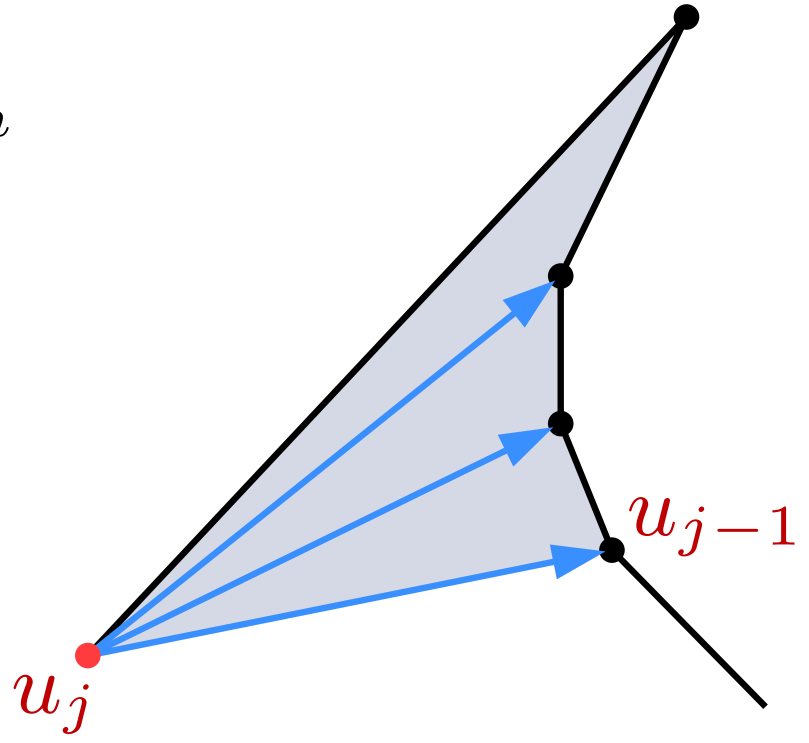


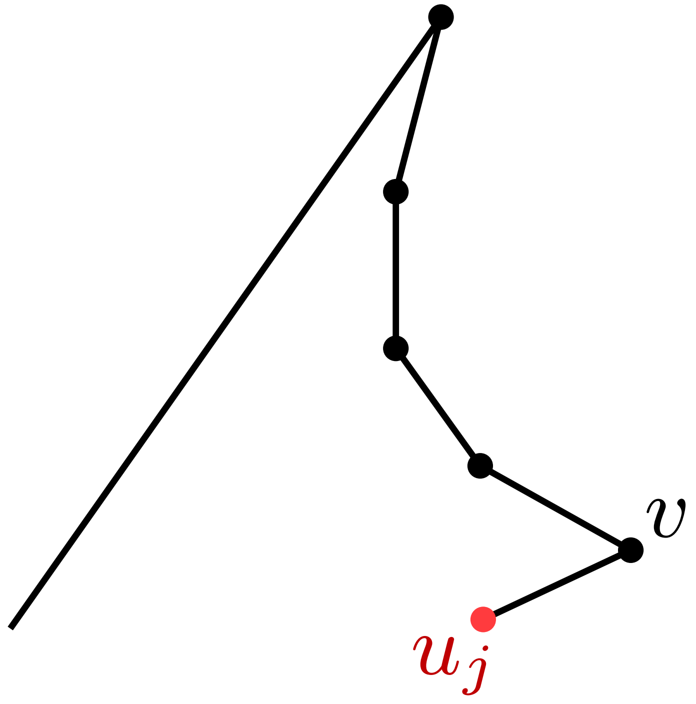


Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(polygon P as DCEL)

merge vertices of left/right boundary $\to$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
  if u_j and S.top() on different boundaries **then**
    while S is not empty **do**
      $v \leftarrow S$.pop()
      if S is not empty **then**
        add (u_j, v)
    S.push(u_{j-1}); S.push(u_j)
  else
    $v \leftarrow S$.pop()
    while S is not empty **and** u_j sees S.top() **do**
      $v \leftarrow S$.pop()
      add diagonal (u_j, v)
    S.push(v); S.push(u_j)

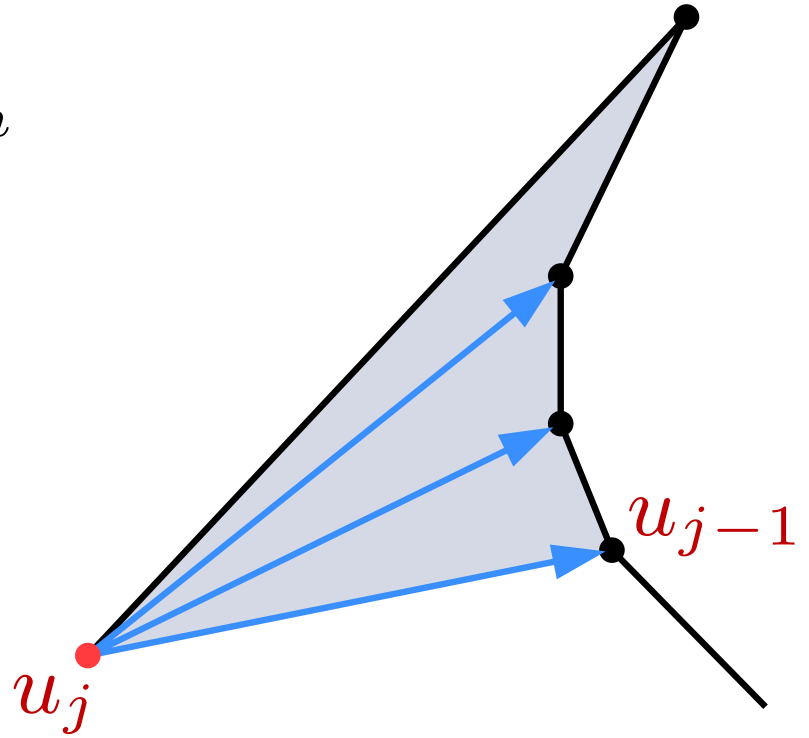


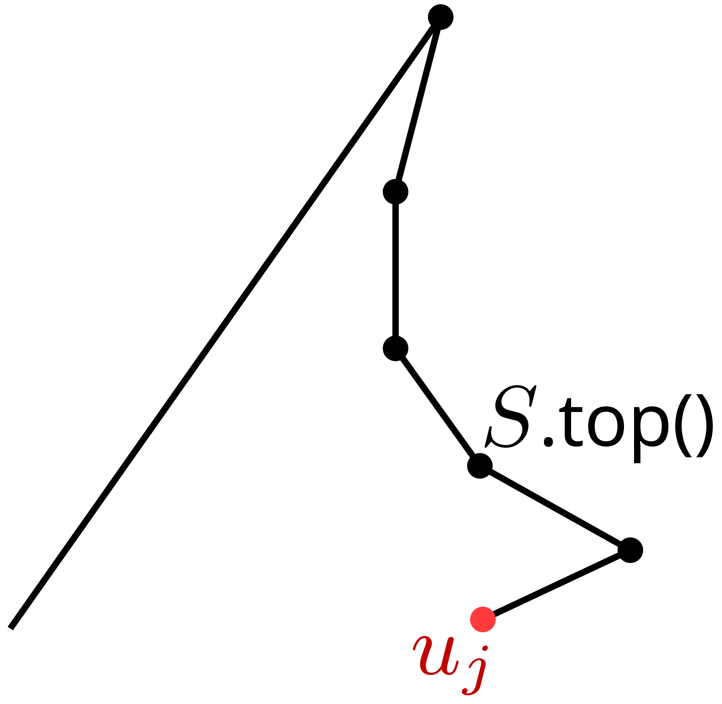


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)

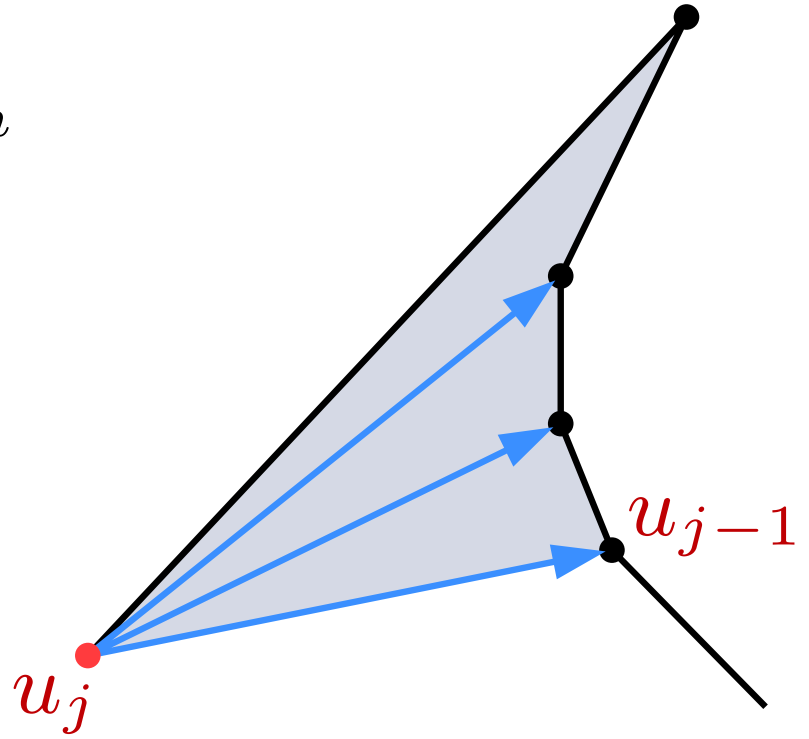


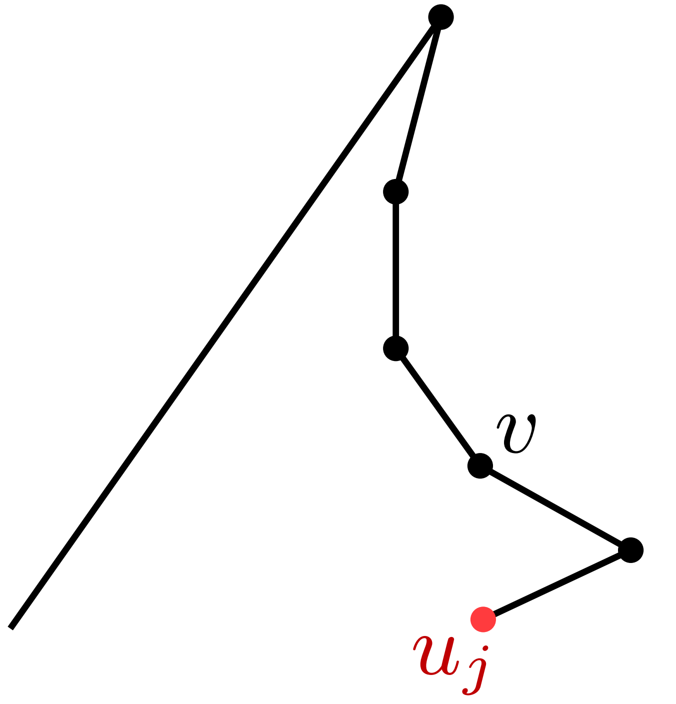


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
  if u_j and S.top() on different boundaries **then**
    while S is not empty **do**
      $v \leftarrow S$.pop()
      if S is not empty **then**
        add (u_j, v)
    S.push(u_{j-1}); S.push(u_j)
  else
    $v \leftarrow S$.pop()
    while S is not empty **and** u_j sees S.top() **do**
      $v \leftarrow S$.pop()
      add diagonal (u_j, v)
    S.push(v); S.push(u_j)

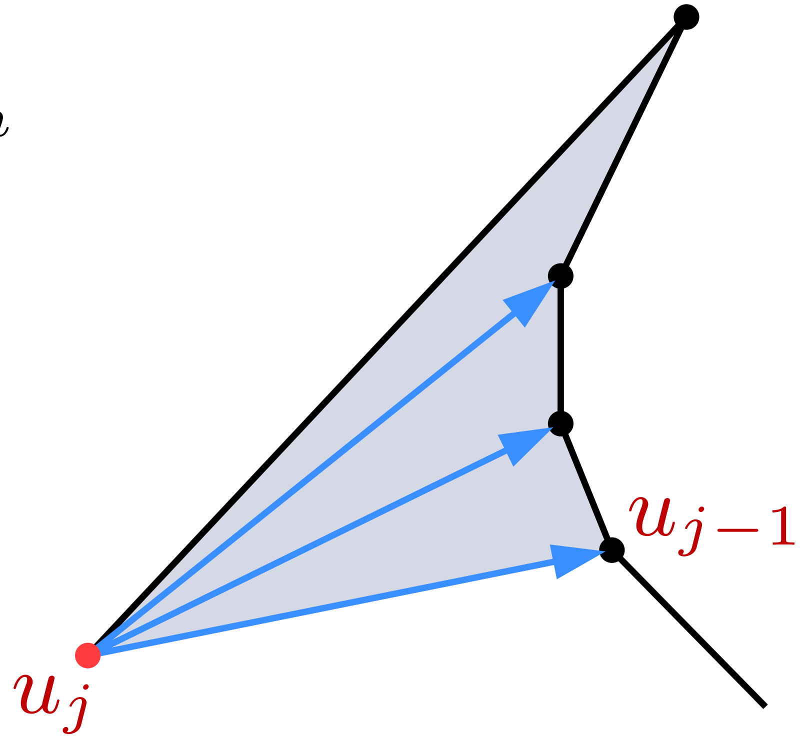


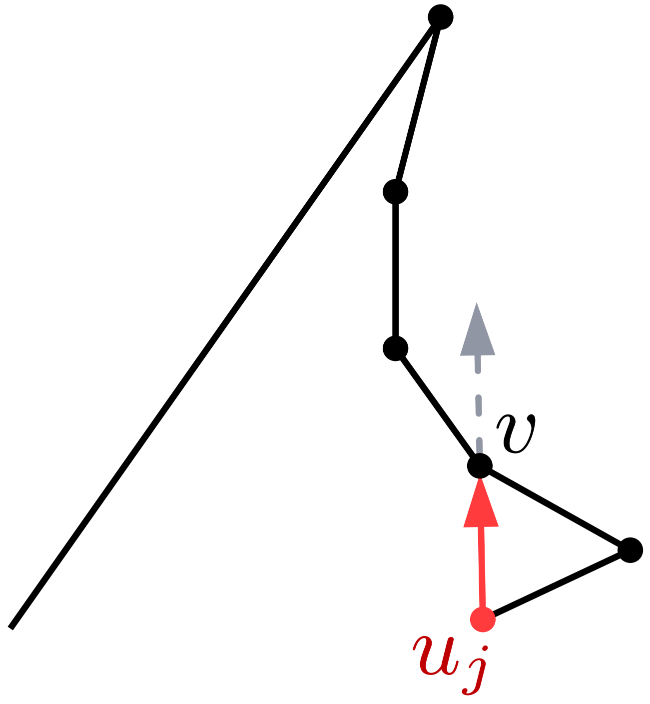


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\to$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)

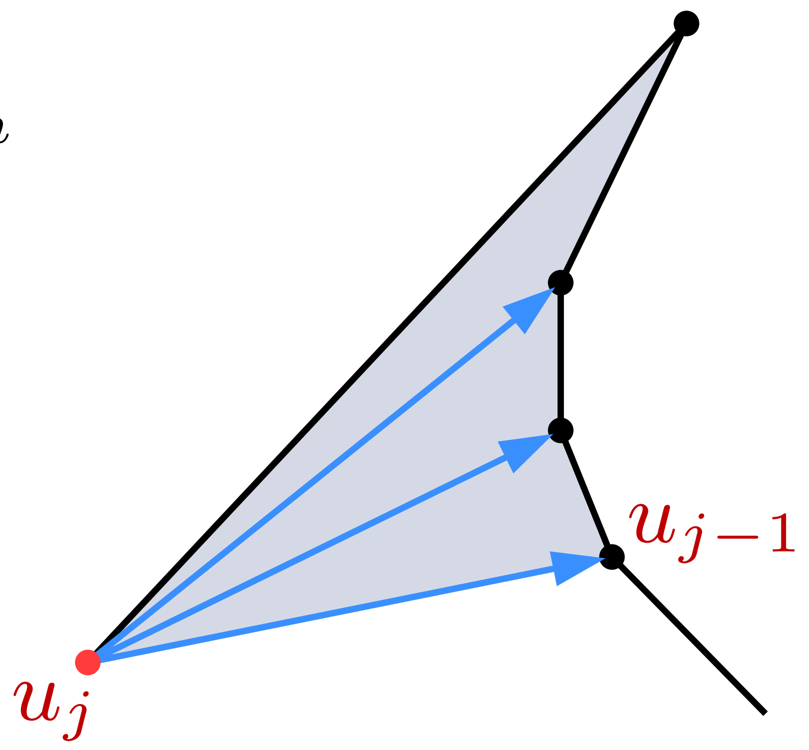


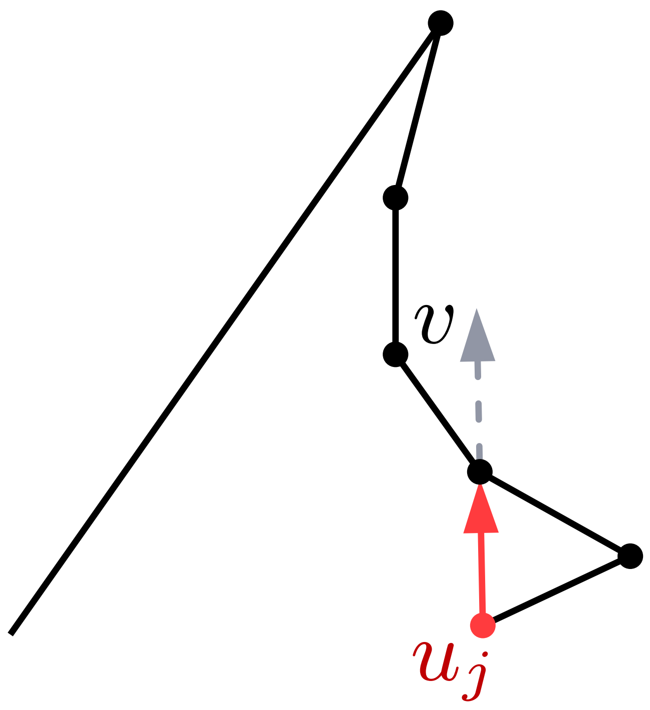


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \varnothing$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)

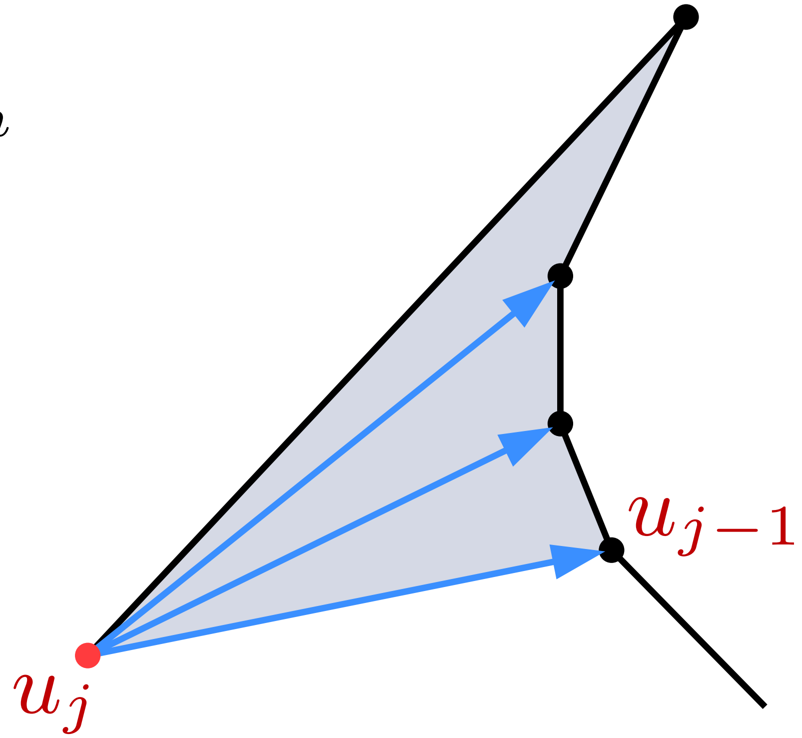


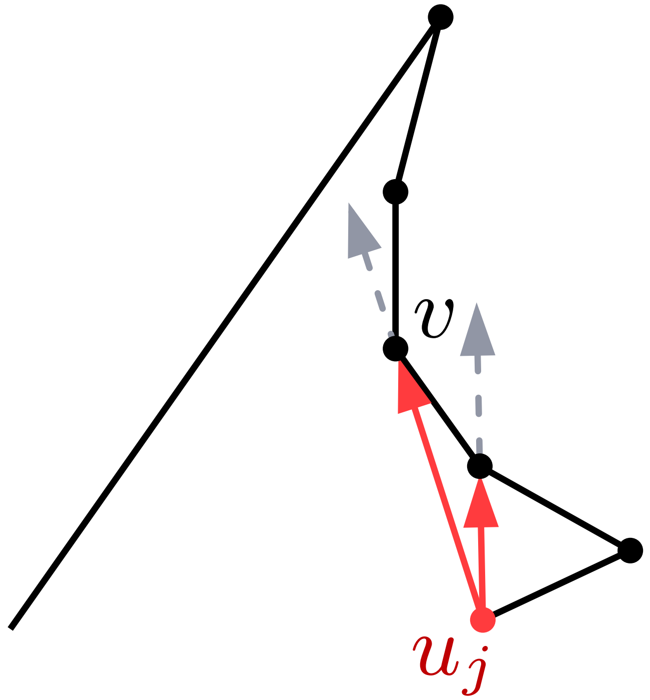


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)

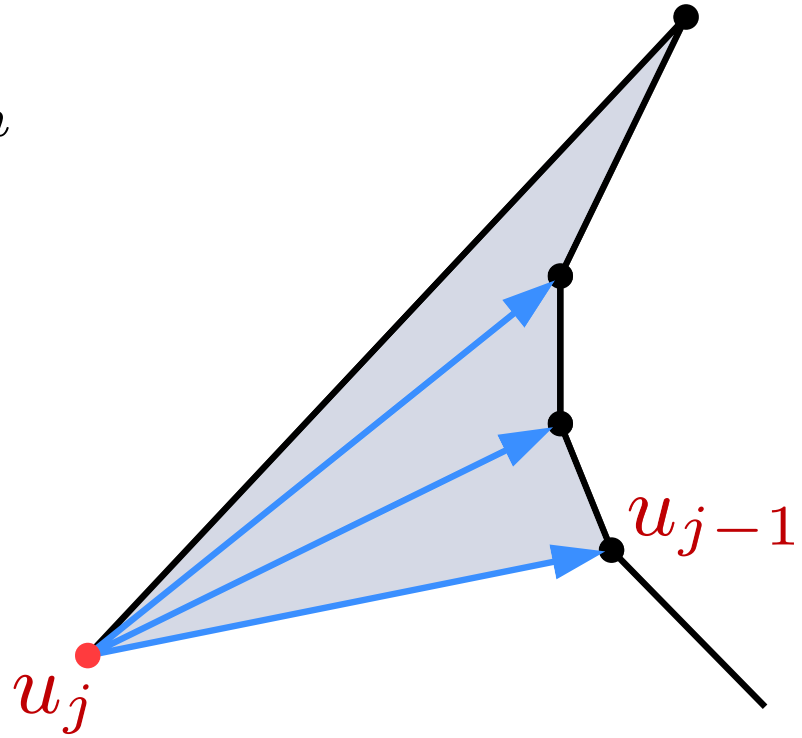


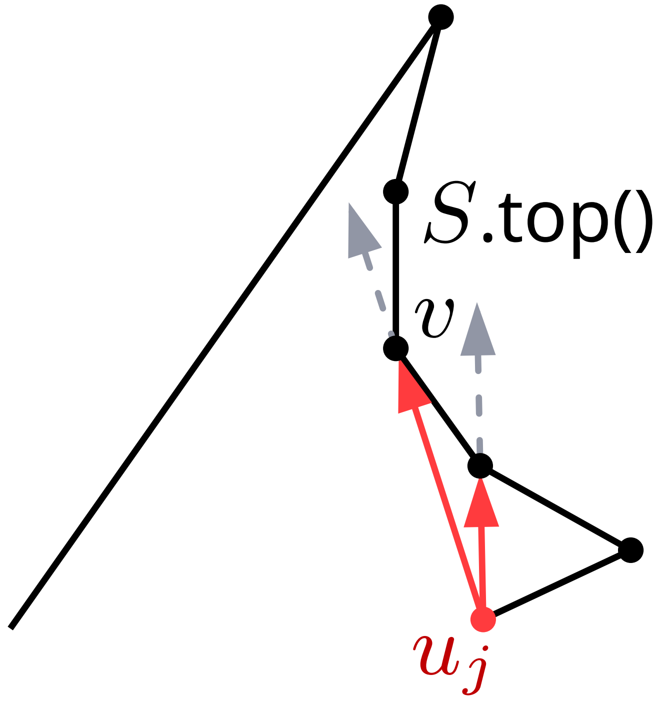


Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(polygon P as DCEL)

merge vertices of left/right boundary $\to$ decreasing seq. $u_1, \dots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
  if u_j and S.top() on different boundaries **then**
    while S is not empty **do**
      $v \leftarrow S$.pop()
      if S is not empty **then**
        add (u_j, v)
    S.push(u_{j-1}); S.push(u_j)
  else
    $v \leftarrow S$.pop()
    while S is not empty **and** u_j sees S.top() **do**
      $v \leftarrow S$.pop()
      add diagonal (u_j, v)
    S.push(v); S.push(u_j)

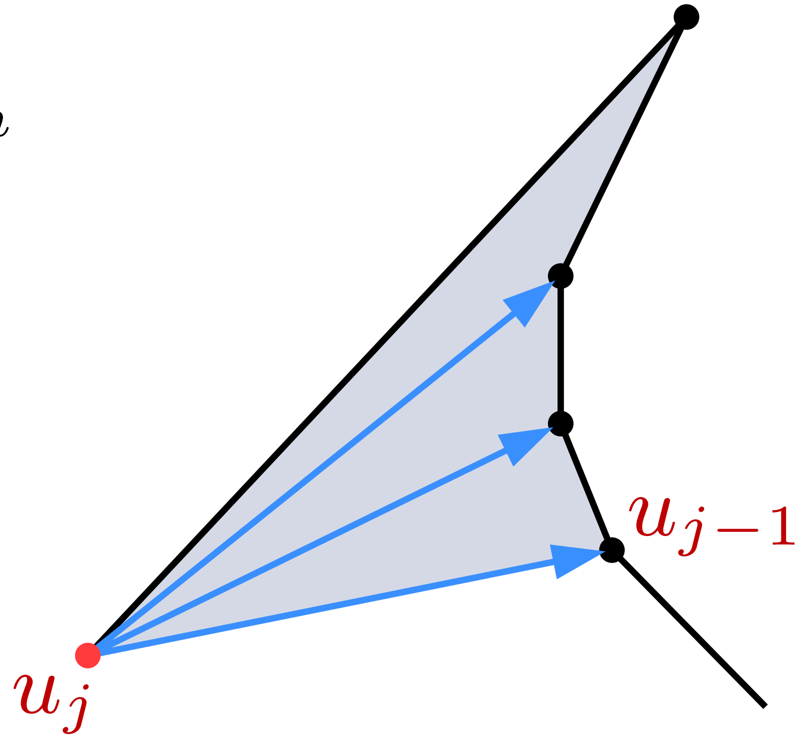


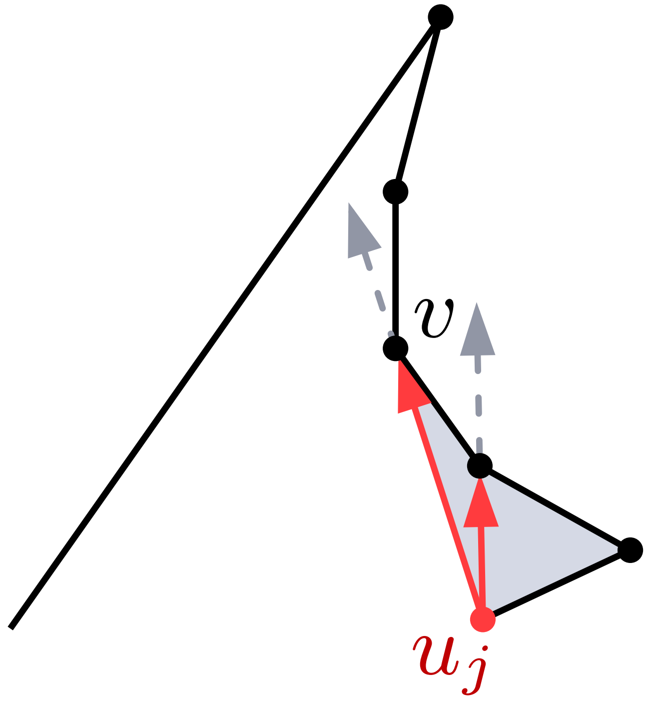


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)

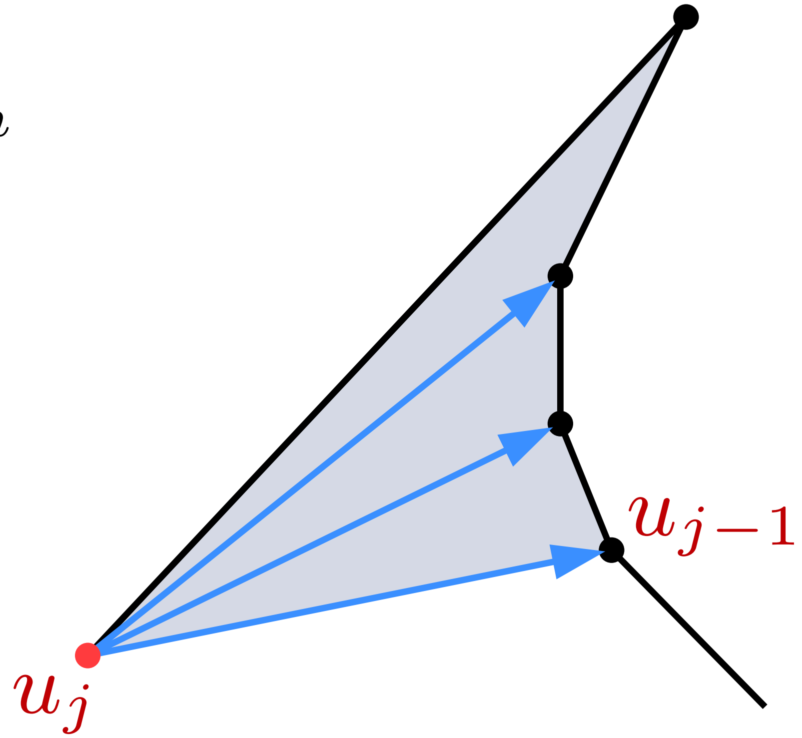


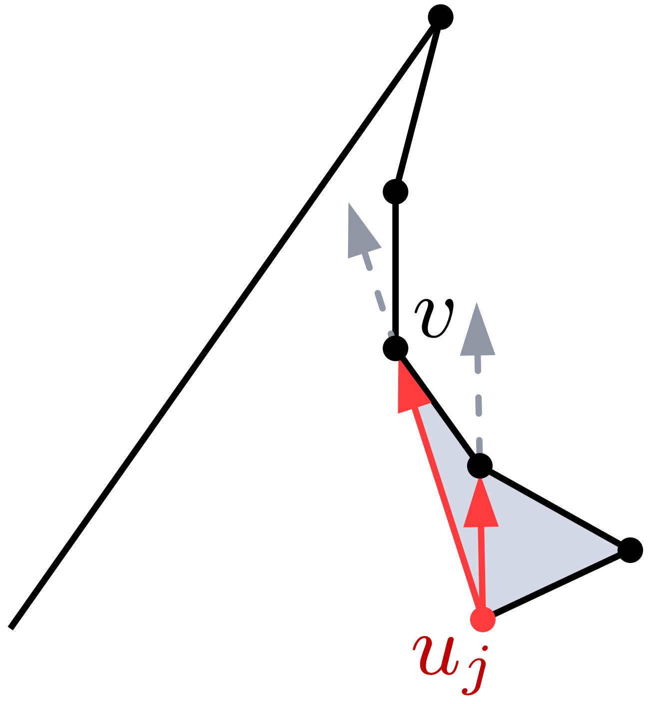


Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)
connect u_n to all vertices in S (except first and last)





Algorithm TriangulateMonotonePolygon

TRIANGULATEMONOTONEPOLYGON(polygon P as DCEL)

merge vertices of left/right boundary $\rightarrow$ decreasing seq. $u_1, \ldots, u_n$
stack $S \leftarrow \emptyset$; S.push(u_1); S.push(u_2)
for $j \leftarrow 3$ to $n-1$ **do**
    if u_j and S.top() on different boundaries **then**
        while S is not empty **do**
            $v \leftarrow S$.pop()
            if S is not empty **then**
                add (u_j, v)
        S.push(u_{j-1}); S.push(u_j)
    else
        $v \leftarrow S$.pop()
        while S is not empty **and** u_j sees S.top() **do**
            $v \leftarrow S$.pop()
            add diagonal (u_j, v)
        S.push(v); S.push(u_j)
connect u_n to all vertices in S (except first and last)

Question:
running time?

Summary (Triangulation)

Theorem 4: A y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Summary (Triangulation)

Theorem 4: A y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(n \log n)$ time using $O(n)$ space into y-monotone polygons.

Summary (Triangulation)

Theorem 4: A y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(n \log n)$ time using $O(n)$ space into y-monotone polygons.

$$\Downarrow$$

Theorem 5: A simple polygon with n vertices can be triangulated in $O(n \log n)$ time using $O(n)$ space.

Summary (Triangulation)

Theorem 4: A y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(n \log n)$ time using $O(n)$ space into y-monotone polygons.

$\Downarrow$ Does this follow immediately?

Theorem 5: A simple polygon with n vertices can be triangulated in $O(n \log n)$ time using $O(n)$ space.

Summary (Triangulation)

Theorem 4: A y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Theorem 3: A simple polygon with n vertices can be partitioned in $O(n \log n)$ time using $O(n)$ space into y-monotone polygons.

at most $n - 3$ diagonals added,
$\Downarrow$ each is part of 2 y-monotone polygons
$\Rightarrow$ summed complexity of y-monotone polygons is $O(n)$

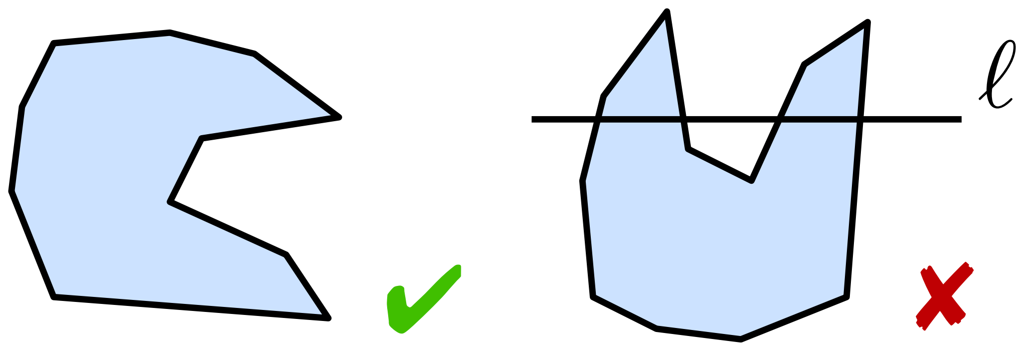
Theorem 5: A simple polygon with n vertices can be triangulated in $O(n \log n)$ time using $O(n)$ space.

Summary (Art Gallery Problem)

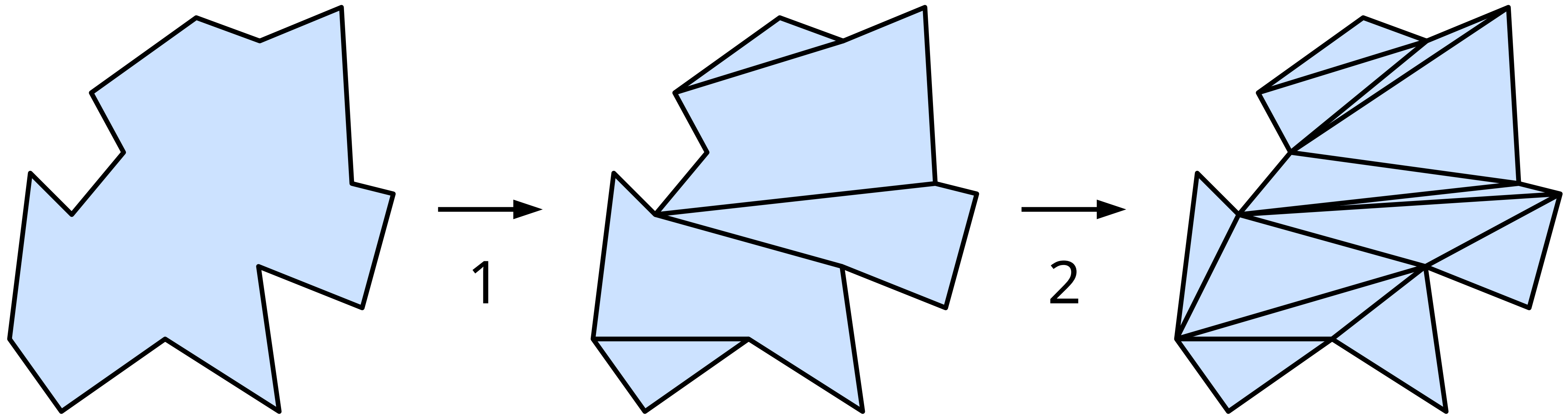
2-step procedure:

- step 1: partition P into y-monotone subpolygons

Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.



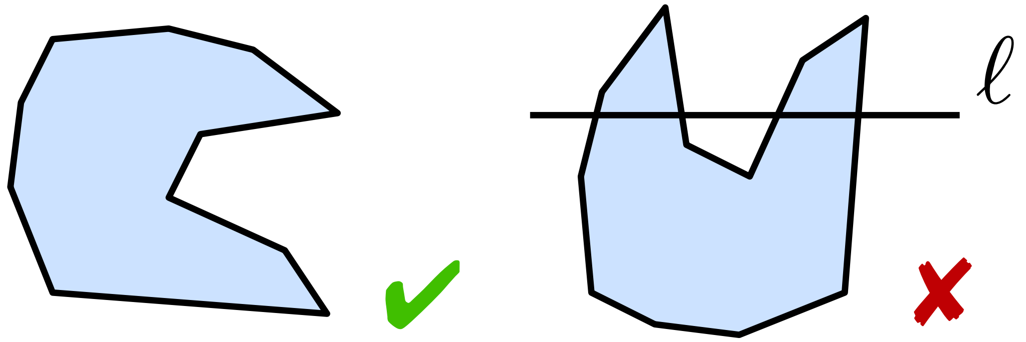
- step 2: triangulate y-monotone subpolyons



Summary (Art Gallery Problem)

2-step procedure:

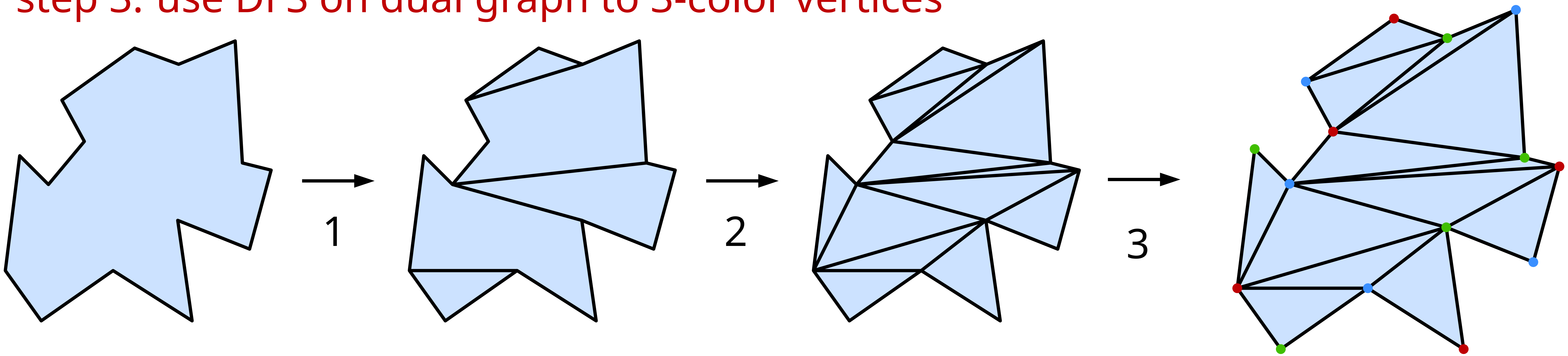
- step 1: partition P into y-monotone subpolygons



Definition: A polygon P is y-monotone if, for every horizontal line ℓ, the intersection $\ell \cap P$ is connected.

- step 2: triangulate y-monotone subpolyons

- step 3: use DFS on dual graph to 3-color vertices



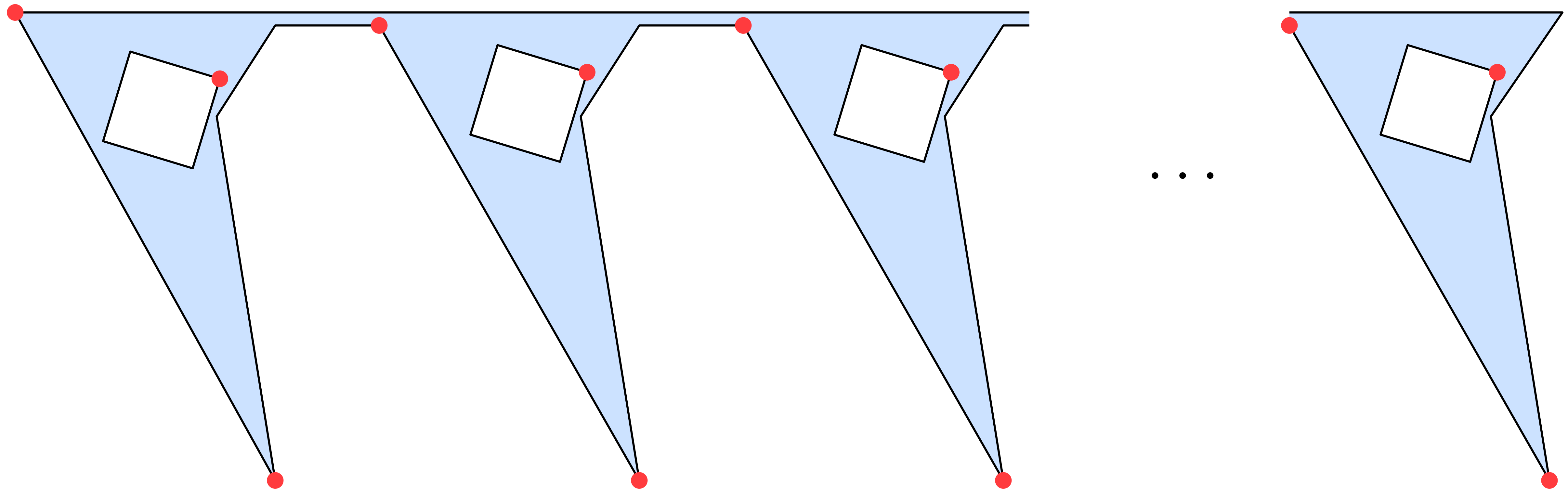
Discussion

Does the triangulation algorithm generalize to polygons with holes?

Discussion

Does the triangulation algorithm generalize to polygons with holes?

- triangulation: yes
- but are $\lfloor n/3 \rfloor$ cameras sufficient?
  No, generalization of the art gallery theorem gives: sometimes $\lfloor (n+h)/3 \rfloor$ cameras are needed [Hoffmann et al. '91]



Discussion

Does the triangulation algorithm generalize to polygons with holes?

- triangulation: yes
- but are $\lfloor n/3 \rfloor$ cameras sufficient?
  No, generalization of the art gallery theorem gives: sometimes $\lfloor (n+h)/3 \rfloor$ cameras are needed [Hoffmann et al. ’91]

Can we triangulate general simple polygons faster?

Discussion

Does the triangulation algorithm generalize to polygons with holes?

- triangulation: yes
- but are $\lfloor n/3 \rfloor$ cameras sufficient?
  No, generalization of the art gallery theorem gives: sometimes $\lfloor (n+h)/3 \rfloor$ cameras are needed [Hoffmann et al. '91]

Can we triangulate general simple polygons faster?

- Yes. This was an open problem for a long time, until increasingly faster (randomized) algorithms were developed by the end of 1980s
- $O(n)$-time algorithm by Chazelle [1990] (complicated)
- There is an elegant $O(n \log^* n)$ expected-time algorithm [Seidel 1991] (simple)