Karlsruhe Institute of Technology

# Computational Geometry · Lecture

## Polygon Triangulation

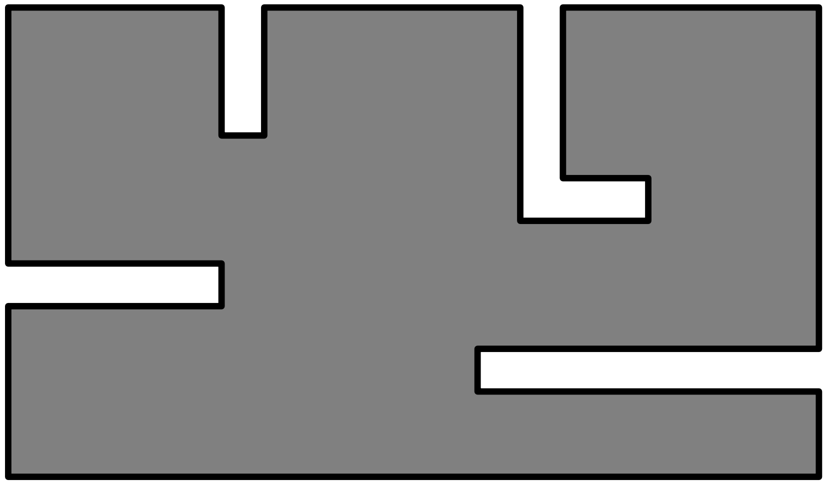
INSTITUTE FOR THEORETICAL INFORMATICS · FACULTY OF INFORMATICS

Tamara Mchedlidze

3.5.2018

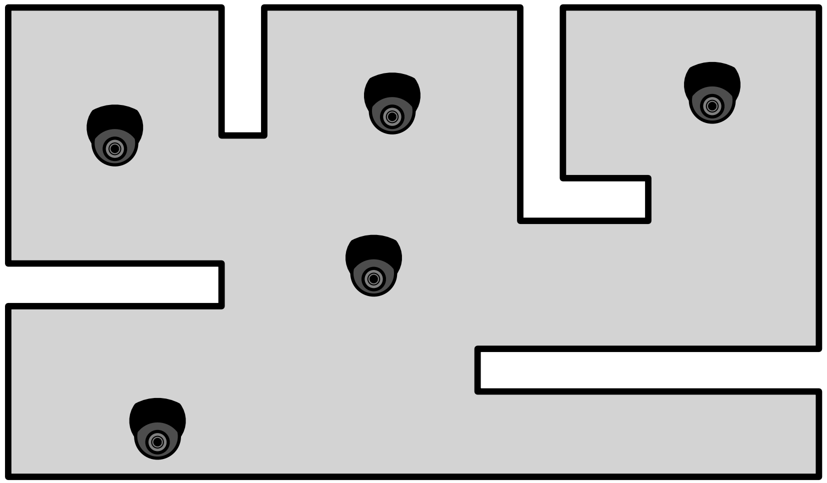
# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.



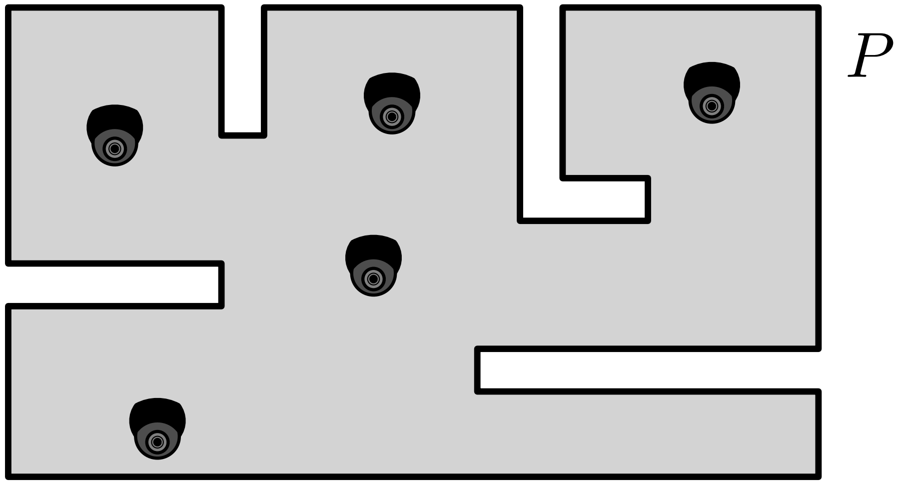
# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.



# The Art-Gallery-Problem

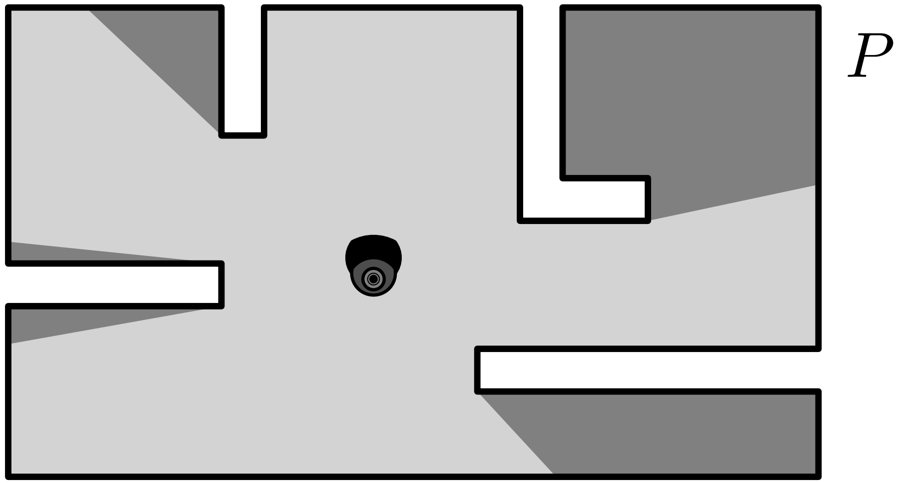
**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.




**Assumption:** Art gallery is a *simple* polygon $P$ with $n$ corners (no self-intersections, no holes)

# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.

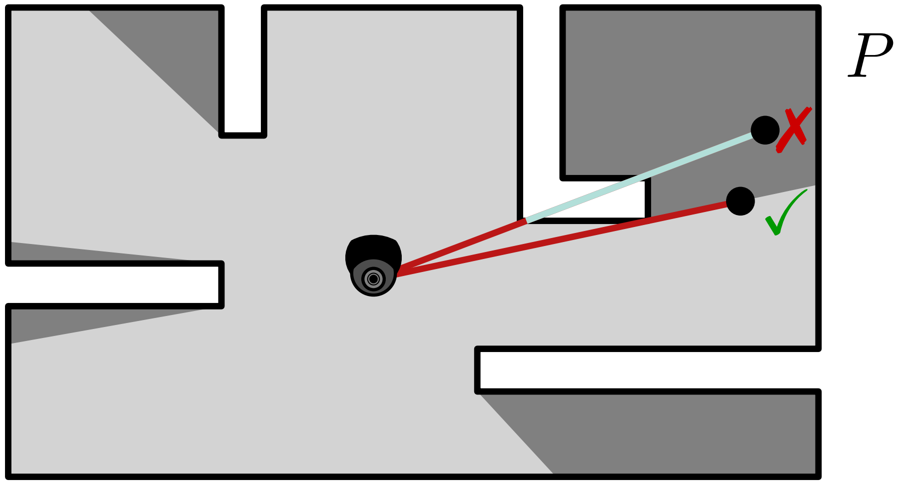



**Assumption:** Art gallery is a *simple* polygon $P$ with $n$ corners (no self-intersections, no holes)

**Observation:** each camera observes a star-shaped region

# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.



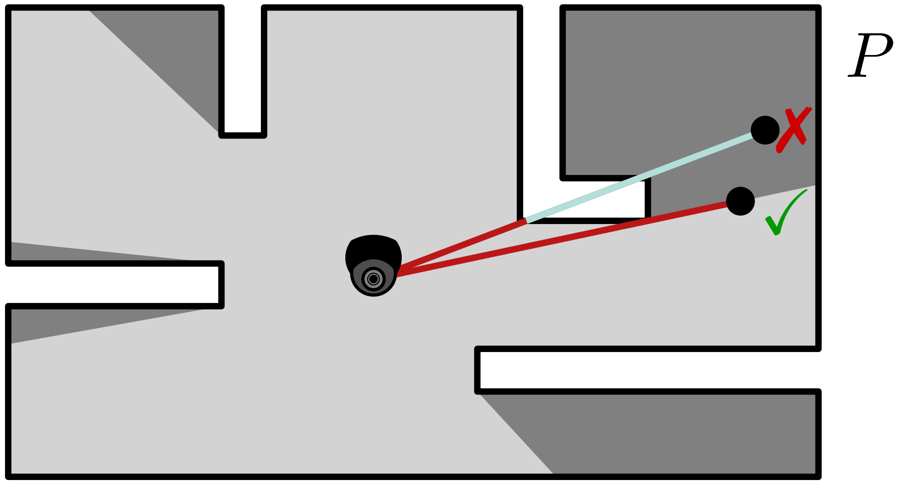

**Assumption:** Art gallery is a *simple* polygon $P$ with $n$ corners (no self-intersections, no holes)

**Observation:** each camera observes a star-shaped region

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

# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.




**Assumption:** Art gallery is a *simple* polygon $P$ with $n$ corners (no self-intersections, no holes)

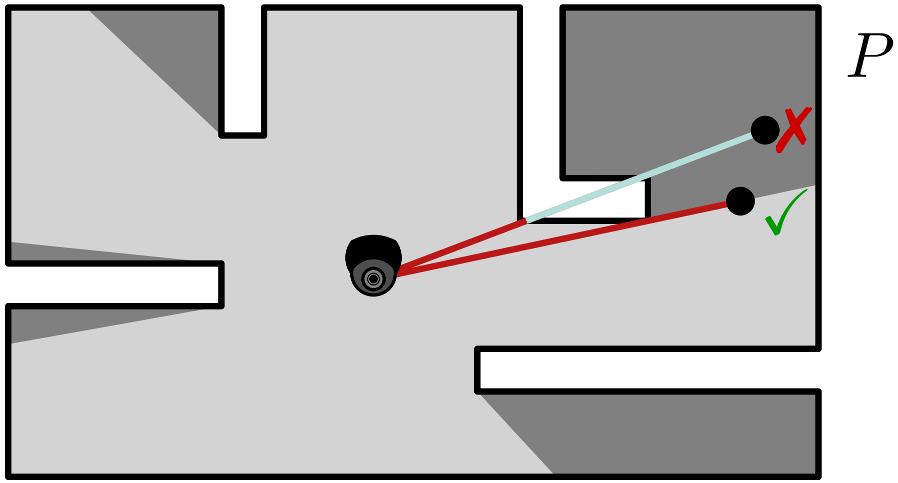
**Observation:** each camera observes a star-shaped region

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

**Goal:** Use as few cameras as possible!

# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.




**Assumption:** Art gallery is a *simple* polygon $P$ with $n$ corners (no self-intersections, no holes)

**Observation:** each camera observes a star-shaped region

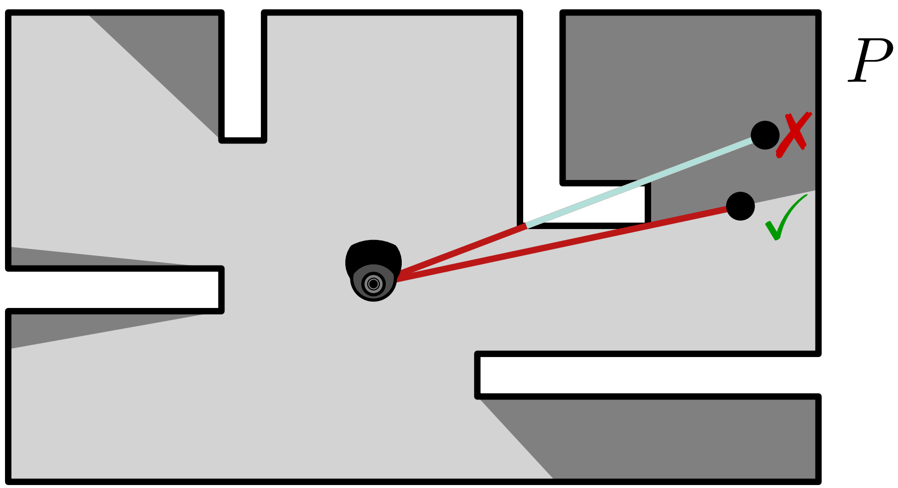
**Definition:** Point $p \in P$ is *visible* from $c \in P$ if $\overline{cp} \in P$

**Goal:** Use as few cameras as possible!

$\rightarrow$ The number depends on the number of corners $n$ and on the shape of $P$

# The Art-Gallery-Problem

**Task:** Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.



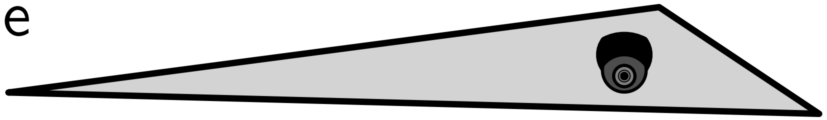

| | |
|---|---|
| **Assumption:** | Art gallery is a *simple* polygon $P$ with $n$ corners (no self-intersections, no holes) |
| **Observation:** | each camera observes a star-shaped region |
| **Definition:** | Point $p \in P$ is *visible* from $c \in P$ if $\overline{cp} \in P$ |
| **Goal:** | Use as few cameras as possible! |

NP-hard!

$\rightarrow$ The number depends on the number of corners $n$ and on the shape of $P$

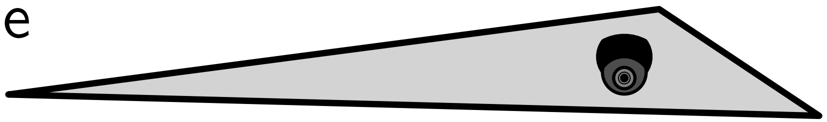
# Problem Simplification

**Observation:** It is easy to guard a triangle

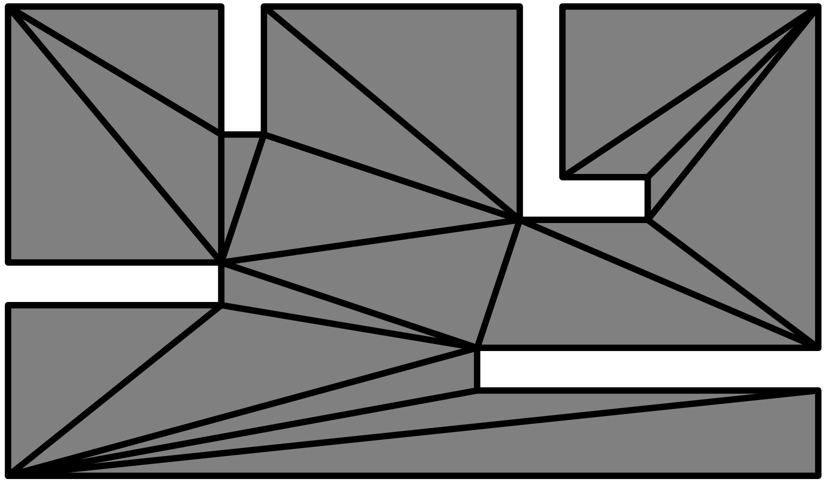


# Problem Simplification

**Observation:** It is easy to guard a triangle

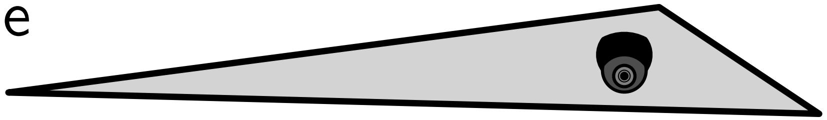


**Idea:** Decompose $P$ into triangles and guard each of them

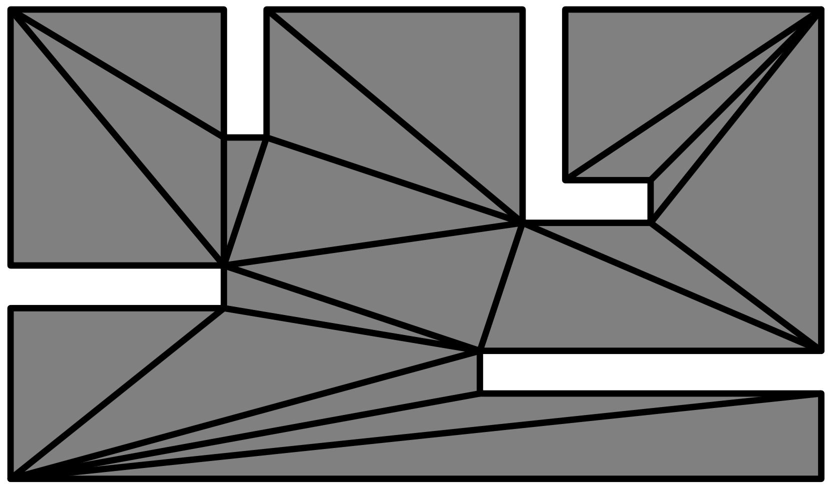


# Problem Simplification

**Observation:** It is easy to guard a triangle



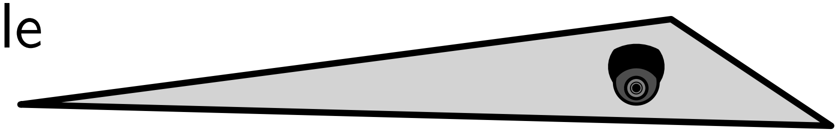
**Idea:** Decompose $P$ into triangles and guard each of them



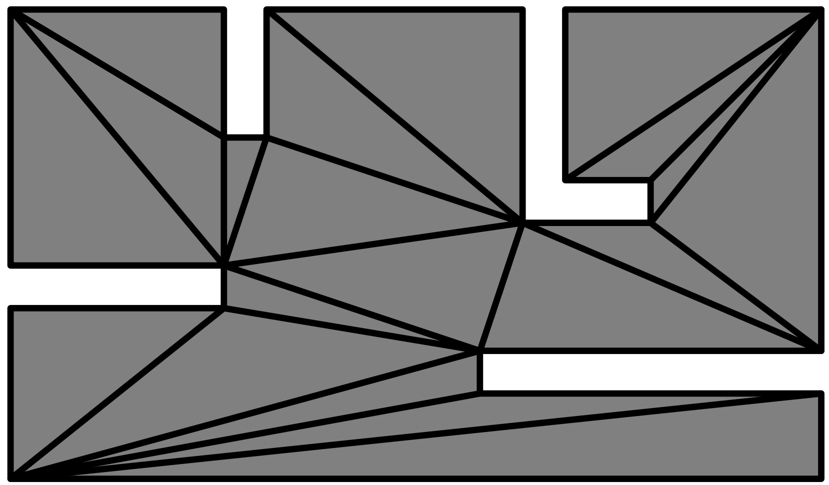
**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them

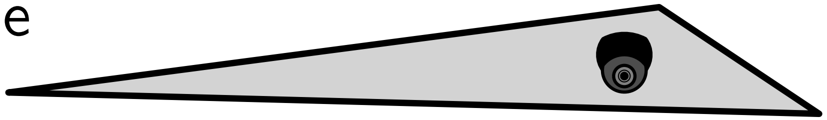


**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

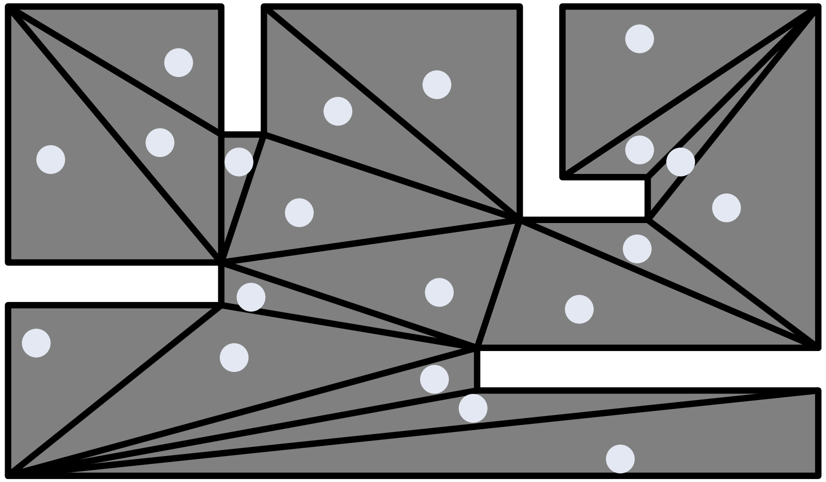
The proof implies a recursive $O(n^2)$-Algorithm!

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them

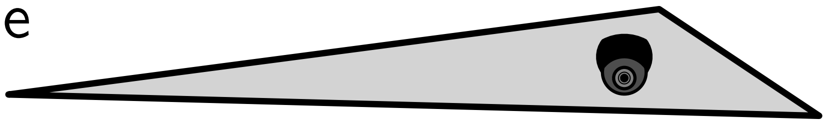


**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

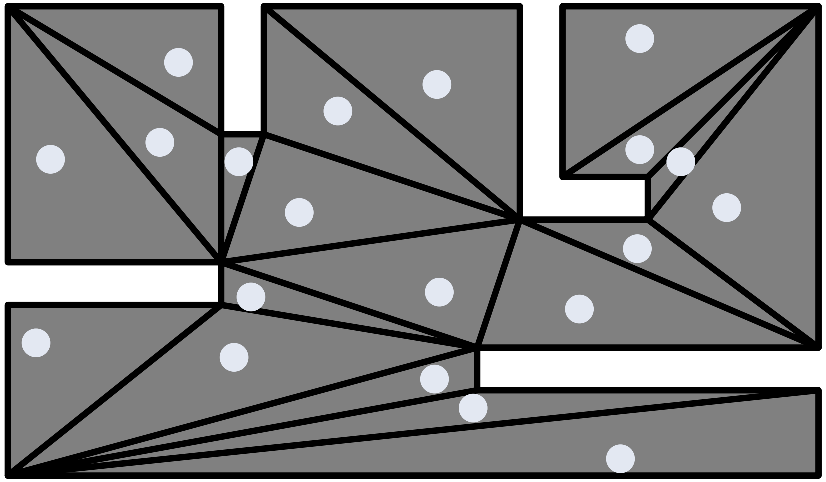
- $P$ could be guarded by $n-2$ cameras placed in the triangles

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them



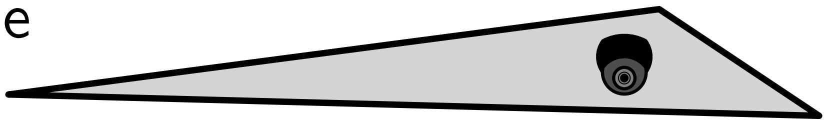
**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

- $P$ could be guarded by $n-2$ cameras placed in the triangles

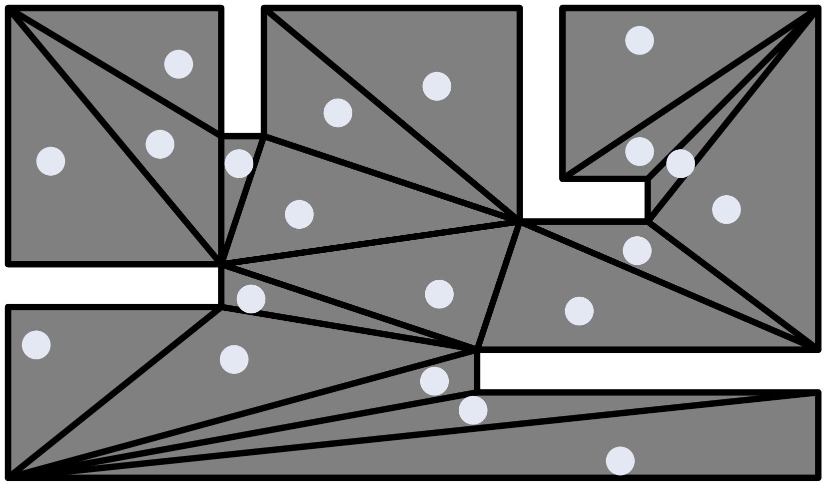
**Can we do better?**

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them

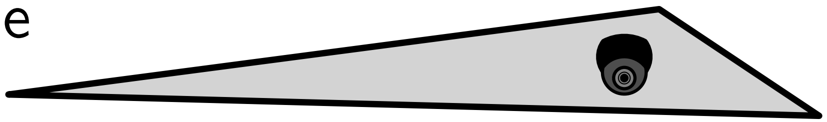


**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

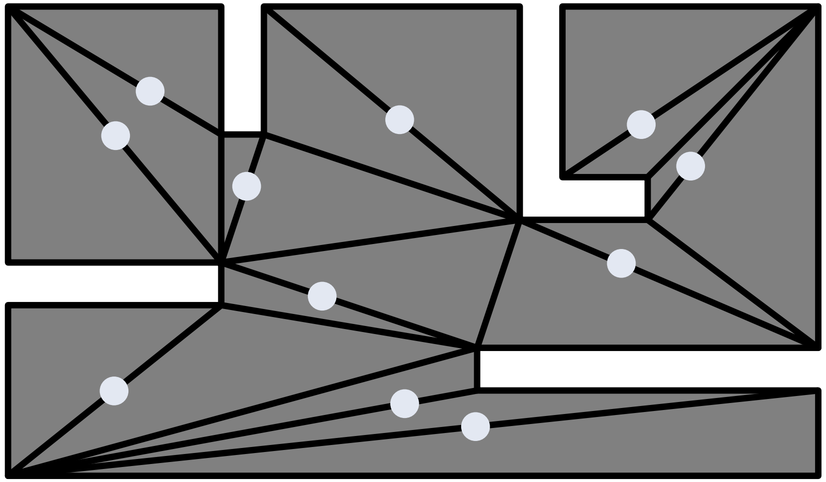
- $P$ could be guarded by $n-2$ cameras placed in the triangles
- $P$ can be guarded by $\approx n/2$ cameras placed on the diagonals

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them

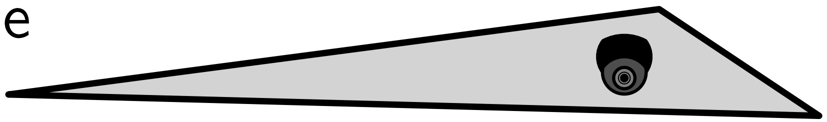


**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

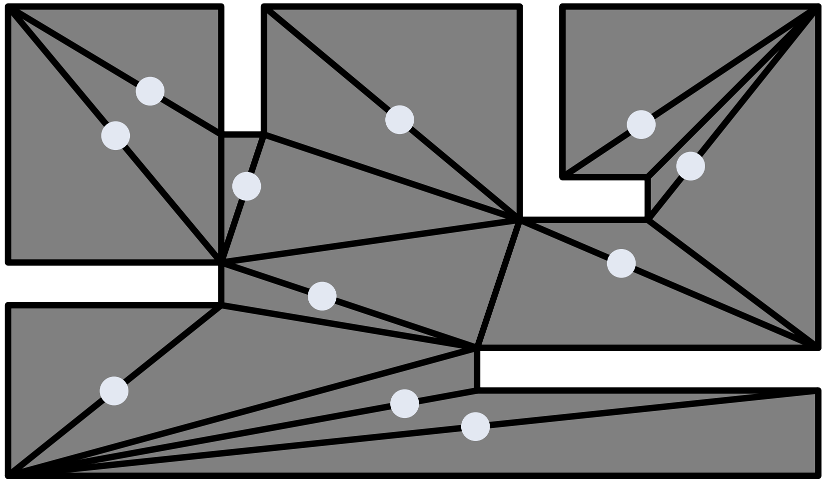
- $P$ could be guarded by $n-2$ cameras placed in the triangles
- $P$ can be guarded by $\approx n/2$ cameras placed on the diagonals

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them

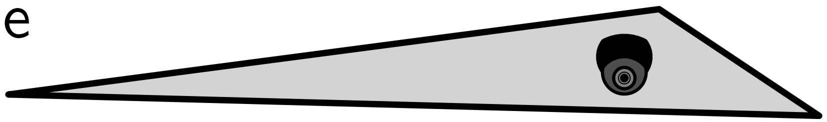


**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

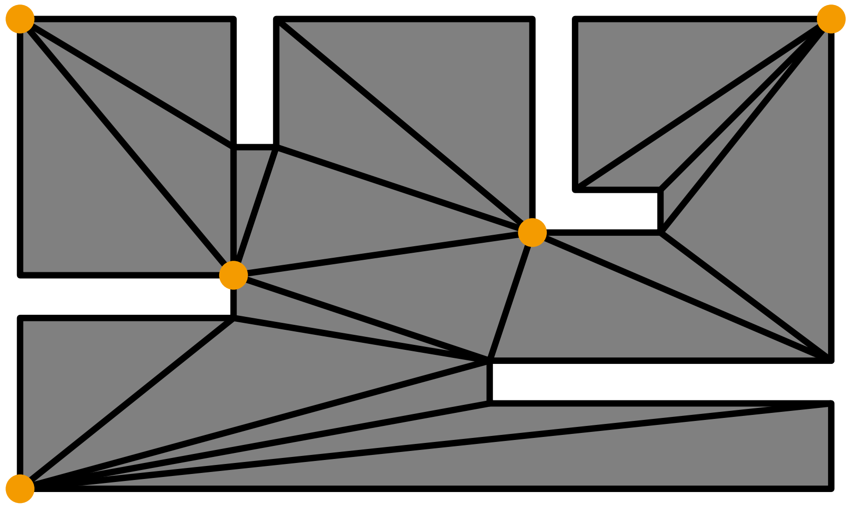
- $P$ could be guarded by $n-2$ cameras placed in the triangles
- $P$ can be guarded by $\approx n/2$ cameras placed on the diagonals
- $P$ can be observed by even smaller number of cameras placed on the corners

# Problem Simplification

**Observation:** It is easy to guard a triangle



**Idea:** Decompose $P$ into triangles and guard each of them



**Theorem 1:** Each simple polygon with $n$ corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

- $P$ could be guarded by $n-2$ cameras placed in the triangles
- $P$ can be guarded by $\approx n/2$ cameras placed on the diagonals
- $P$ can be observed by even smaller number of cameras placed on the corners

# The Art-Gallery-Theorem [Chvátal '75]

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

# The Art-Gallery-Theorem [Chvátal '75]

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

**Proof:**

- Find a simple polygon with $n$ corners that requires $\approx n/3$ cameras!

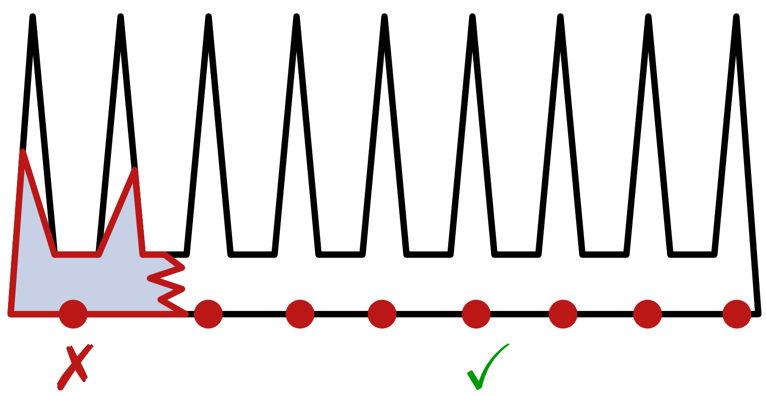
**Discuss it with your neighbour for 2 minutes**

# The Art-Gallery-Theorem [Chvátal '75]

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

**Proof:**

- Find a simple polygon with $n$ corners that requires $\approx n/3$ cameras!

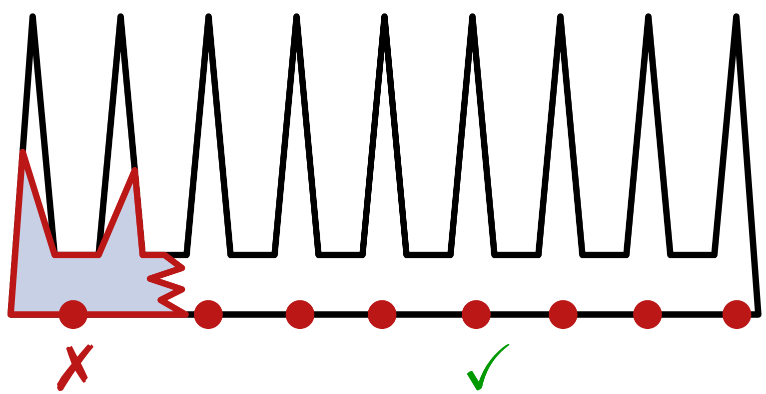


# The Art-Gallery-Theorem [Chvátal '75]

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

**Proof:**

- Find a simple polygon with $n$ corners that requires $\approx n/3$ cameras!



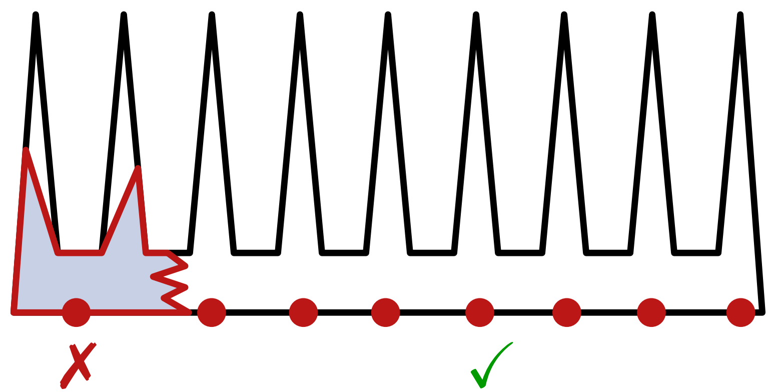

- Sufficiency on the board

# The Art-Gallery-Theorem [Chvátal '75]

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

**Proof:**

- Find a simple polygon with $n$ corners that requires $\approx n/3$ cameras!




- Sufficiency on the board

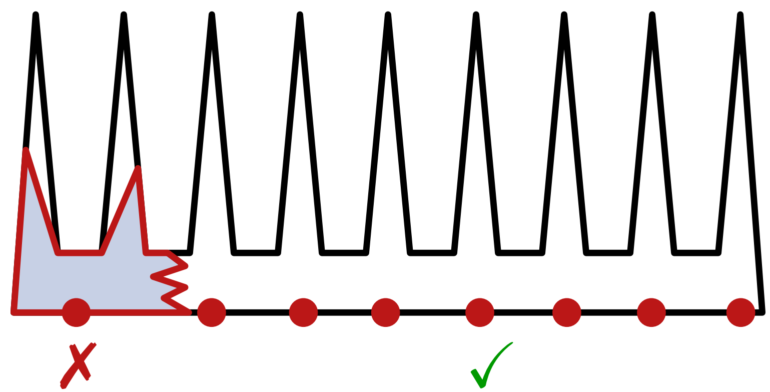
**Conclusion:** Given a triangulation, the $\lfloor n/3 \rfloor$ cameras that guard the polygon can be placed in $O(n)$ time.

# The Art-Gallery-Theorem [Chvátal '75]

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

**Proof:**

- Find a simple polygon with $n$ corners that requires $\approx n/3$ cameras!




- Sufficiency on the board

**Conclusion:** Given a triangulation, the $\lfloor n/3 \rfloor$ cameras that guard the polygon can be placed in $O(n)$ time.

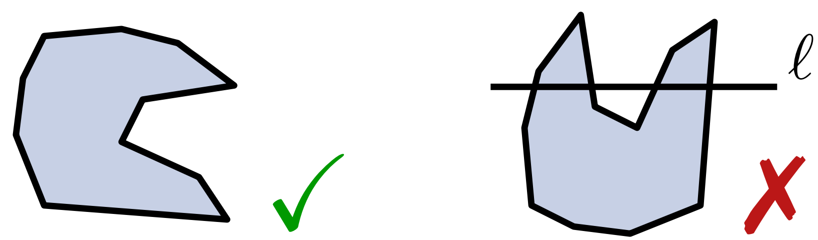
**Can we do better than $O(n^2)$ described before?**

# Proof of Art-Gallery-Theorem: Overview

3-step process:

- Step 1: Decompose $P$ into $y$-monotone polygons

**Definition:** A polygon is $y$-monotone, if for any horizontal line $\ell$, the interection $\ell \cap P$ is connected.



# Proof of Art-Gallery-Theorem: Overview

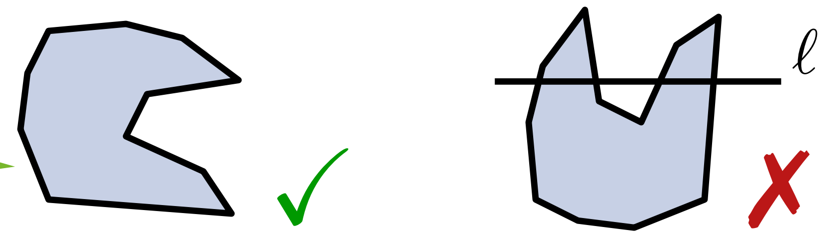


3-step process:

- Step 1: Decompose $P$ into $y$-monotone polygons

**Definition:** A polygon is $y$-monotone, if for any horizontal line $\ell$, the interection $\ell \cap P$ is connected.

The two paths from the topmost to the bottomost point bounding the polygon, never go upward



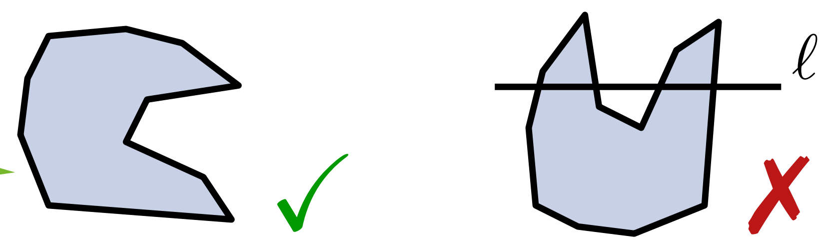
# Proof of Art-Gallery-Theorem: Overview

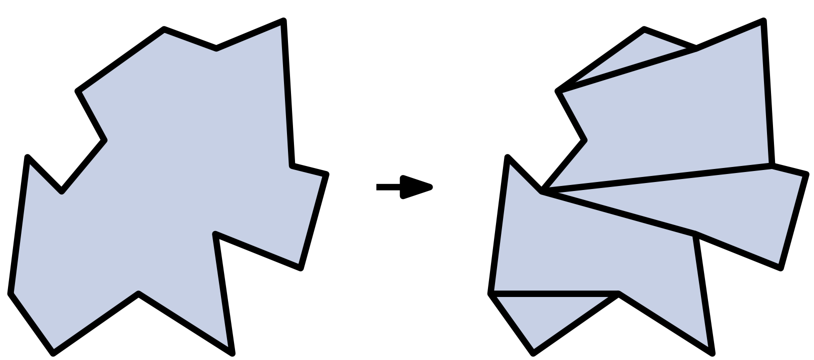
3-step process:

- Step 1: Decompose $P$ into $y$-monotone polygons

**Definition:** A polygon is $y$-monotone, if for any horizontal line $\ell$, the interection $\ell \cap P$ is connected.

The two paths from the topmost to the bottomost point bounding the polygon, never go upward



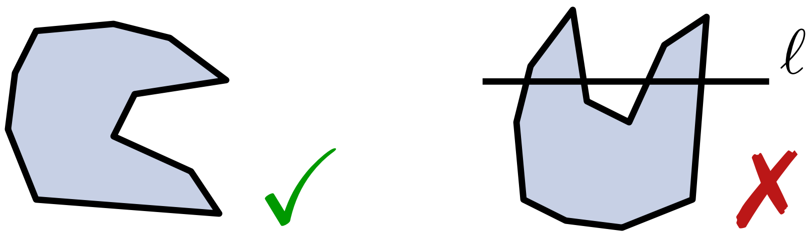



# Proof of Art-Gallery-Theorem: Overview

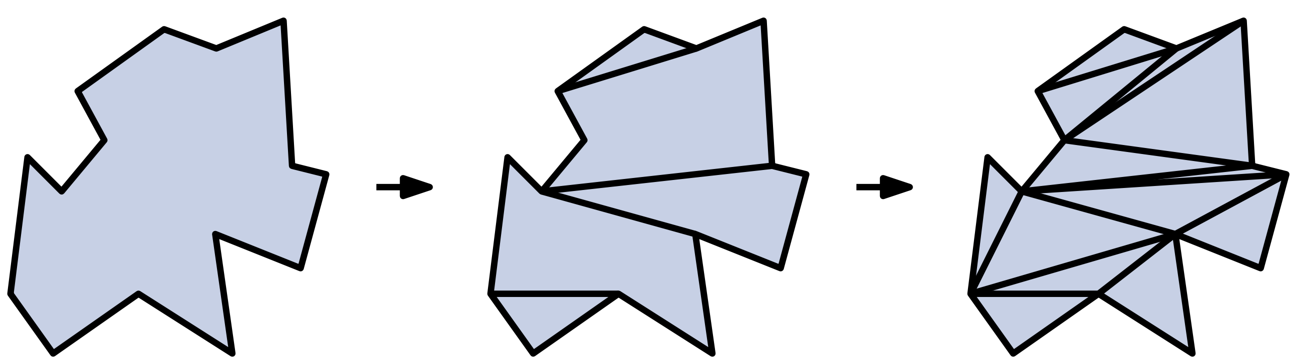
3-step process:

- Step 1: Decompose $P$ into $y$-monotone polygons

  **Definition:** A polygon is $y$-monotone, if for any horizontal line $\ell$, the interection $\ell \cap P$ is connected.




- Step 2: Triangulate the resulting $y$-monotone polygons

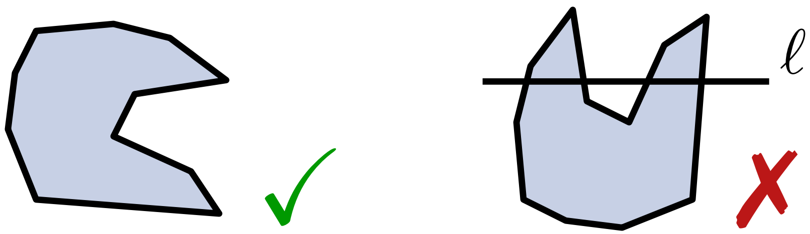


# Proof of Art-Gallery-Theorem: Overview

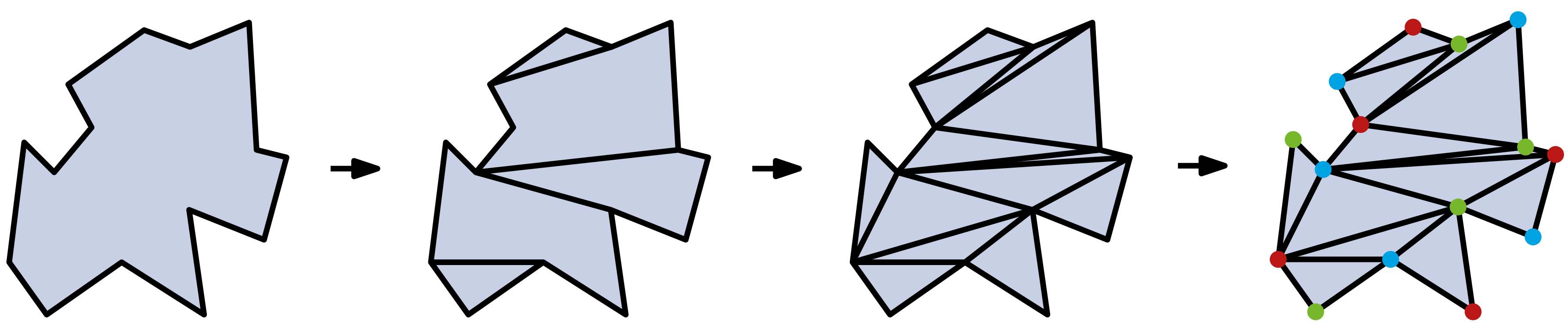
3-step process:

- Step 1: Decompose $P$ into $y$-monotone polygons

**Definition:** A polygon is $y$-monotone, if for any horizontal line $\ell$, the interection $\ell \cap P$ is connected.

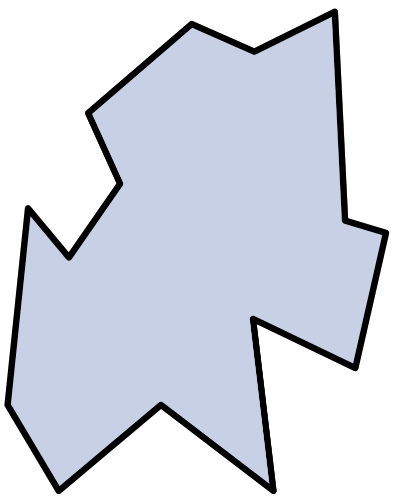



- Step 2: Triangulate the resulting $y$-monotone polygons
- Step 3: use DFS to color the vertices of the polygon



# Partition into $y$-monotone Polygons

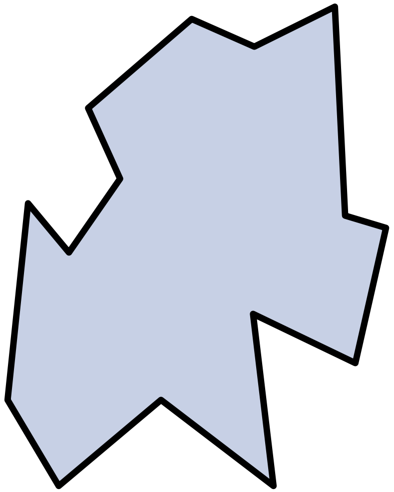
**Idea:** Five different types of vertices



# Partition into $y$-monotone Polygons

**Idea:** Five different types of vertices

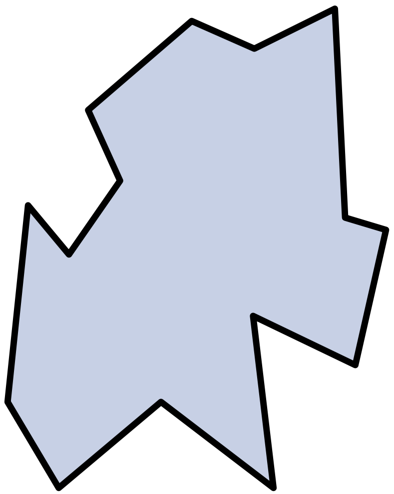
– *Turn vertices*:



# Partition into $y$-monotone Polygons

**Idea:** Five different types of vertices

– *Turn vertices*:
vertical change in direction

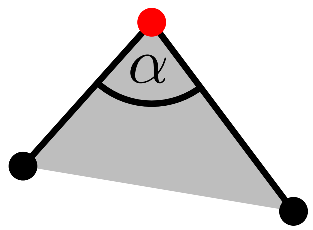


# Partition into $y$-monotone Polygons

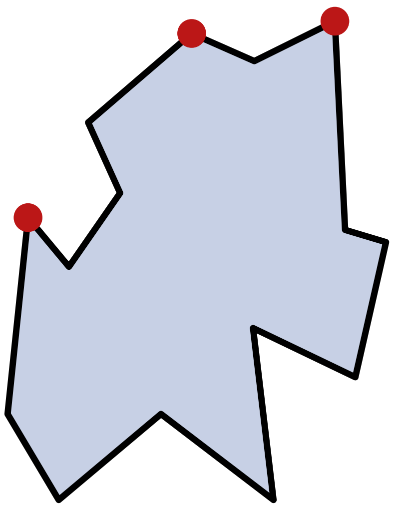
**Idea:** Five different types of vertices

– *Turn vertices*:
vertical change in direction

- *start* vertices



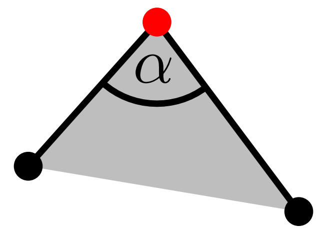

if $\alpha < 180°$



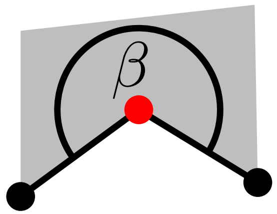
# Partition into $y$-monotone Polygons

**Idea:** Five different types of vertices

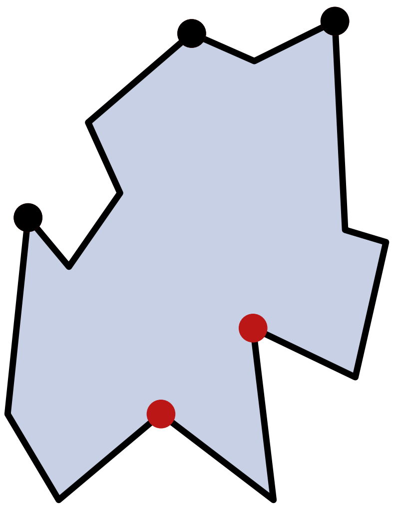
– *Turn vertices*:
vertical change in direction

- *start* vertices  if $\alpha < 180°$



- *split* vertices  if $\beta > 180°$



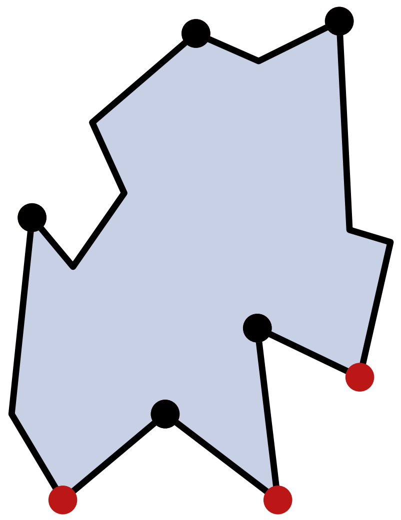


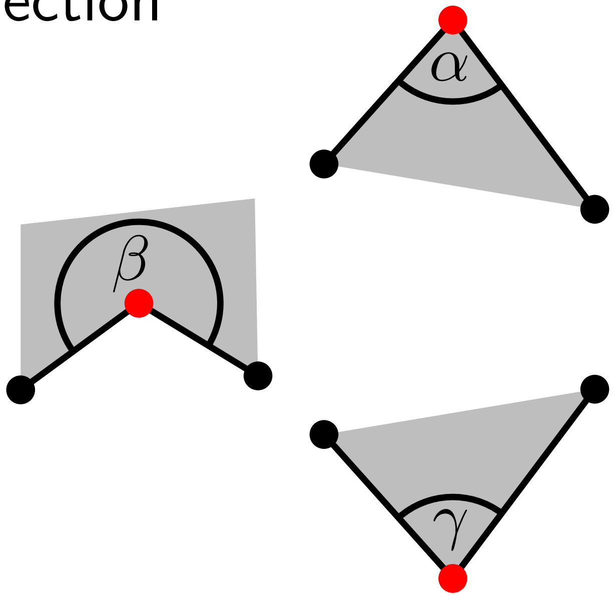
# Partition into $y$-monotone Polygons

**Idea:** Five different types of vertices

– *Turn vertices*:
vertical change in direction

- *start* vertices
- *split* vertices
- *end* vertices






if $\alpha < 180^\circ$

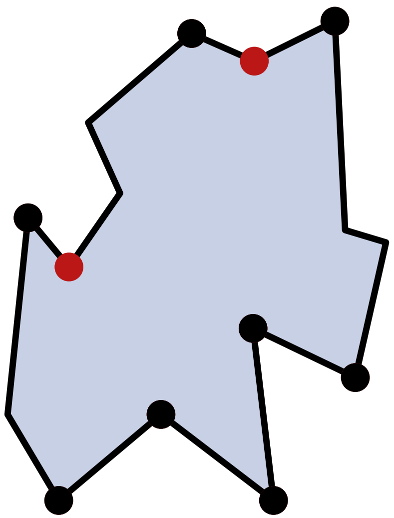
if $\beta > 180^\circ$

if $\gamma < 180^\circ$

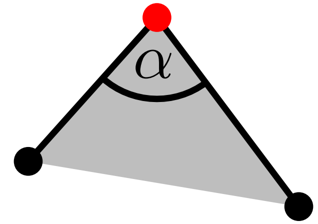
# Partition into $y$-monotone Polygons

**Idea:** Five different types of vertices

– *Turn vertices*:
vertical change in direction

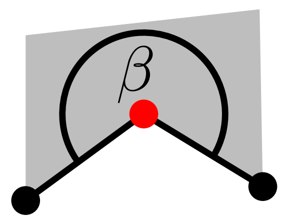


- *start* vertices



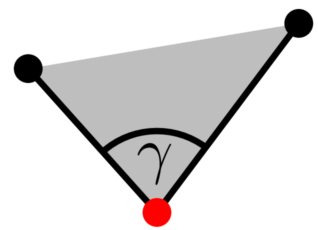

if $\alpha < 180°$

- *split* vertices



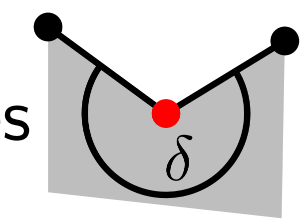

if $\beta > 180°$

- *end* vertices




if $\gamma < 180°$

- *merge* vertices

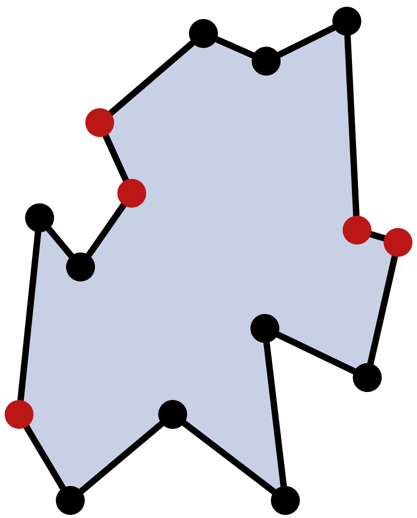



if $\delta > 180°$

# Partition into $y$-monotone Polygons

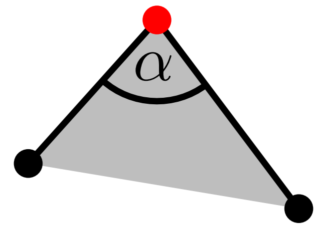


**Idea:** Five different types of vertices



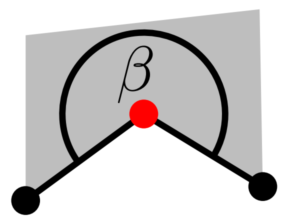
– *Turn vertices*:
vertical change in direction

- *start* vertices



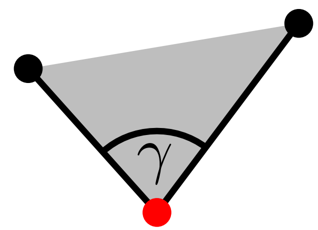

if $\alpha < 180°$

- *split* vertices



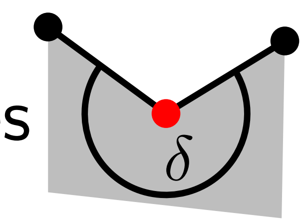

if $\beta > 180°$

- *end* vertices



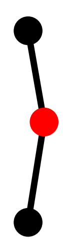

if $\gamma < 180°$

- *merge* vertices




if $\delta > 180°$

– *regular* vertices



# Characterization

**Lemma 1:** A polygon is $y$-monotone if it has no split vertices or merge vertices.

# Characterization

**Lemma 1:** A polygon is $y$-monotone if it has no split vertices or merge vertices.

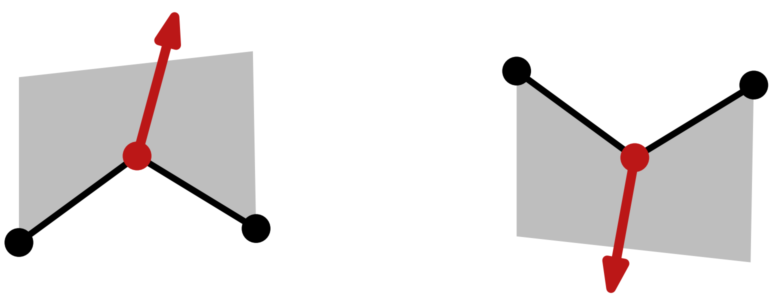
**Proof:** On the blackboard

# Characterization

**Lemma 1:** A polygon is $y$-monotone if it has no split vertices or merge vertices.

**Proof:** On the blackboard

$\Rightarrow$ We need to eliminate all split and merge vertices by using diagonals

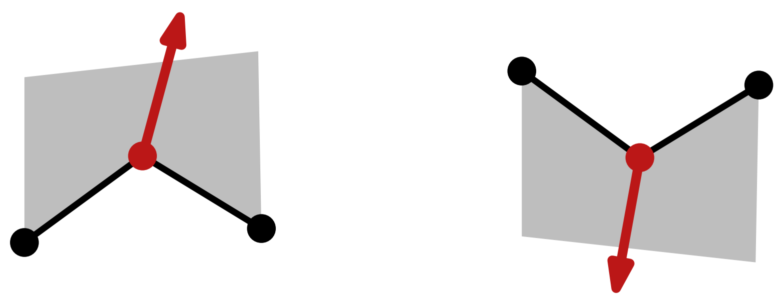


# Characterization

**Lemma 1:** A polygon is $y$-monotone if it has no split vertices or merge vertices.

**Proof:** On the blackboard

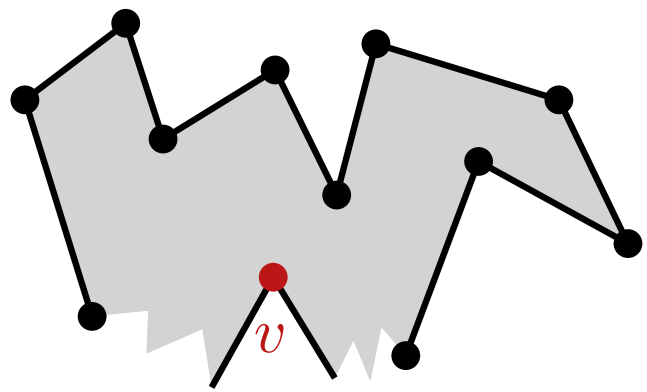
$\Rightarrow$ We need to eliminate all split and merge vertices by using diagonals



**Observation:** The diagonals should neither cross the edges of $P$ nor the other diagonals

# Ideas for Sweep-Line-Algorithm

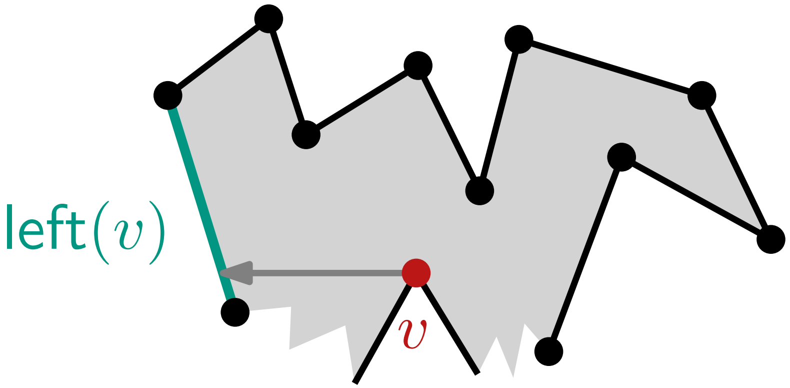
## 1) Diagonals for the split vertices



# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices

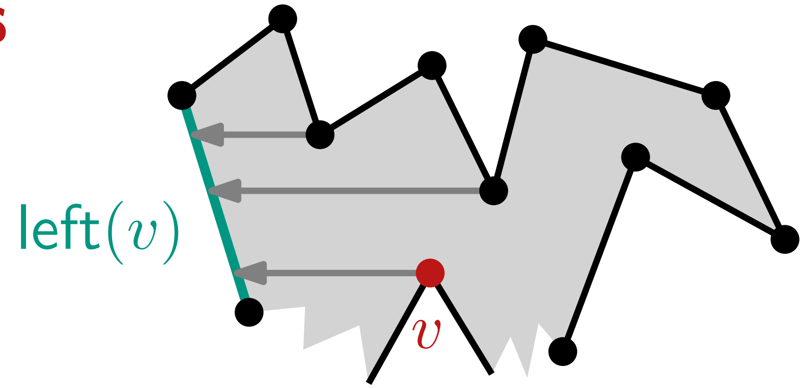
- compute for each vertex $v$ its left adjacent edge left($v$) with respect to the horizontal sweep line $\ell$



# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices

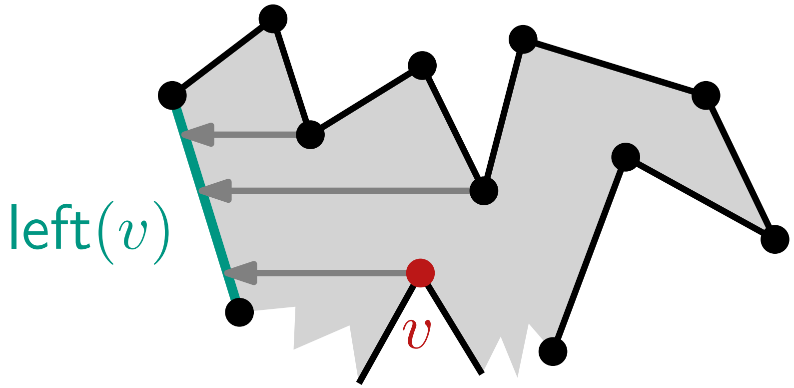
- compute for each vertex $v$ its left adjacent edge left$(v)$ with respect to the horizontal sweep line $\ell$



# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices

- compute for each vertex $v$ its left adjacent edge left$(v)$ with respect to the horizontal sweep line $\ell$

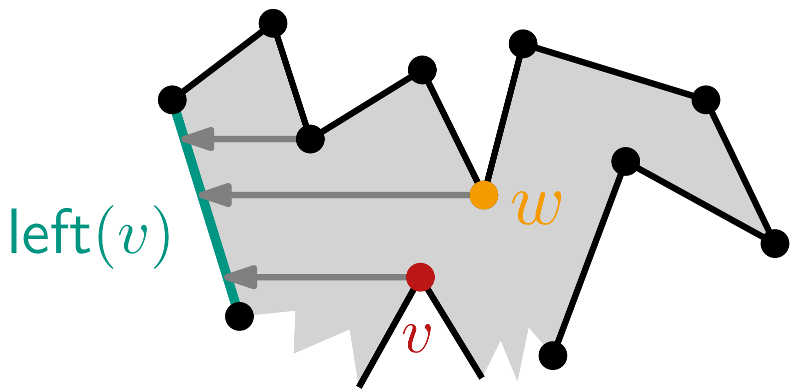



- connect split vertex $v$ to the nearest vertex $w$ above $v$, such that left$(w) =$ left$(v)$

# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices

- compute for each vertex $v$ its left adjacent edge left$(v)$ with respect to the horizontal sweep line $\ell$

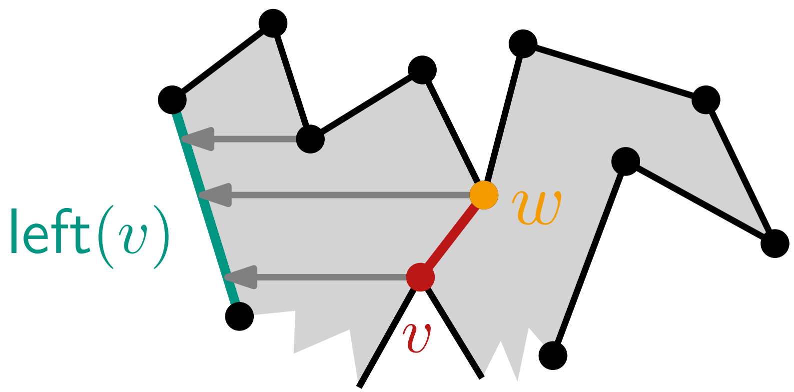



- connect split vertex $v$ to the nearest vertex $w$ above $v$, such that left$(w) =$ left$(v)$

# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices

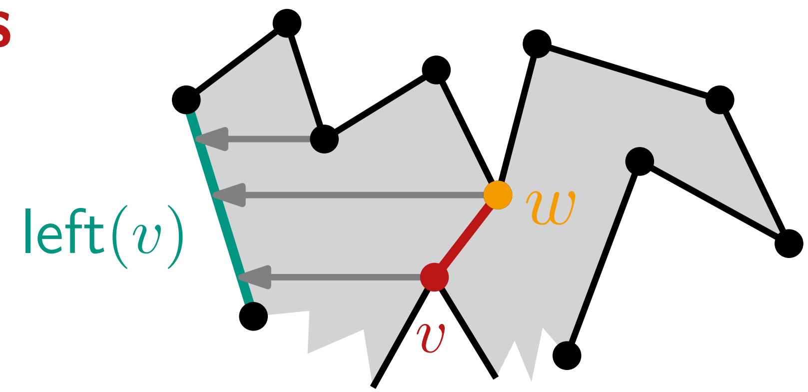
- compute for each vertex $v$ its left adjacent edge left$(v)$ with respect to the horizontal sweep line $\ell$



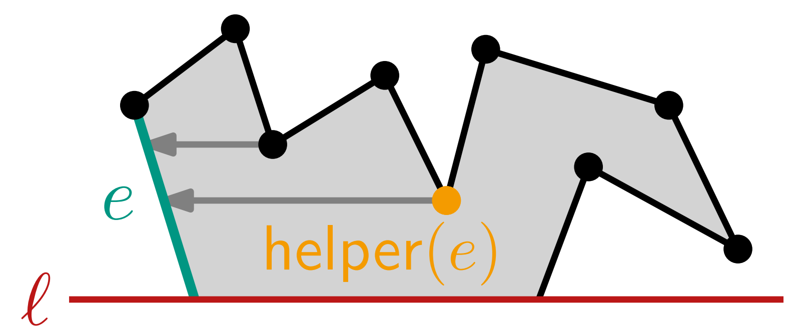

- connect split vertex $v$ to the nearest vertex $w$ above $v$, such that left$(w) =$ left$(v)$

# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices



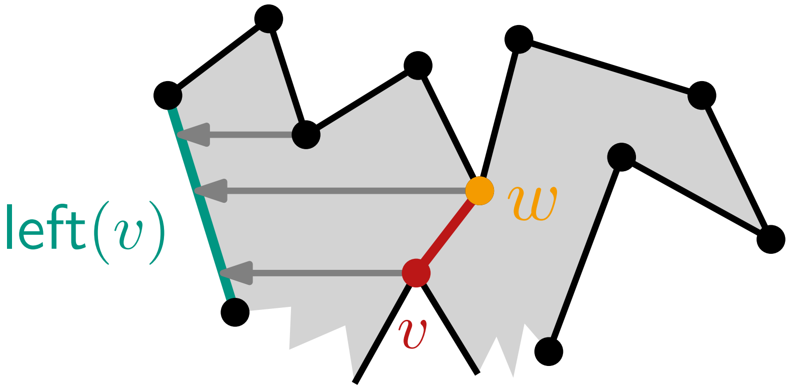

- compute for each vertex $v$ its left adjacent edge left$(v)$ with respect to the horizontal sweep line $\ell$
- connect split vertex $v$ to the nearest vertex $w$ above $v$, such that left$(w) =$ left$(v)$
- for each edge $e$ save the botommost vertex $w$ such that left$(w) = e$; notation helper$(e) :=$ w



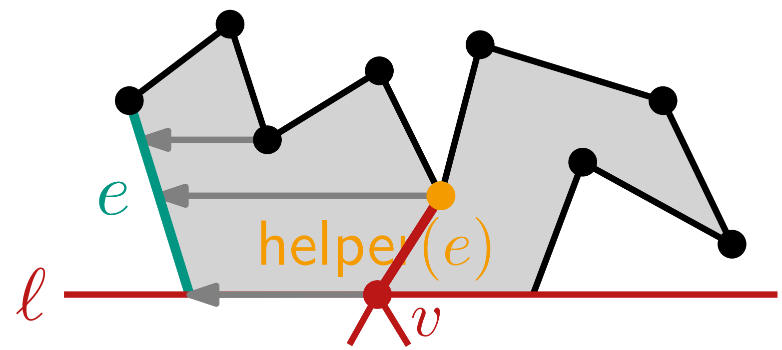
# Ideas for Sweep-Line-Algorithm

## 1) Diagonals for the split vertices

- compute for each vertex $v$ its left adjacent edge left$(v)$ with respect to the horizontal sweep line $\ell$



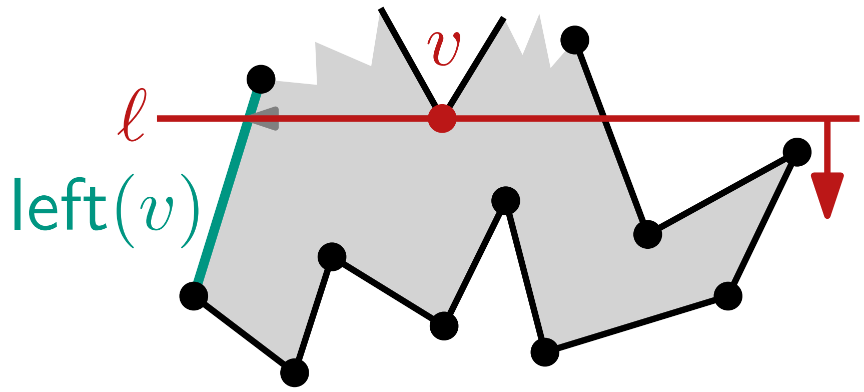

- connect split vertex $v$ to the nearest vertex $w$ above $v$, such that left$(w)$ = left$(v)$
- for each edge $e$ save the botommost vertex $w$ such that left$(w) = e$; notation helper$(e)$ := w
- when $\ell$ passes through a split vertex $v$, we connect $v$ with helper(left$(v)$)



# Ideas for Sweep-Line-Algorithm

## 2) Diagonals for merge vertices

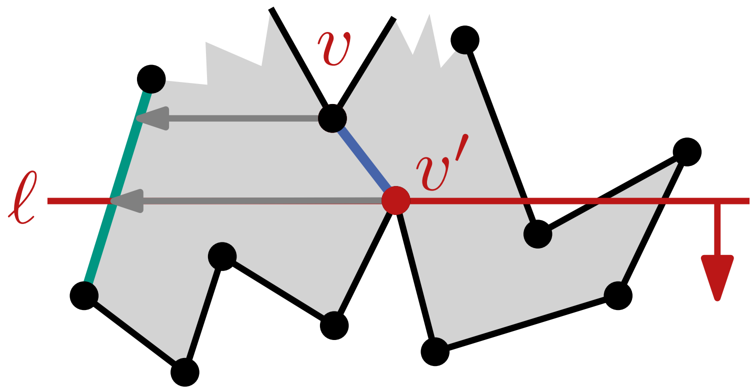
- when the vertex $v$ is reached, we set helper(left$(v)) = v$



# Ideas for Sweep-Line-Algorithm

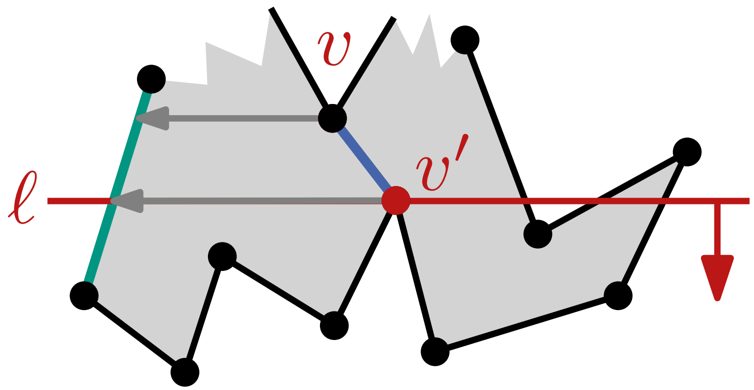
## 2) Diagonals for merge vertices

- when the vertex $v$ is reached, we set helper(left$(v)) = v$
- when we reach a split vertex $v'$ such that left$(v')$ = left$(v)$ the diagonal $(v, v')$ is introduced

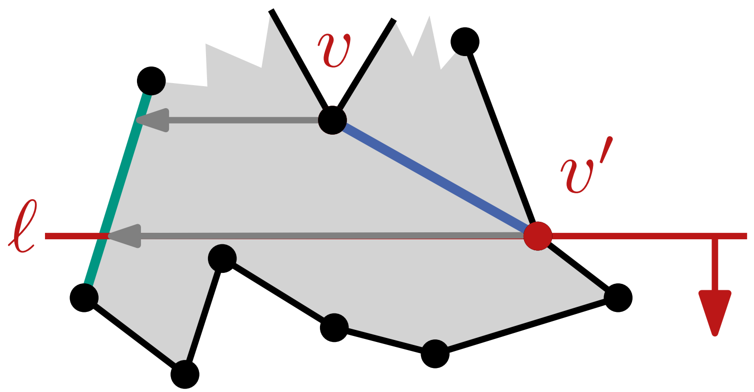


## 2) Diagonals for merge vertices

- when the vertex $v$ is reached, we set helper(left$(v)) = v$
- when we reach a split vertex $v'$ such that left$(v')$ = left$(v)$ the diagonal $(v, v')$ is introduced




- in case we reach a regular vertex $v'$ such that helper(left$(v')$) is $v$ the diagonal $(v, v')$ is introduced

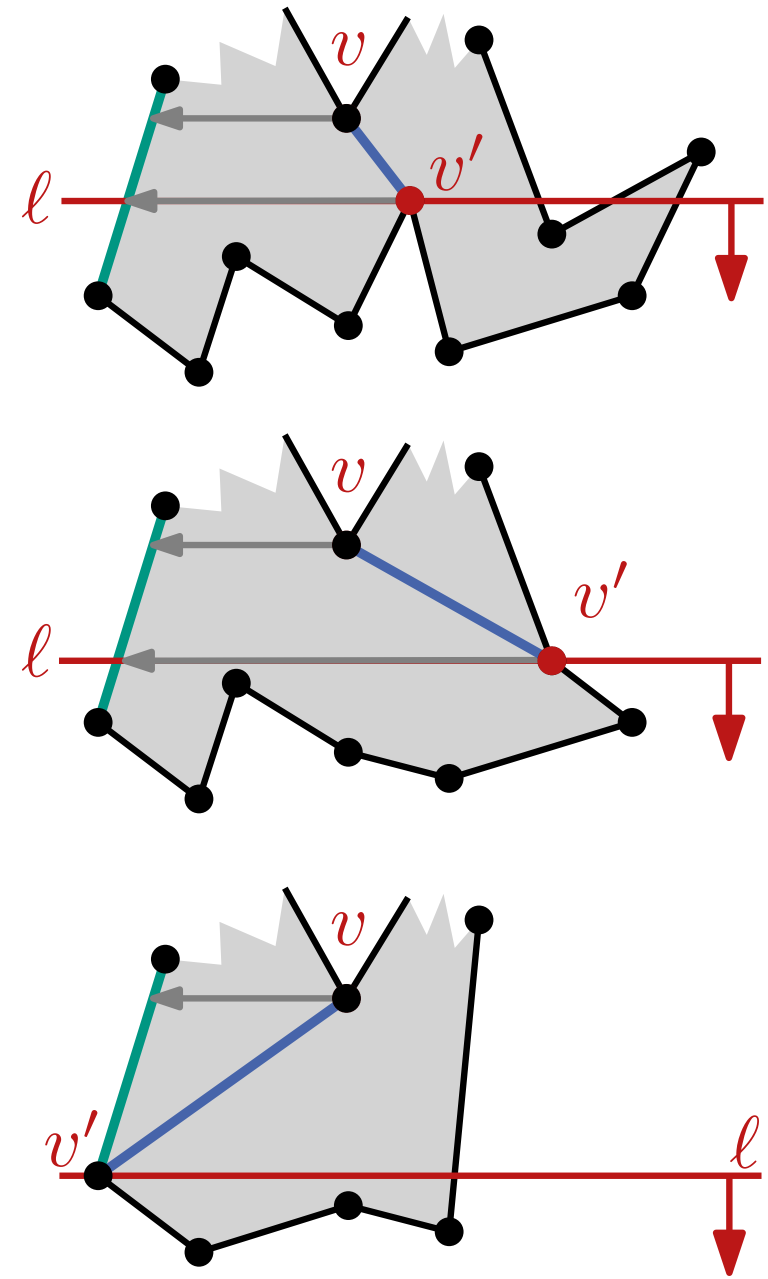


# Ideas for Sweep-Line-Algorithm



## 2) Diagonals for merge vertices

- when the vertex $v$ is reached, we set helper(left$(v)) = v$
- when we reach a split vertex $v'$ such that left$(v')$ = left$(v)$ the diagonal $(v, v')$ is introduced
- in case we reach a regular vertex $v'$ such that helper(left$(v')$) is $v$ the diagonal $(v, v')$ is introduced
- if the end of $v'$ of left$(v)$ is reached, then the diagonal $(v, v')$ is introduced



# Algorithm MakeMonotone(P)

**MakeMonotone**(Polygon $P$)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$ (binary search tree for sweep-line status)
**while** $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex($v$)
  handleVertex($v$)
**return** $\mathcal{D}$

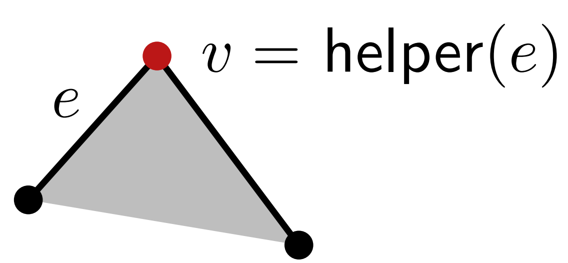
# Algorithm MakeMonotone(P)

**MakeMonotone**(Polygon $P$)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$ (binary search tree for sweep-line status)
**while** $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex($v$)
  handleVertex($v$)
**return** $\mathcal{D}$

**handleStartVertex**(vertex $v$)

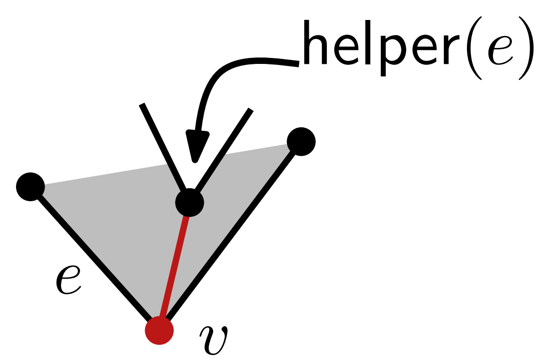
$\mathcal{T} \leftarrow$ add the left edge $e$
helper($e$) $\leftarrow v$



# Algorithm MakeMonotone(P)

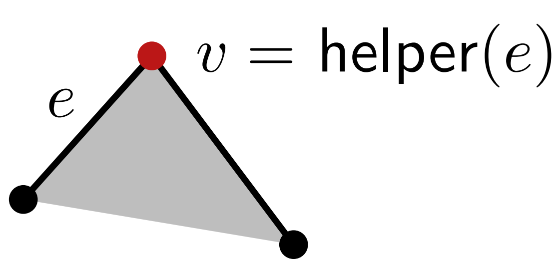
**MakeMonotone**(Polygon $P$)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$ (binary search tree for sweep-line status)
**while** $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex($v$)
  handleVertex($v$)
**return** $\mathcal{D}$




**handleStartVertex**(vertex $v$)

$\mathcal{T} \leftarrow$ add the left edge $e$
helper($e$) $\leftarrow v$




**handleEndVertex**(vertex $v$)

$e \leftarrow$ left edge
**if** isMergeVertex(helper($e$)) **then**
  $\mathcal{D} \leftarrow$ add edge (helper($e$), $v$)
remove $e$ from $\mathcal{T}$

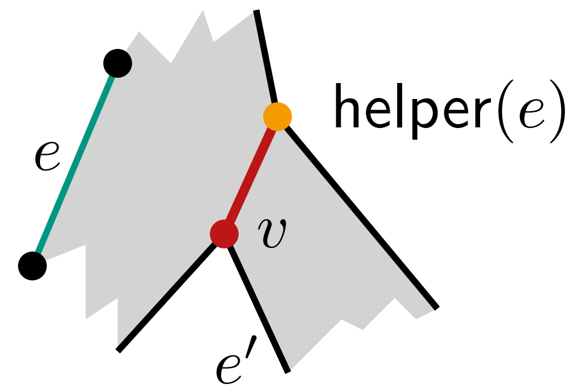
# Algorithm MakeMonotone(P)

**MakeMonotone**(Polygon $P$)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$ (binary search tree for sweep-line status)
**while** $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex($v$)
  handleVertex($v$)
**return** $\mathcal{D}$

**handleSplitVertex**(vertex $v$)

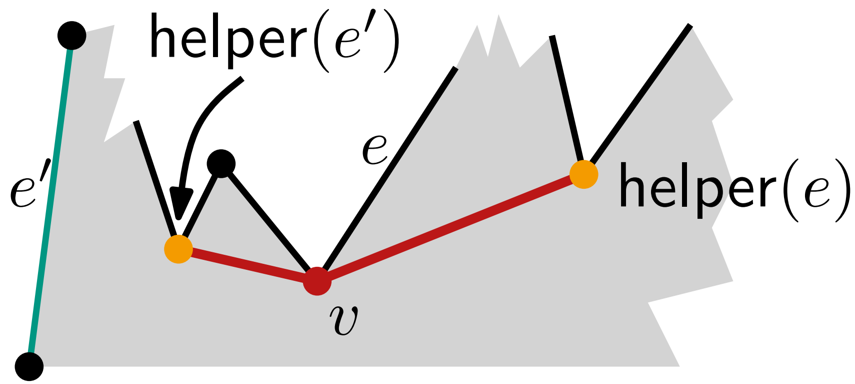
$e \leftarrow$ Edge to the left of $v$ in $\mathcal{T}$
$\mathcal{D} \leftarrow$ add edge (helper($e$), $v$)
helper($e$) $\leftarrow v$
$\mathcal{T} \leftarrow$ add the right edge $e'$ of $v$
helper($e'$) $\leftarrow v$



# Algorithm MakeMonotone(P)

**MakeMonotone**(Polygon $P$)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$ (binary search tree for sweep-line status)
**while** $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex($v$)
  handleVertex($v$)
**return** $\mathcal{D}$



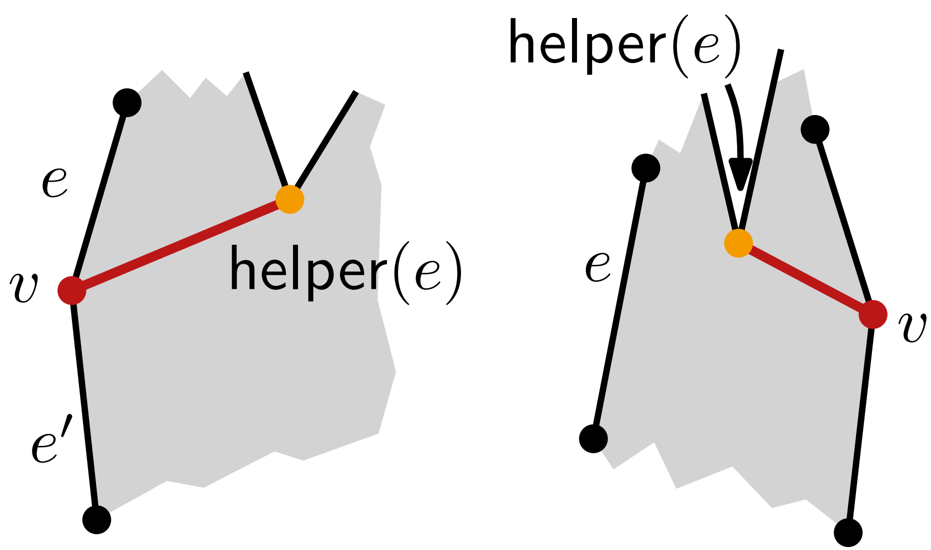

**handleMergeVertex**(vertex $v$)

$e \leftarrow$ right edge
**if** isMergeVertex(helper($e$)) **then**
  $\mathcal{D} \leftarrow$ add edge (helper($e$), $v$)
remove $e$ from $\mathcal{T}$
$e' \leftarrow$ edge to the left of $v$ in $\mathcal{T}$
**if** isMergeVertex(helper($e'$)) **then**
  $\mathcal{D} \leftarrow$ add edge (helper($e'$), $v$)
helper($e'$) $\leftarrow v$

# Algorithm MakeMonotone(P)

**MakeMonotone**(Polygon $P$)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$ (binary search tree for sweep-line status)
**while** $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex($v$)
  handleVertex($v$)
**return** $\mathcal{D}$




**handleRegularVertex**(vertex $v$)

**if** $P$ lies locally to the right of $v$ **then**
  $e, e' \leftarrow$ above, below edge
  **if** isMergeVertex(helper($e$)) **then**
    $\mathcal{D} \leftarrow$ add edge (helper($e$), $v$)
  remove $e$ from $\mathcal{T}$
  $\mathcal{T} \leftarrow$ add $e'$; helper($e'$) $\leftarrow v$
**else**
  $e \leftarrow$ edge to the left of $v$
  add $e$ to $\mathcal{T}$
  **if** isMergeVertex(helper($e$)) **then**
    $\mathcal{D} \leftarrow$ add (helper($e$), $v$)
  helper($e$) $\leftarrow v$

# Analysis

**Lemma 2:** The algorithm MakeMonotone computes a set of crossing-free diagonals of $P$, which partitions $P$ into $y$-monotone polygons.

# Analysis

**Lemma 2:** The algorithm MakeMonotone computes a set of crossing-free diagonals of $P$, which partitions $P$ into $y$-monotone polygons.

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

# Analysis

**Lemma 2:** The algorithm MakeMonotone computes a set of crossing-free diagonals of $P$, which partitions $P$ into $y$-monotone polygons.

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

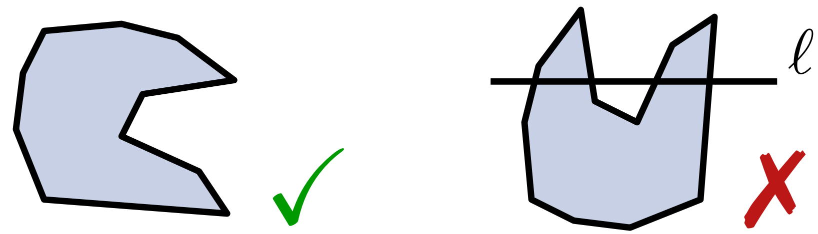
- Construct priority queue $\mathcal{Q}$: $O(n)$
- Initialize sweep-line status $\mathcal{T}$: $O(1)$
- Handle a single event: $O(\log n)$
  - $\mathcal{Q}$.deleteMax: $O(\log n)$
  - Find, remove, add element in $\mathcal{T}$: $O(\log n)$
  - Add diagonals to $\mathcal{D}$: $O(1)$
- Space: obviously $O(n)$

# Proof of Art-Gallery-Theorem: Overview

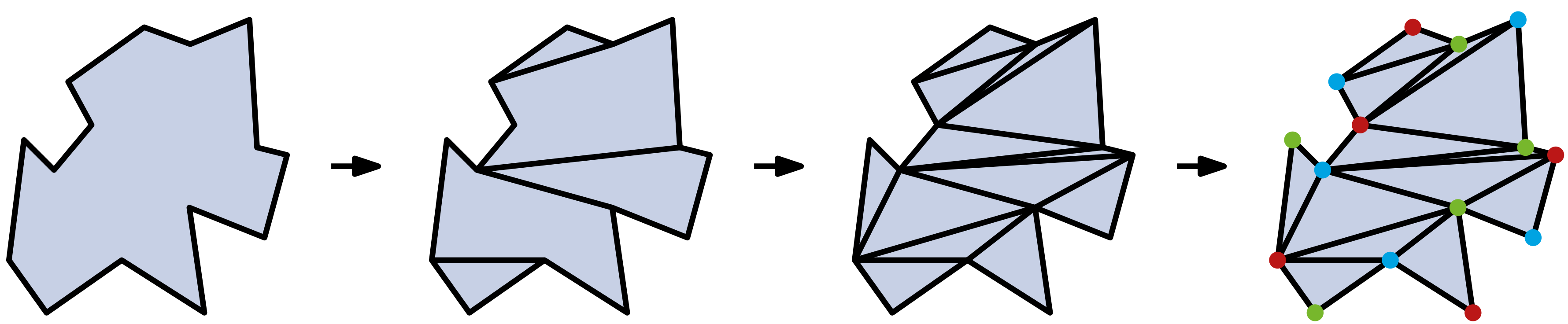
Three-step procedure:

- Step 1: Decompose $P$ in $y$-monotone polygons ✓

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



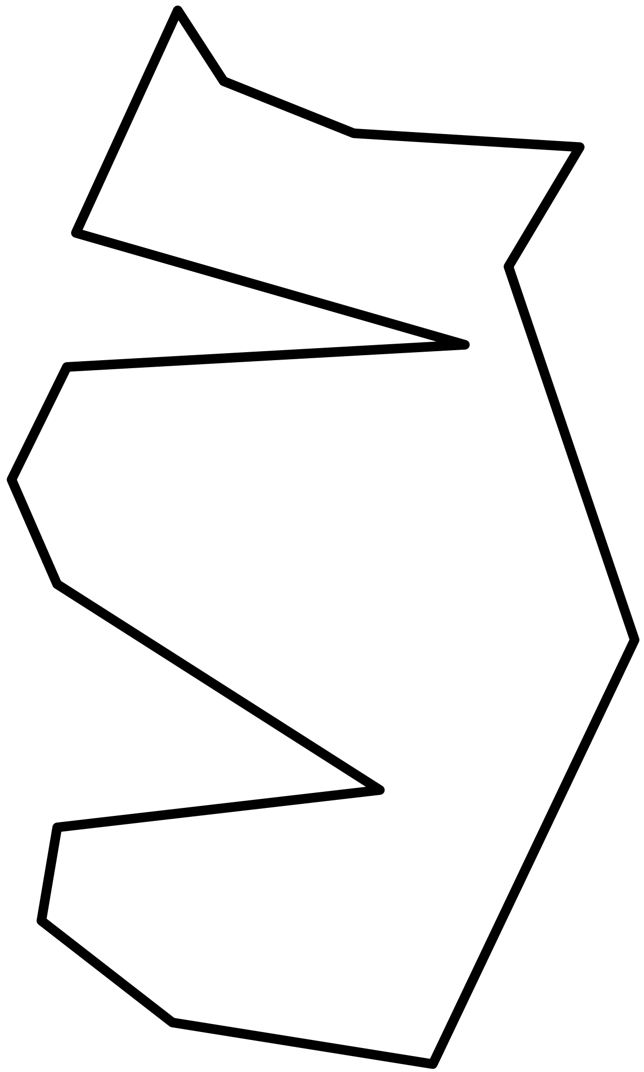

- Step 2: Triangulate $y$-monotone polygons **ToDo!**
- Step 3: use DFS to color the triangulated polygon ✓



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

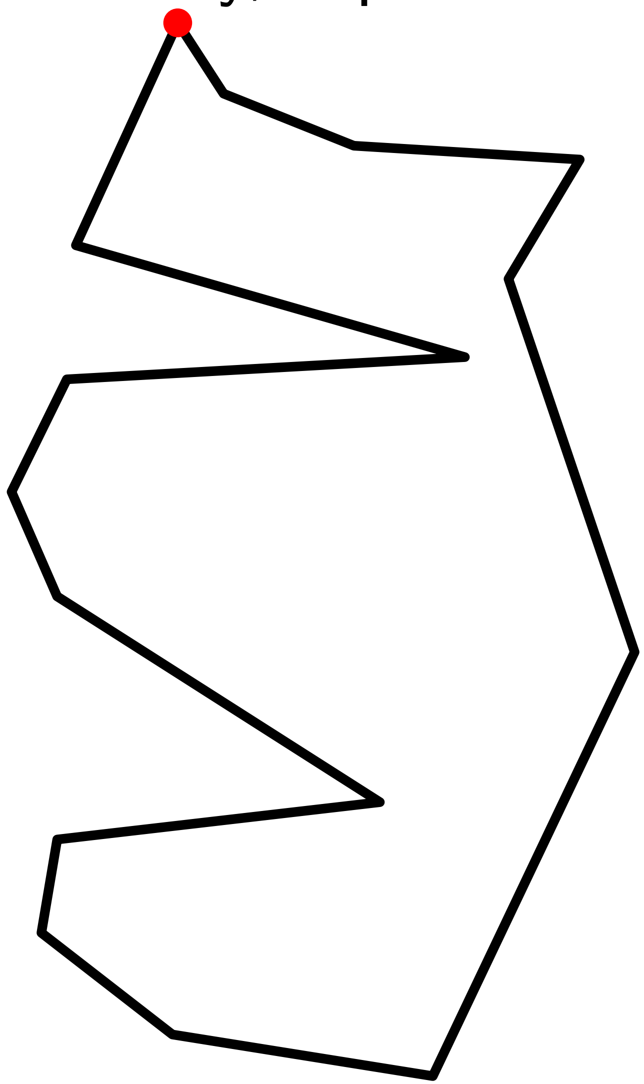
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

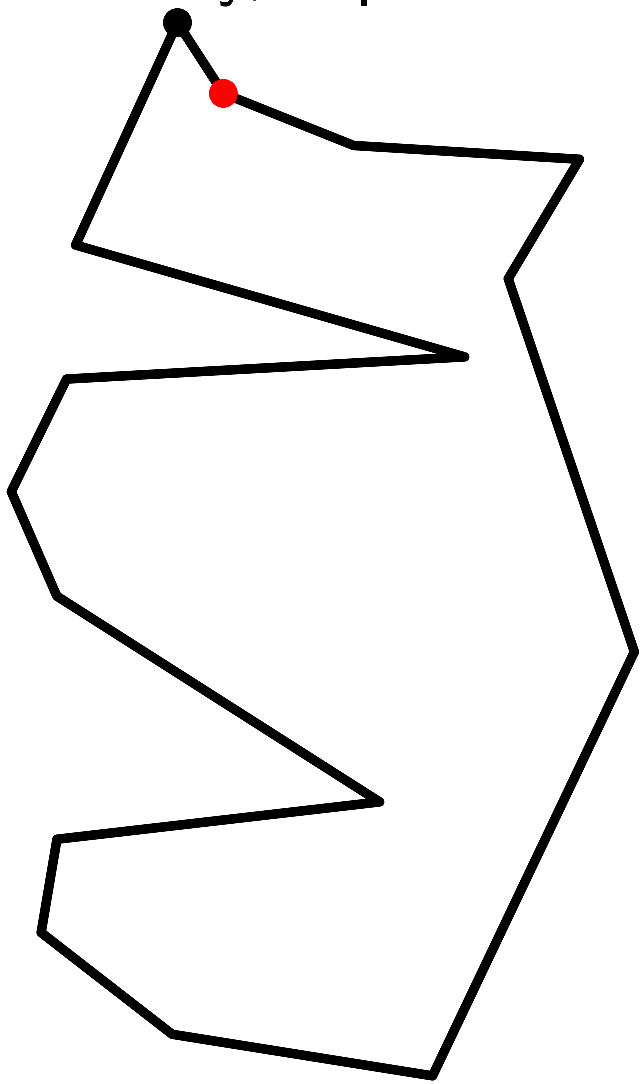
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

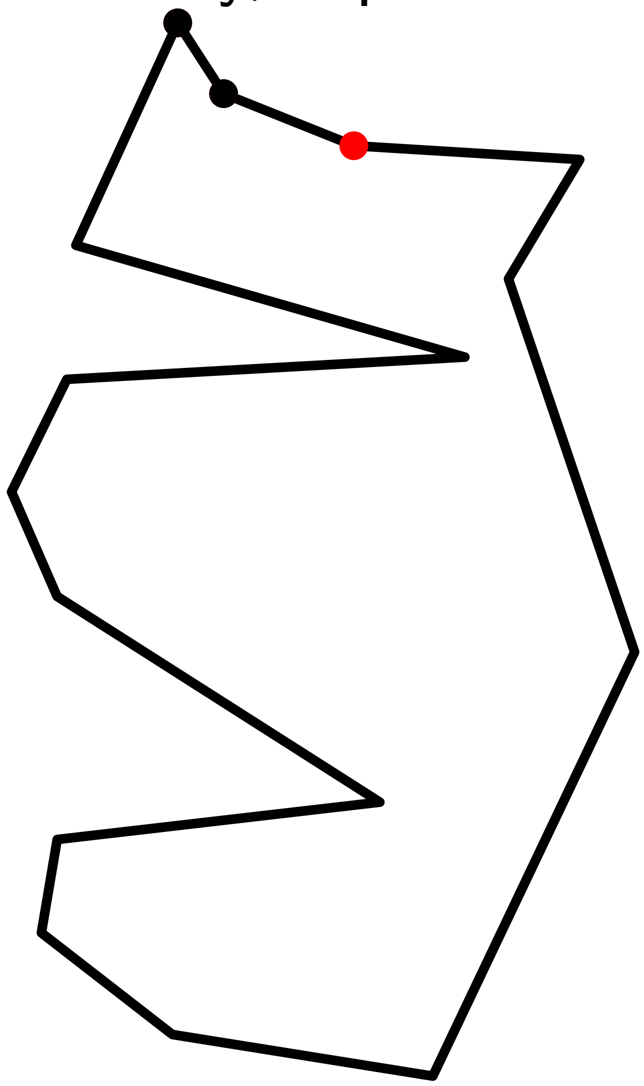
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

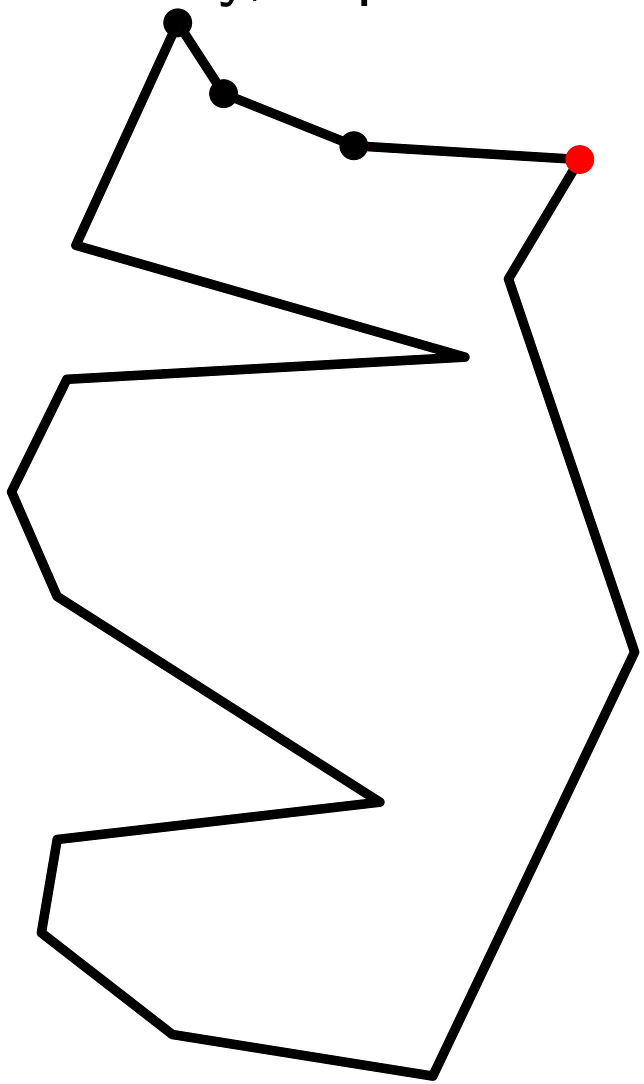
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

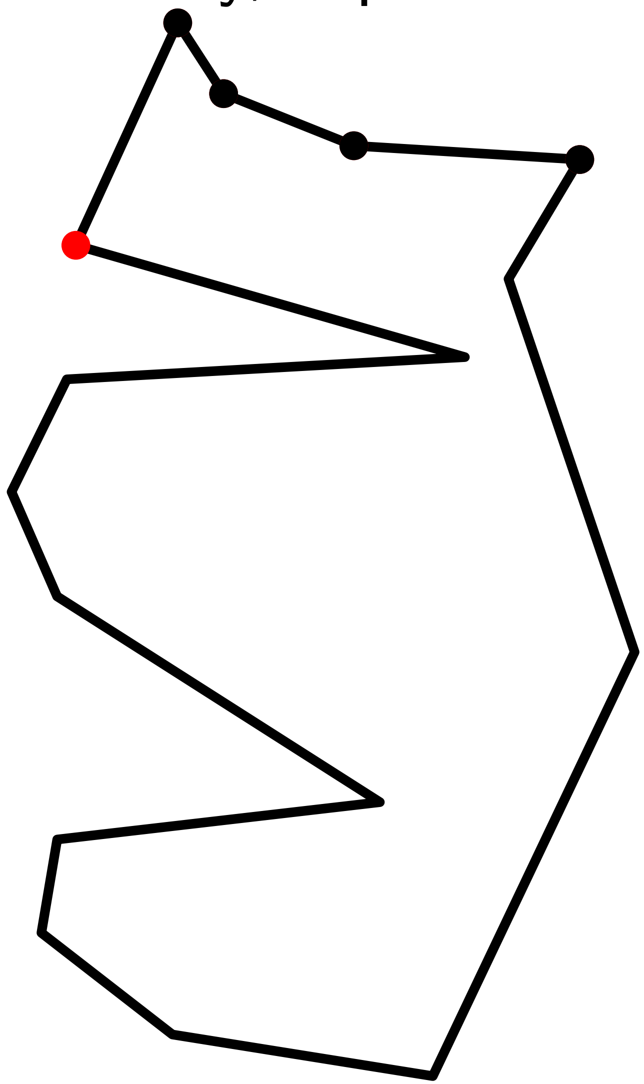
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

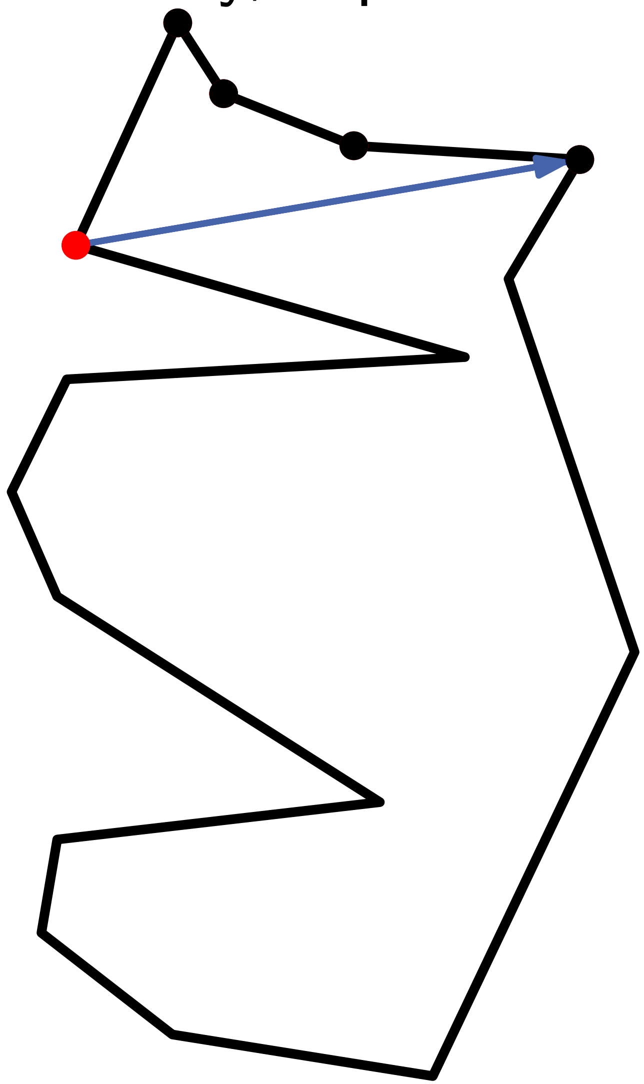
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

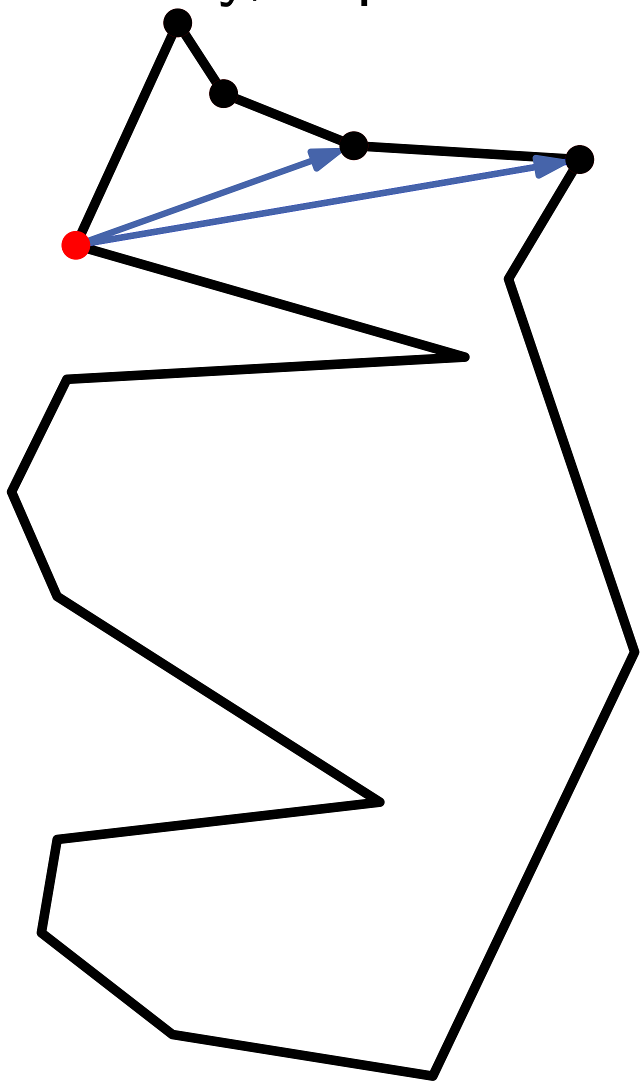
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

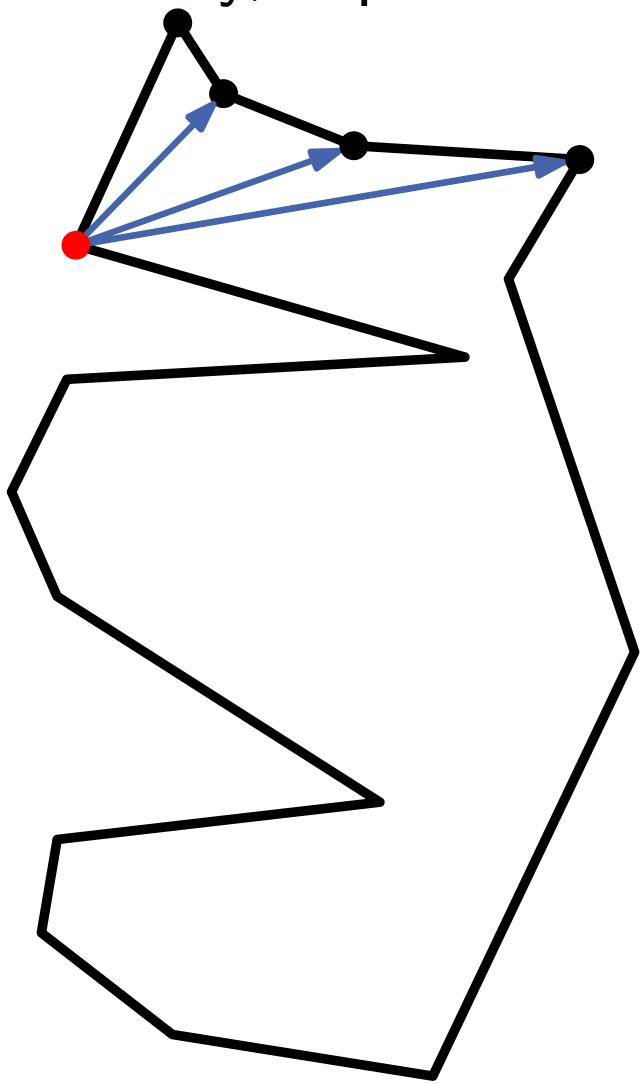
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

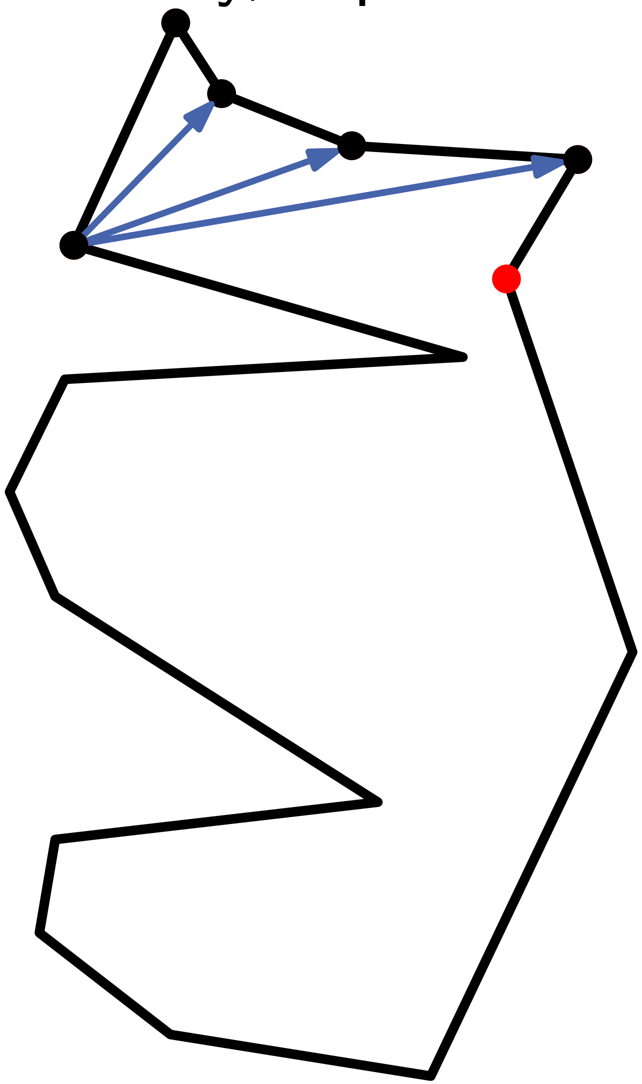
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

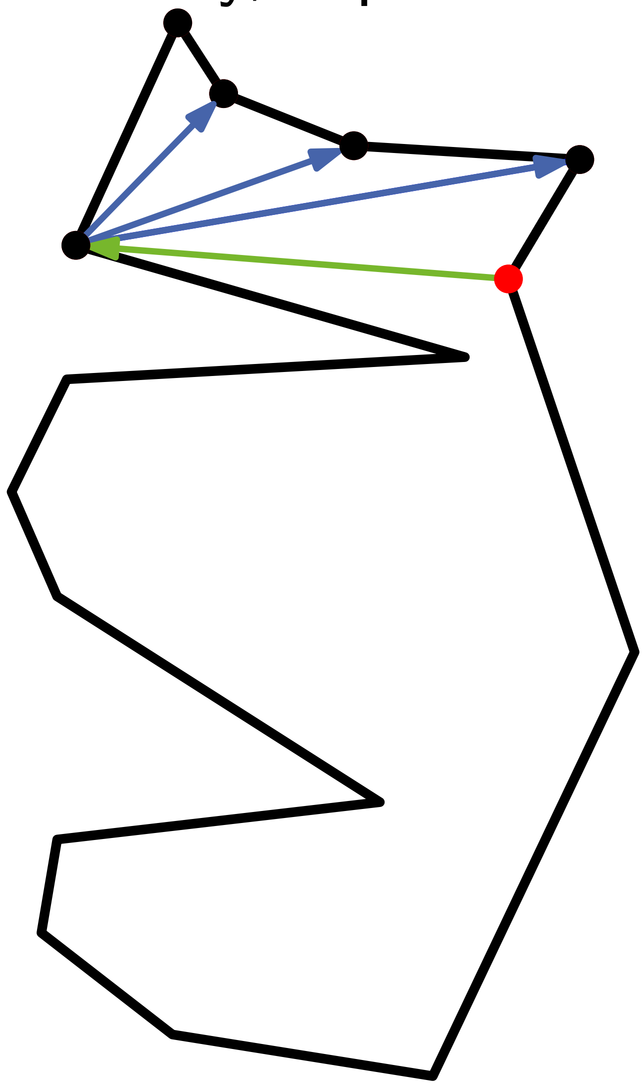
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

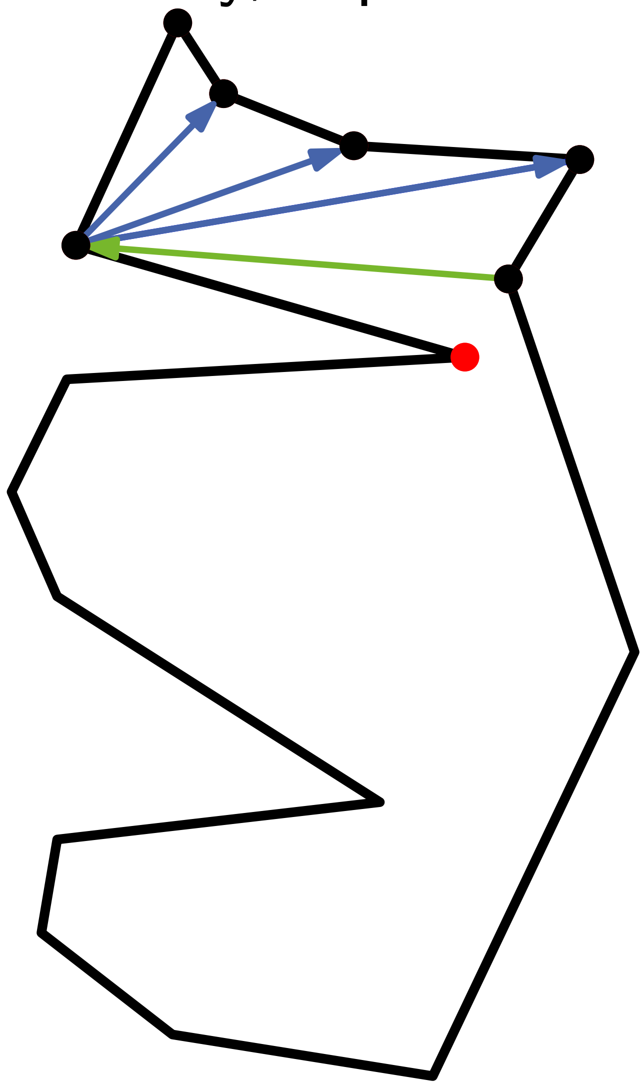
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

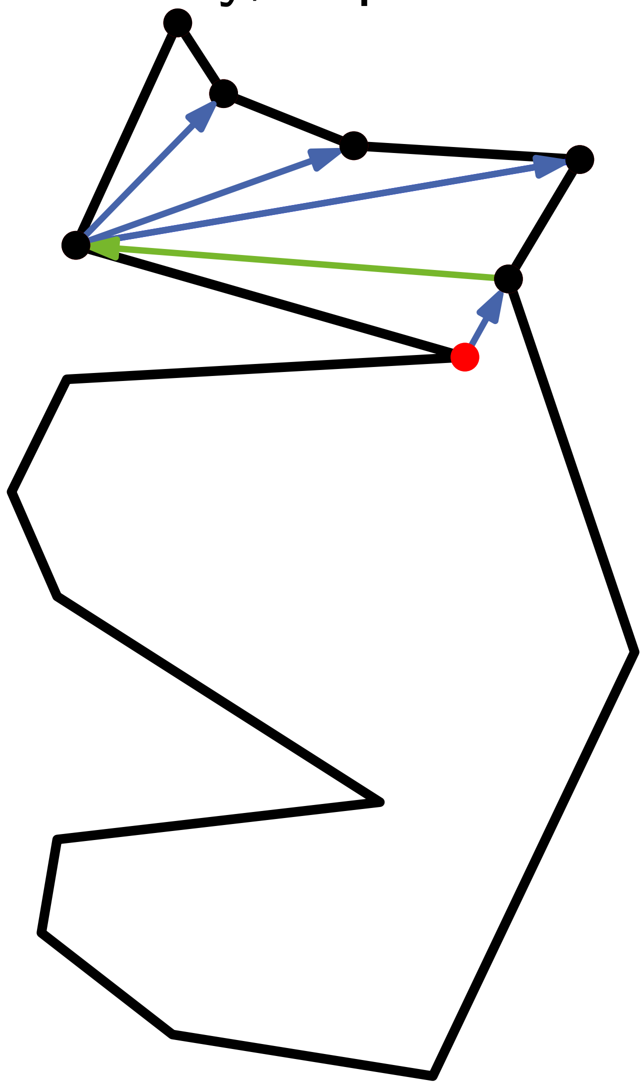
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

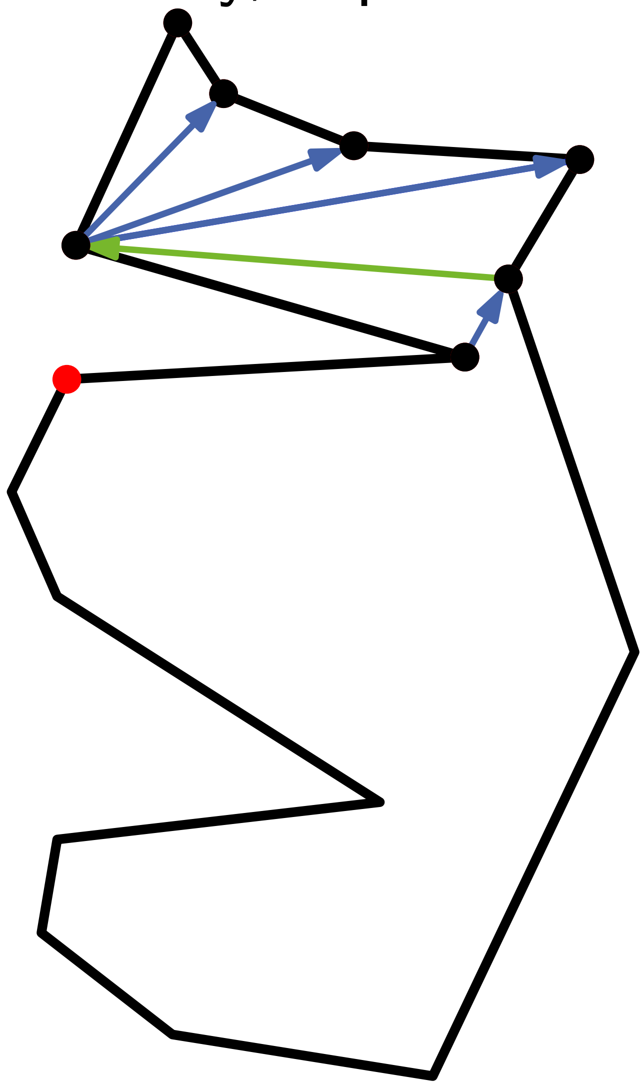
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

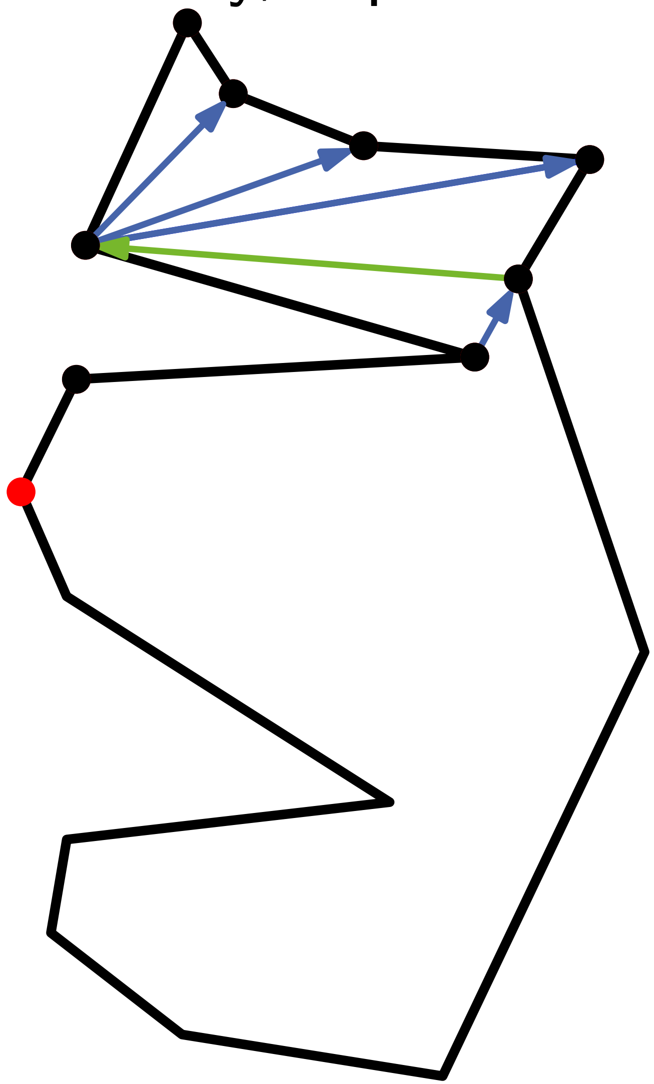
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

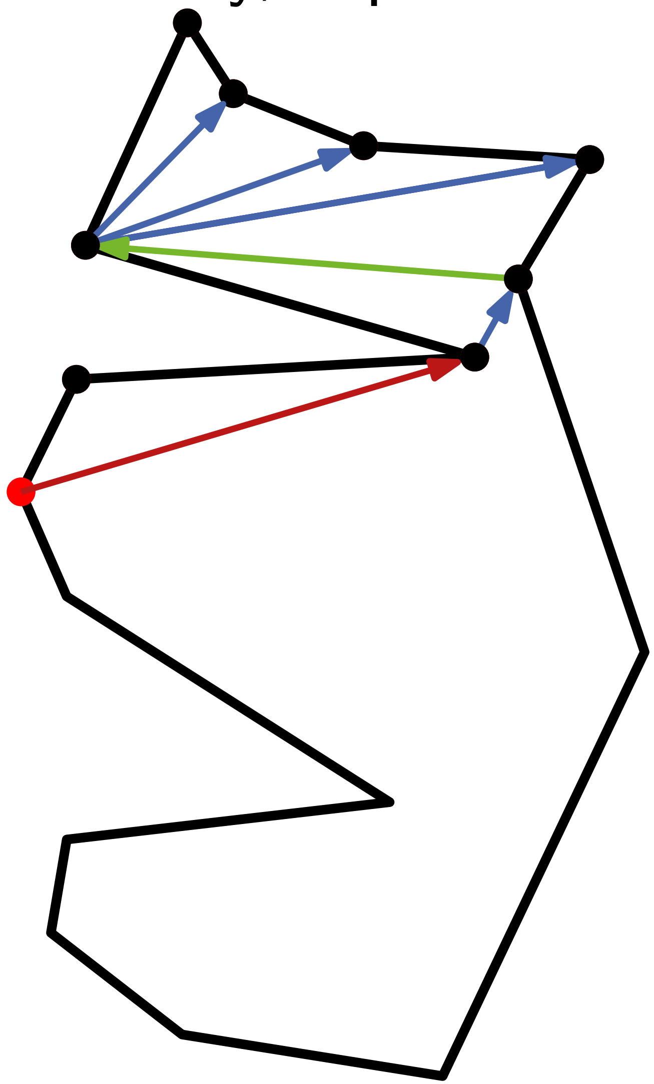
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

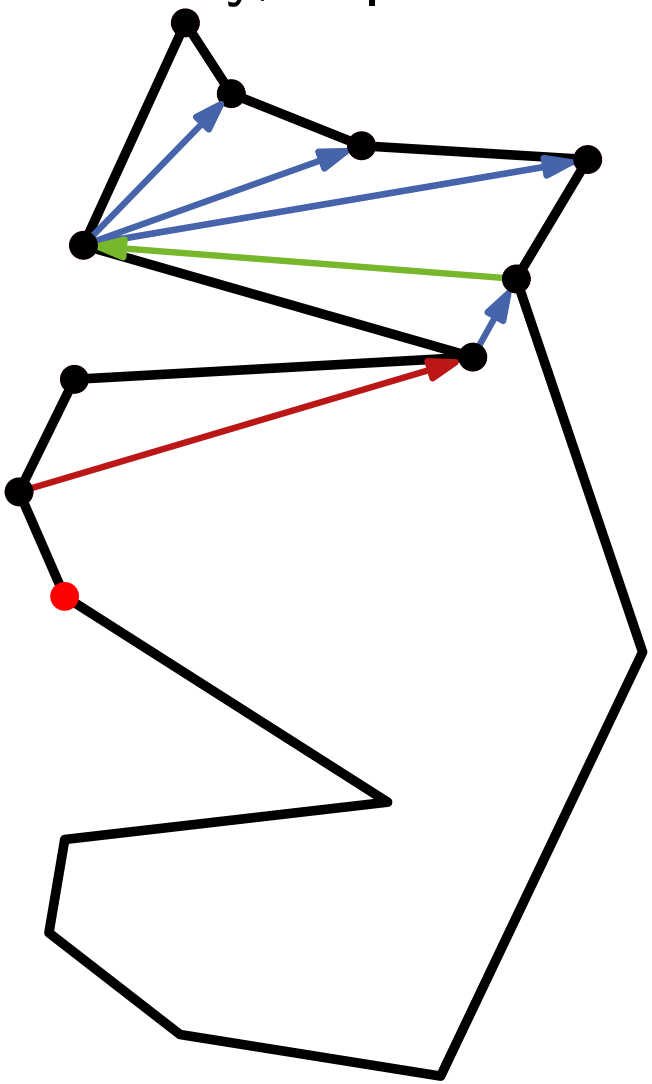
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

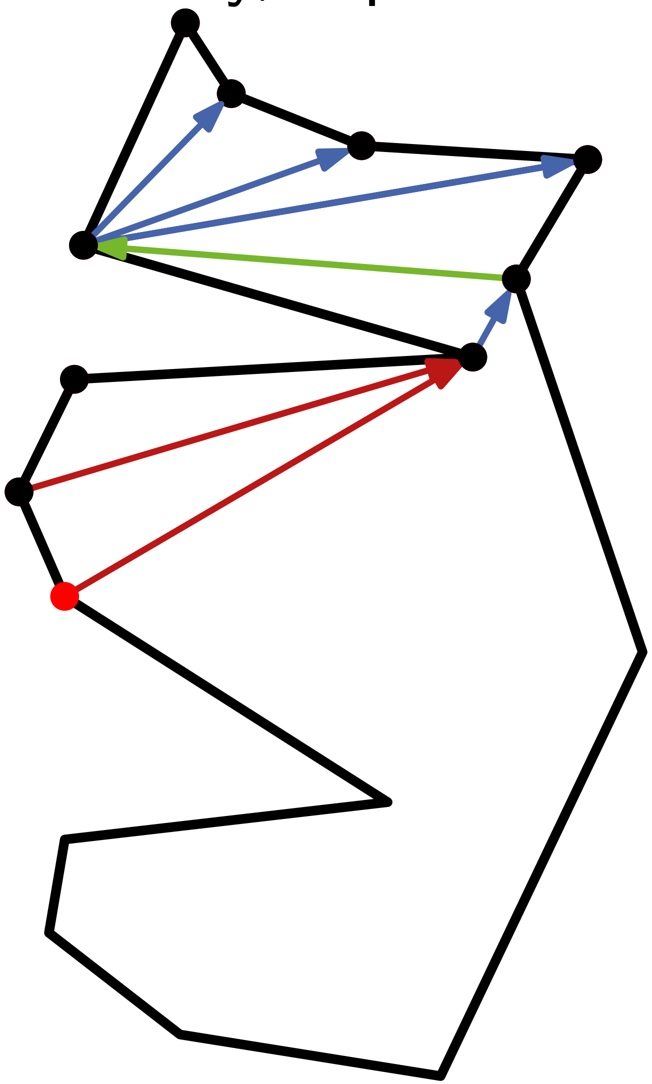
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

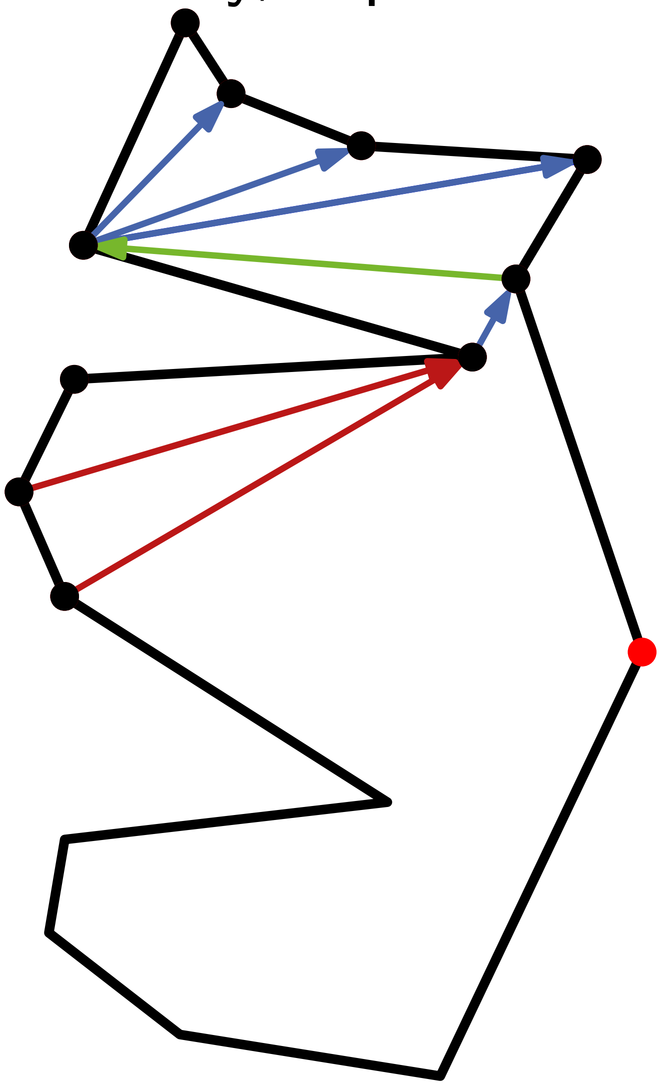
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

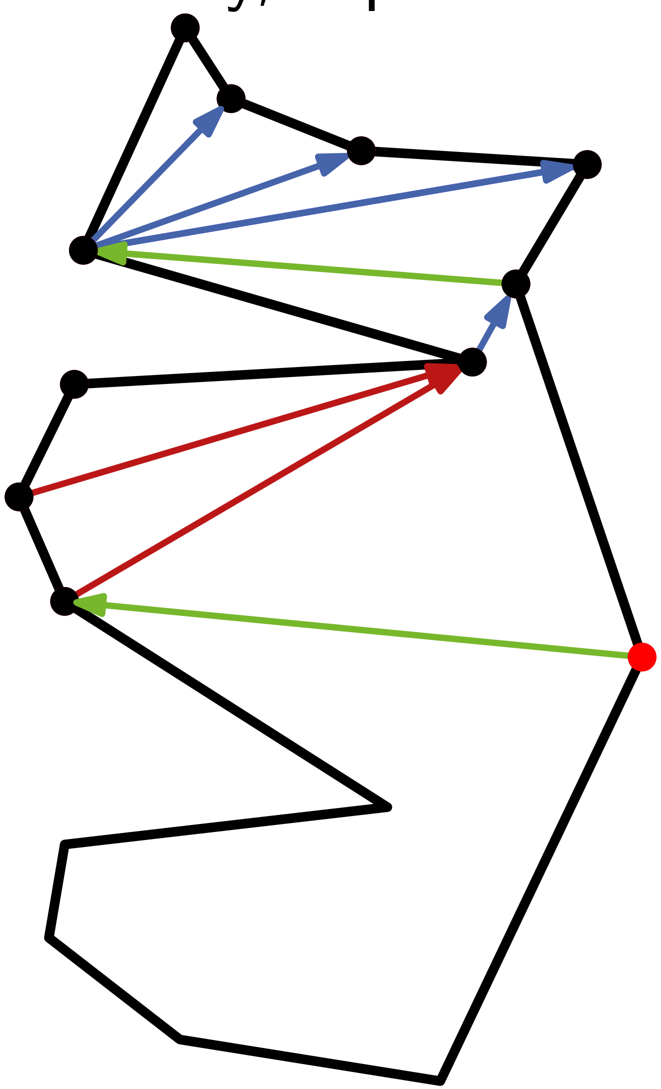
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

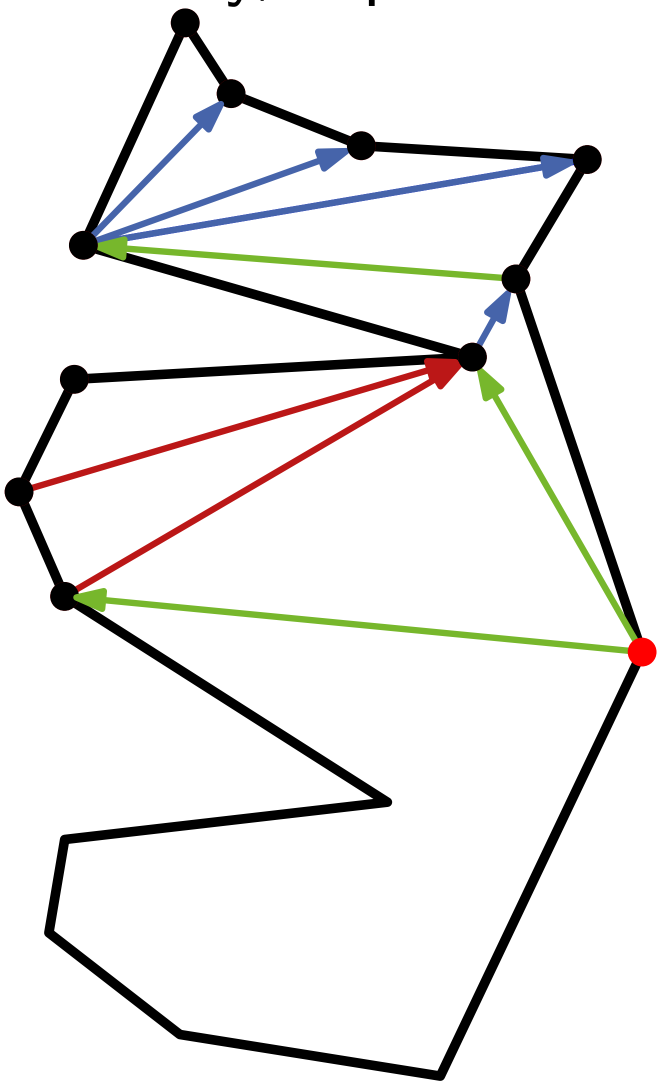
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

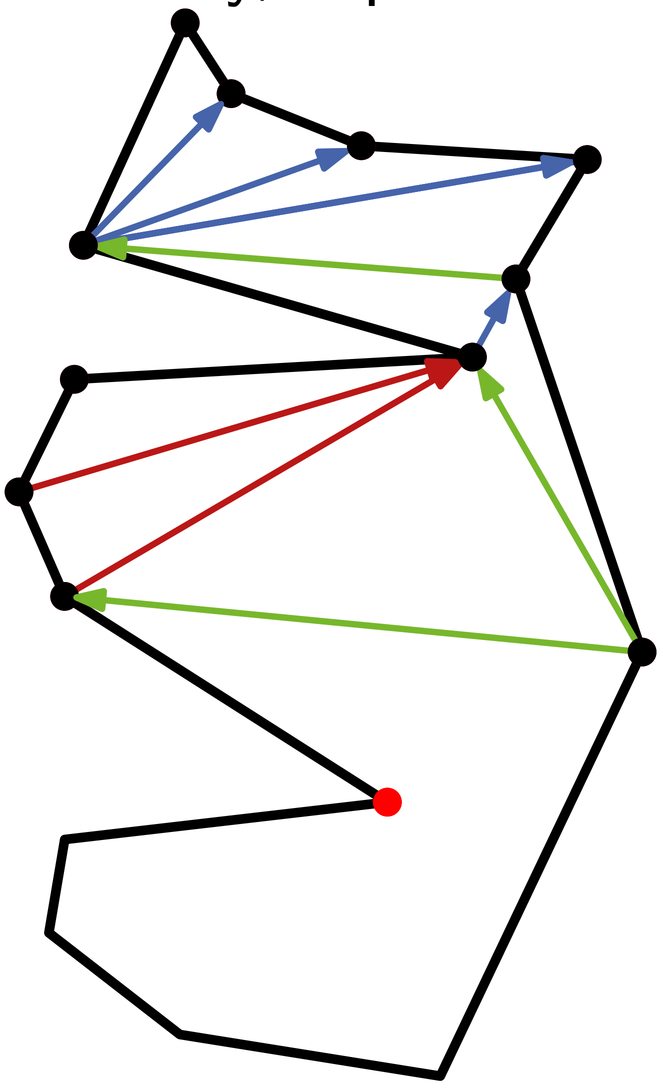
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

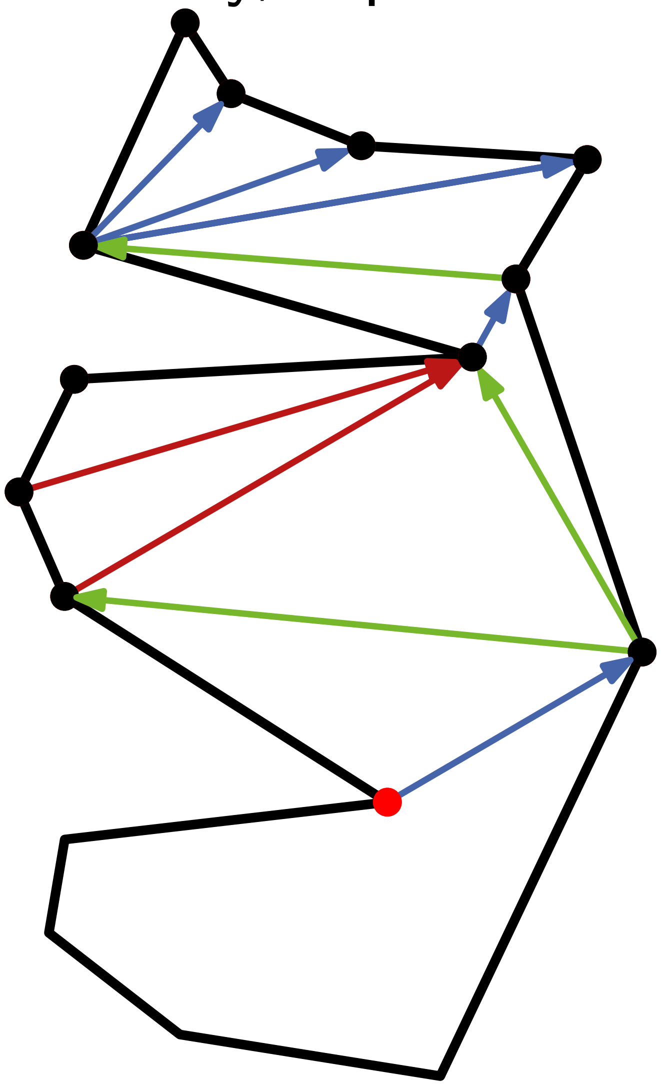
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

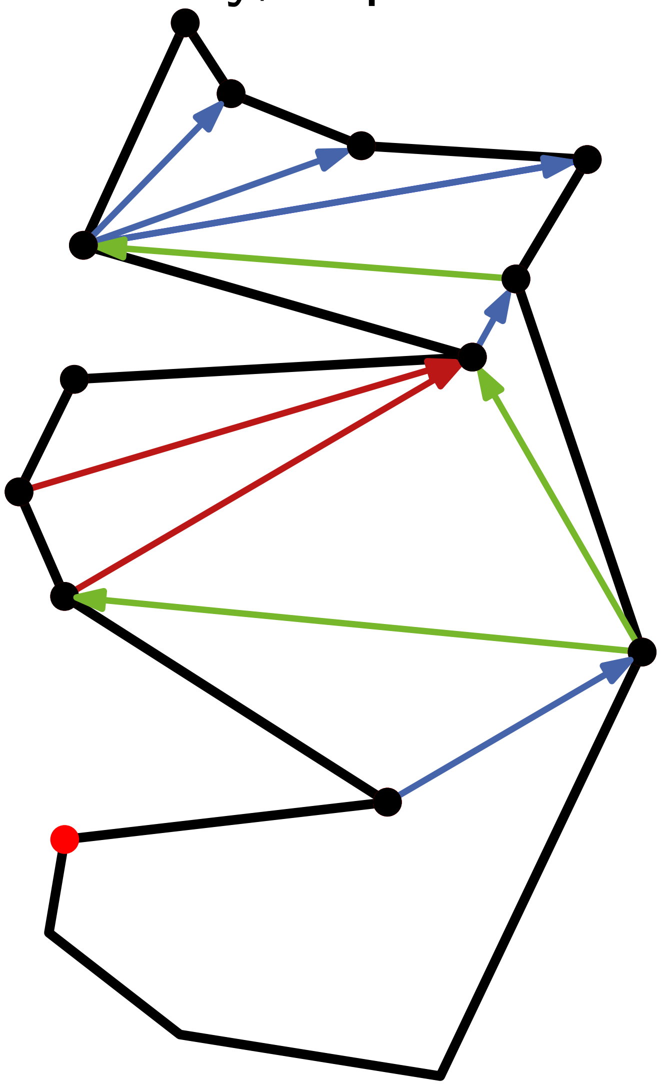
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

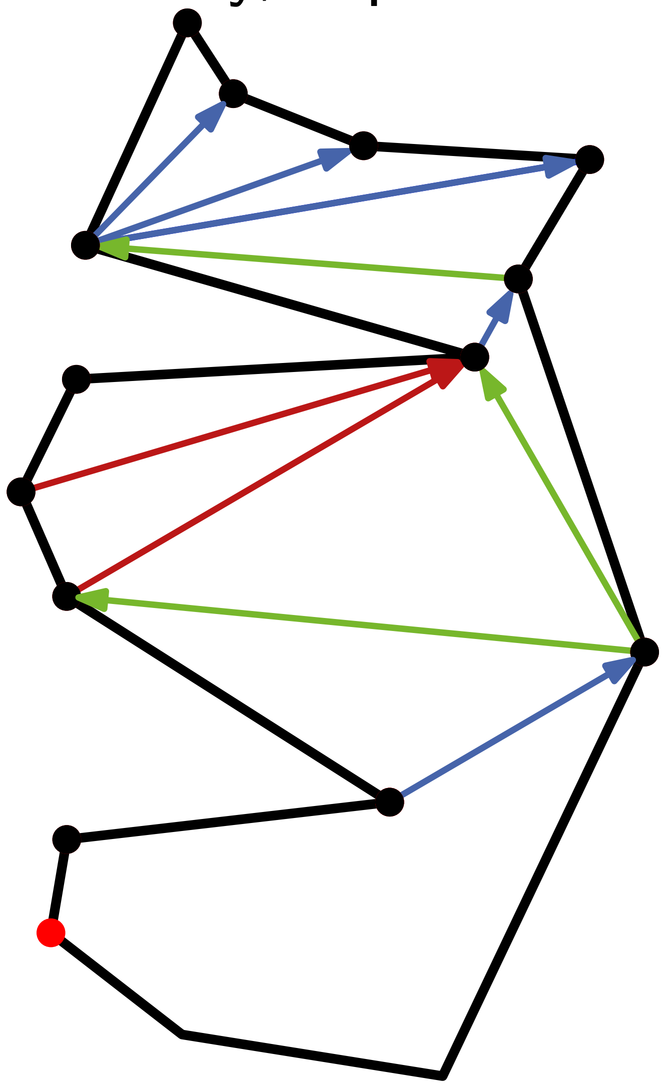
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

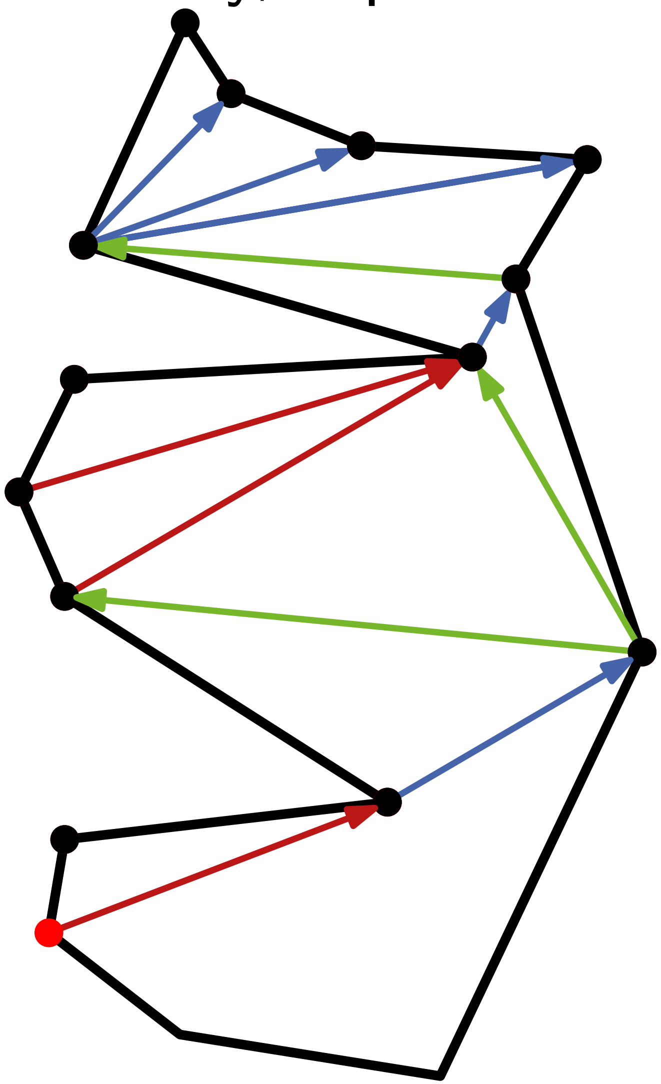
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

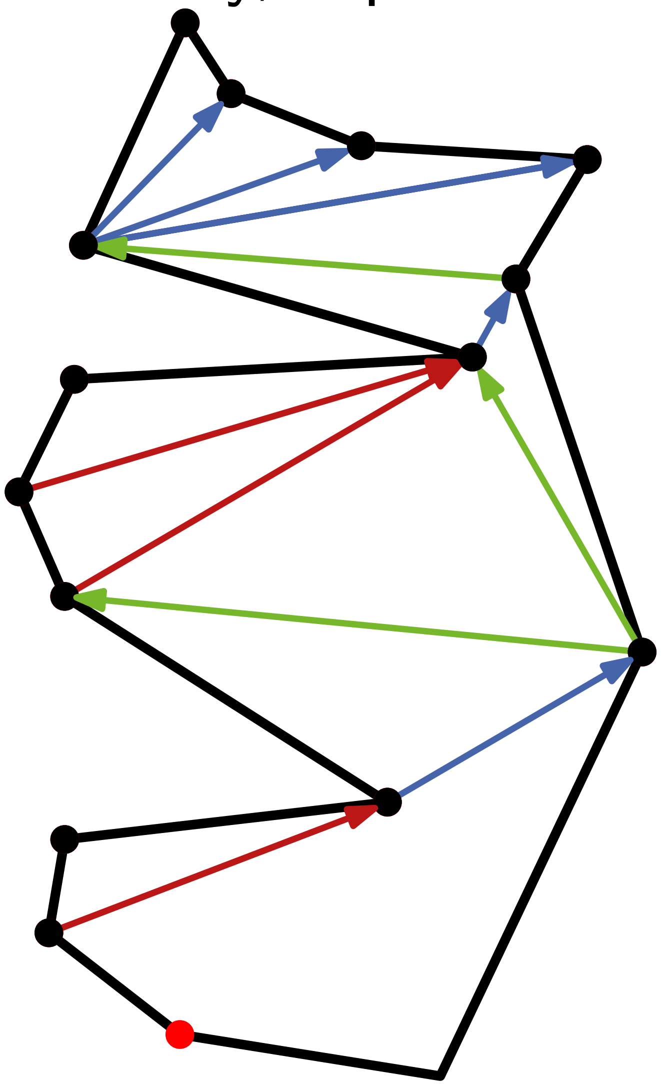
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

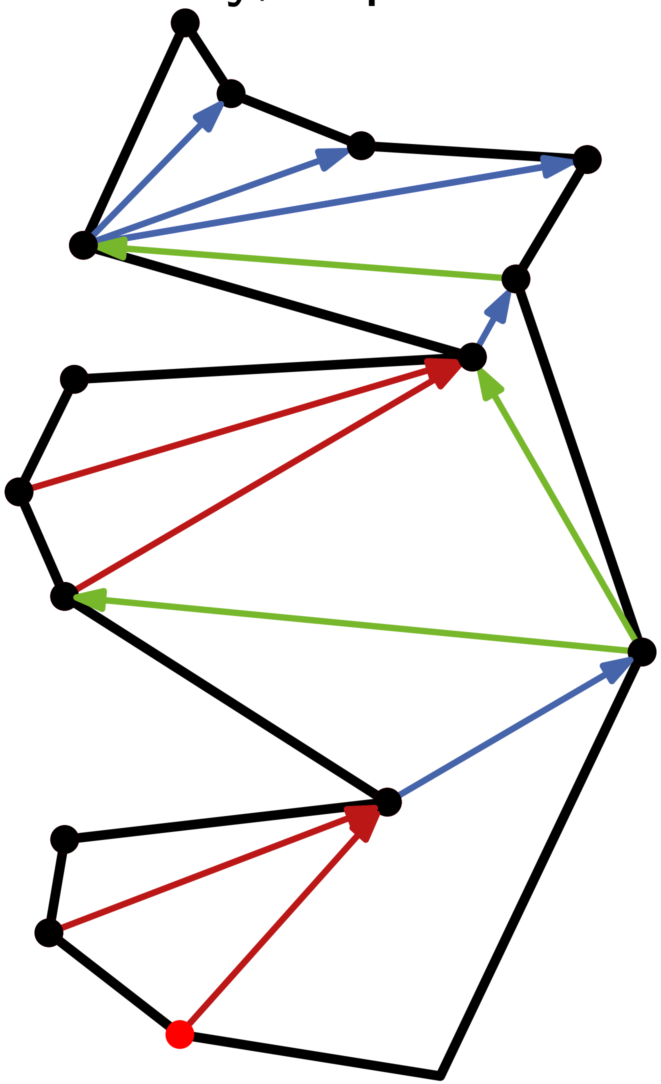
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

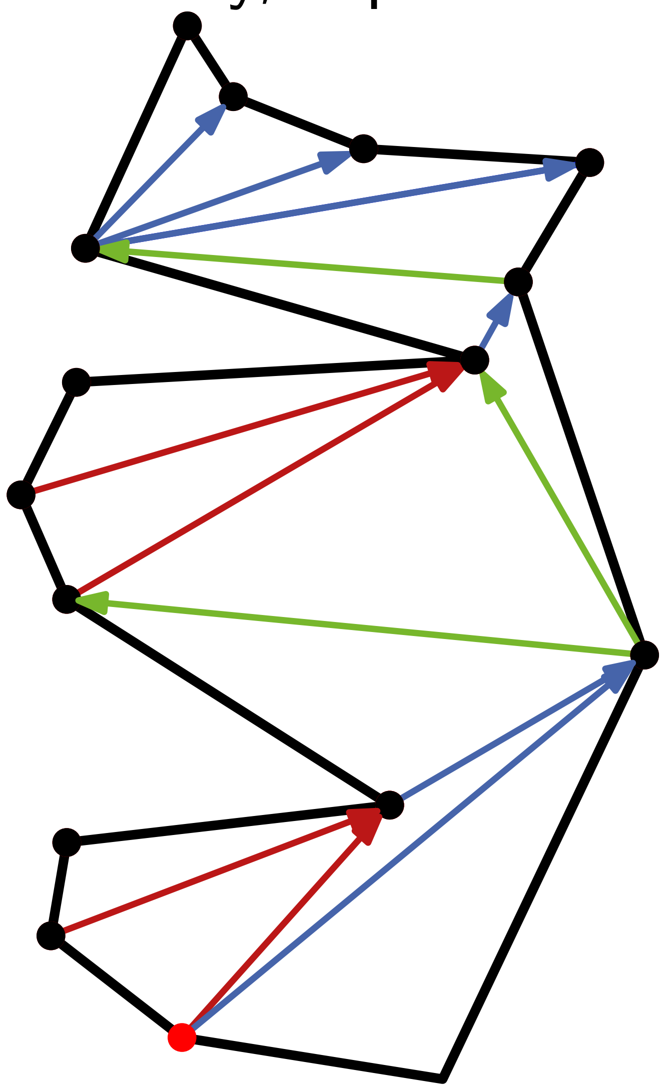
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

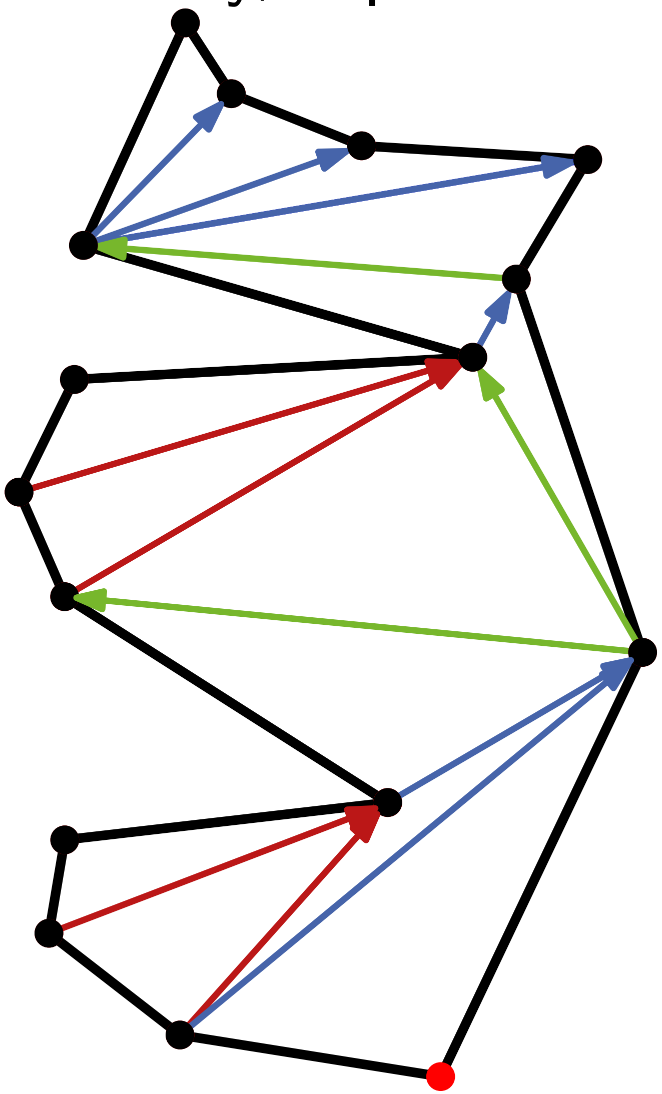
**Approach:** Greedy, top down traversal of both sides



# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

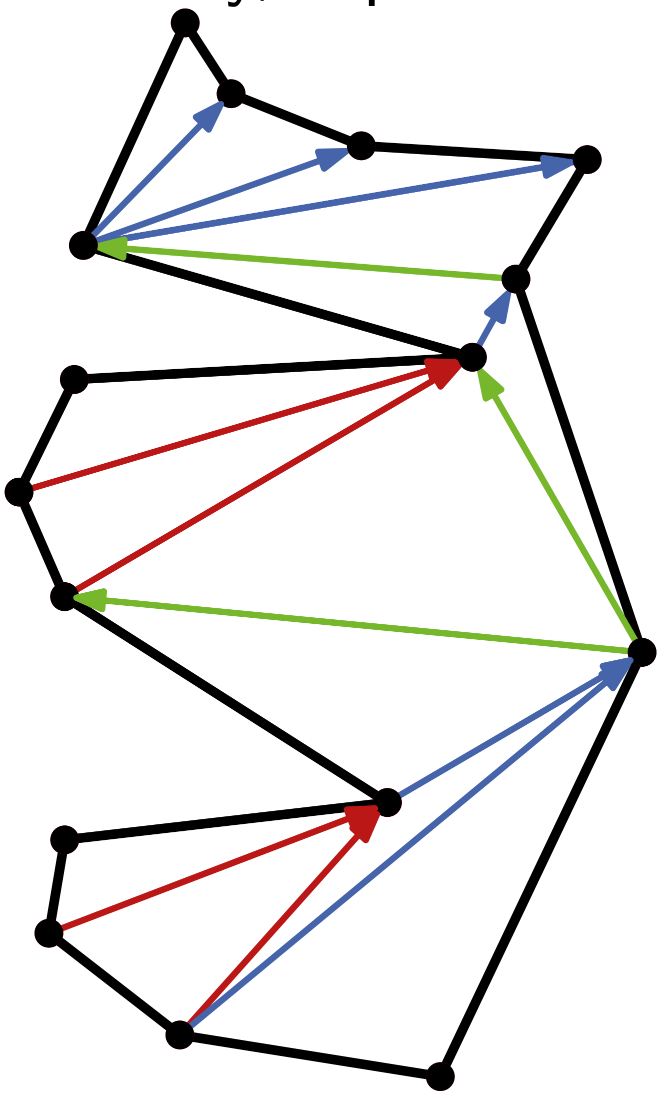


# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

**Invariant?**

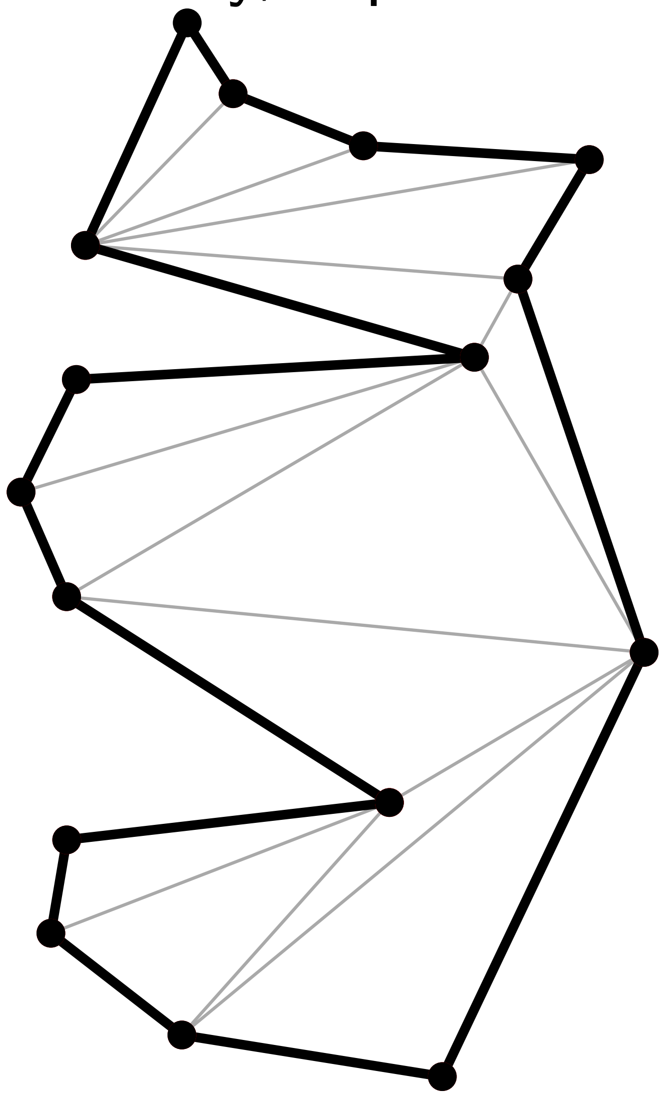


# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

**Invariant?**

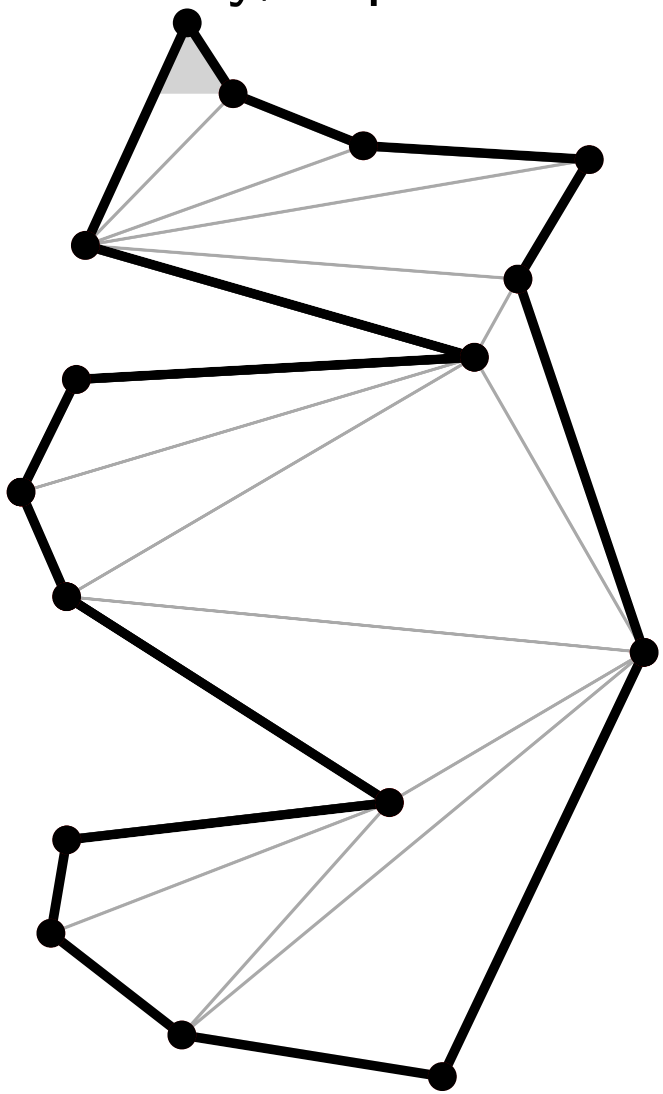


# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

**Invariant?**

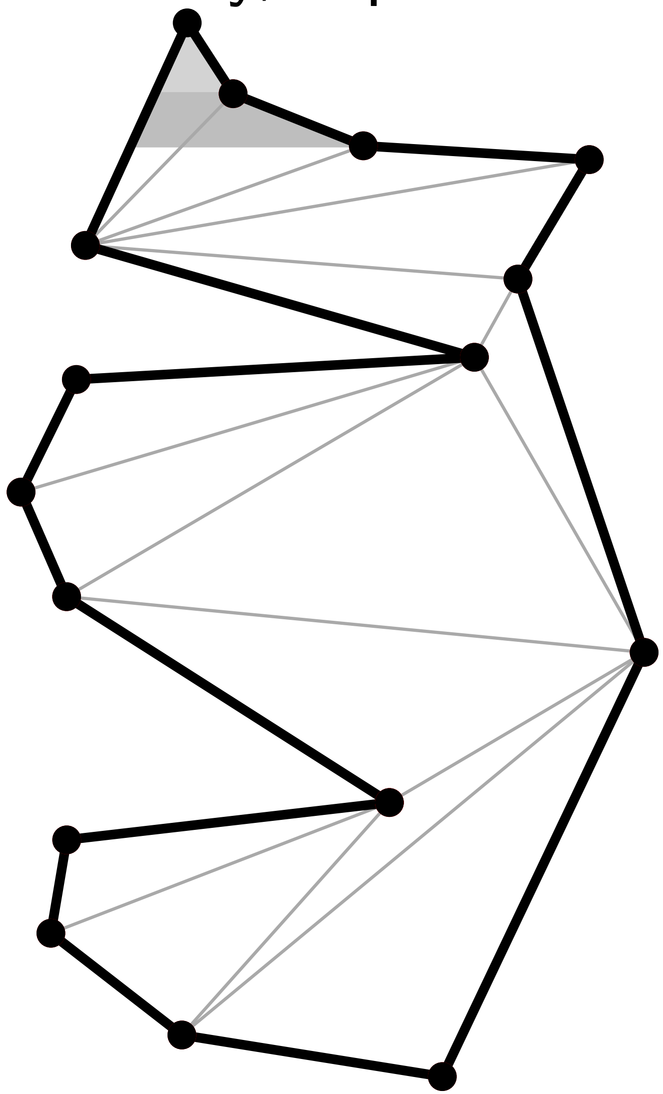


# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

**Invariant?**

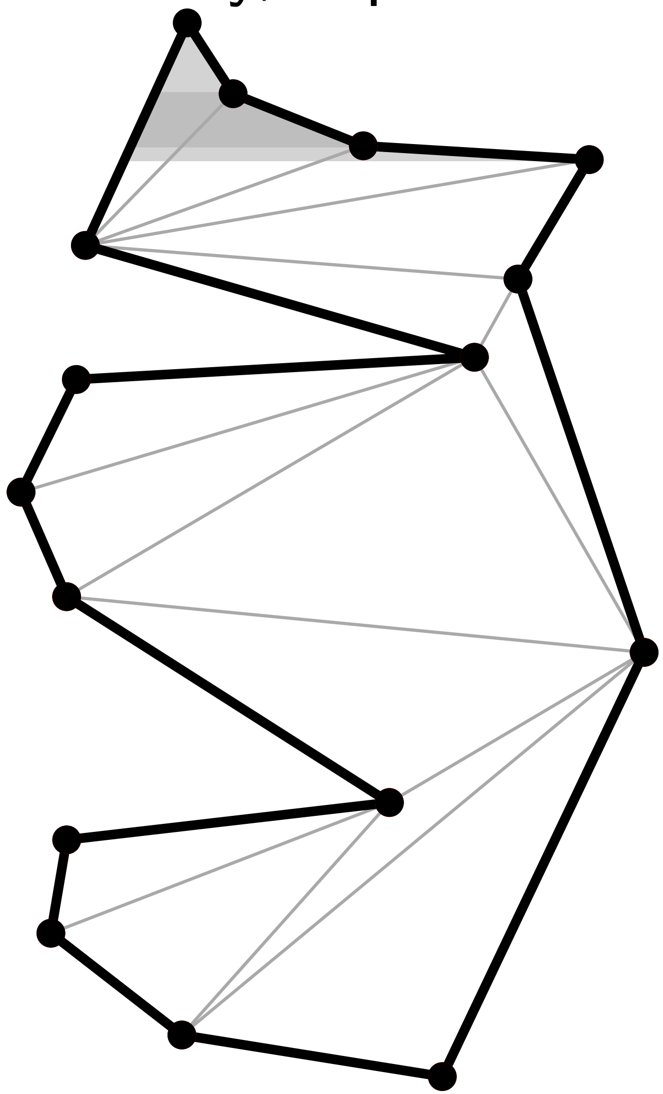


# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

**Invariant?**

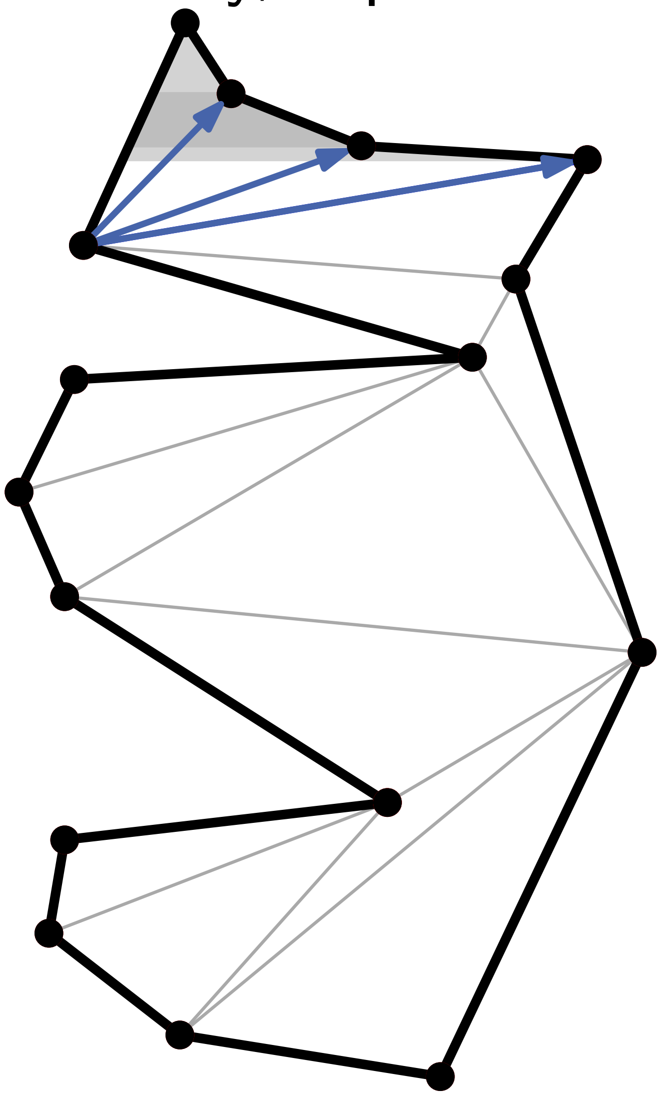


# Triangulate $y$-monotone Polygon

**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

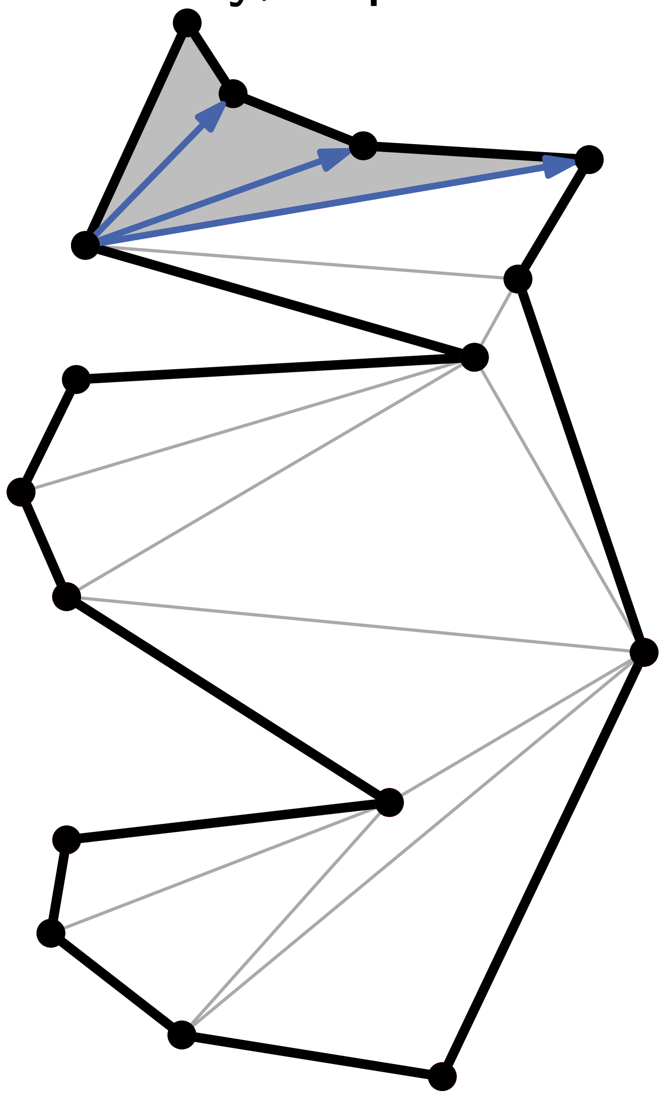
**Invariant?**



# Triangulate $y$-monotone Polygon

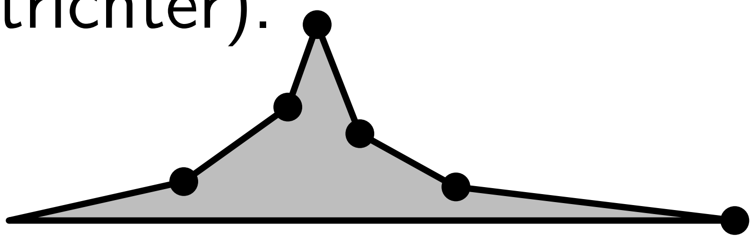
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides



**Invariant?**

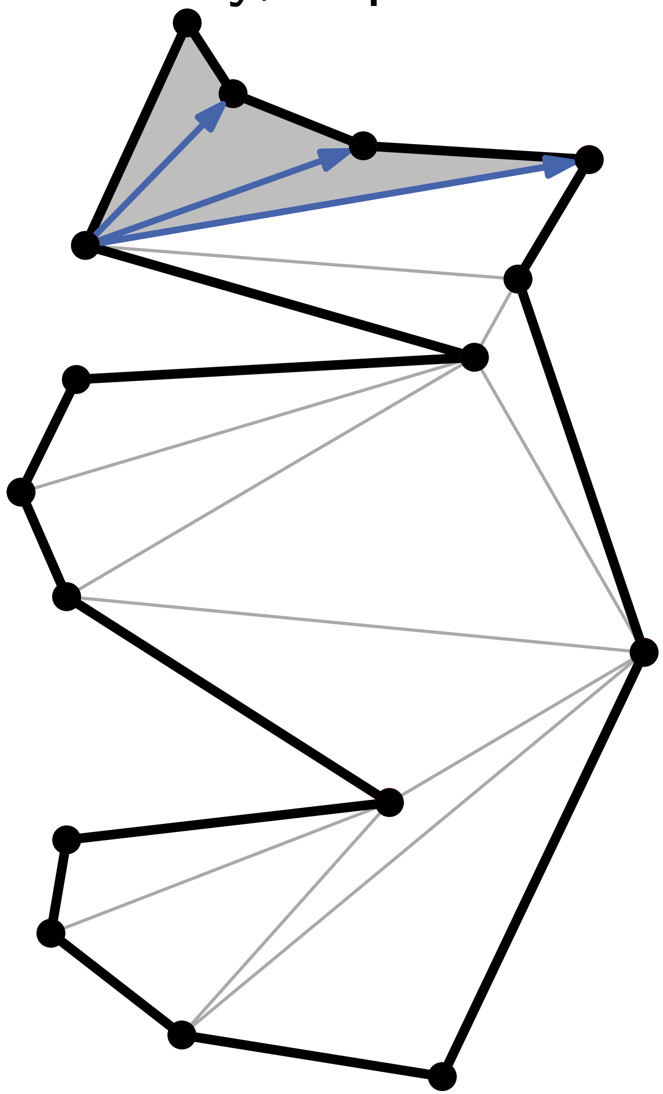
The already visited but not triangulated polygon has the shape of a *funnel* (trichter).



# Triangulate $y$-monotone Polygon

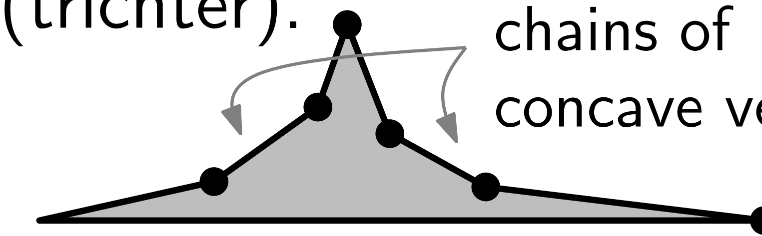
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides



**Invariant?**

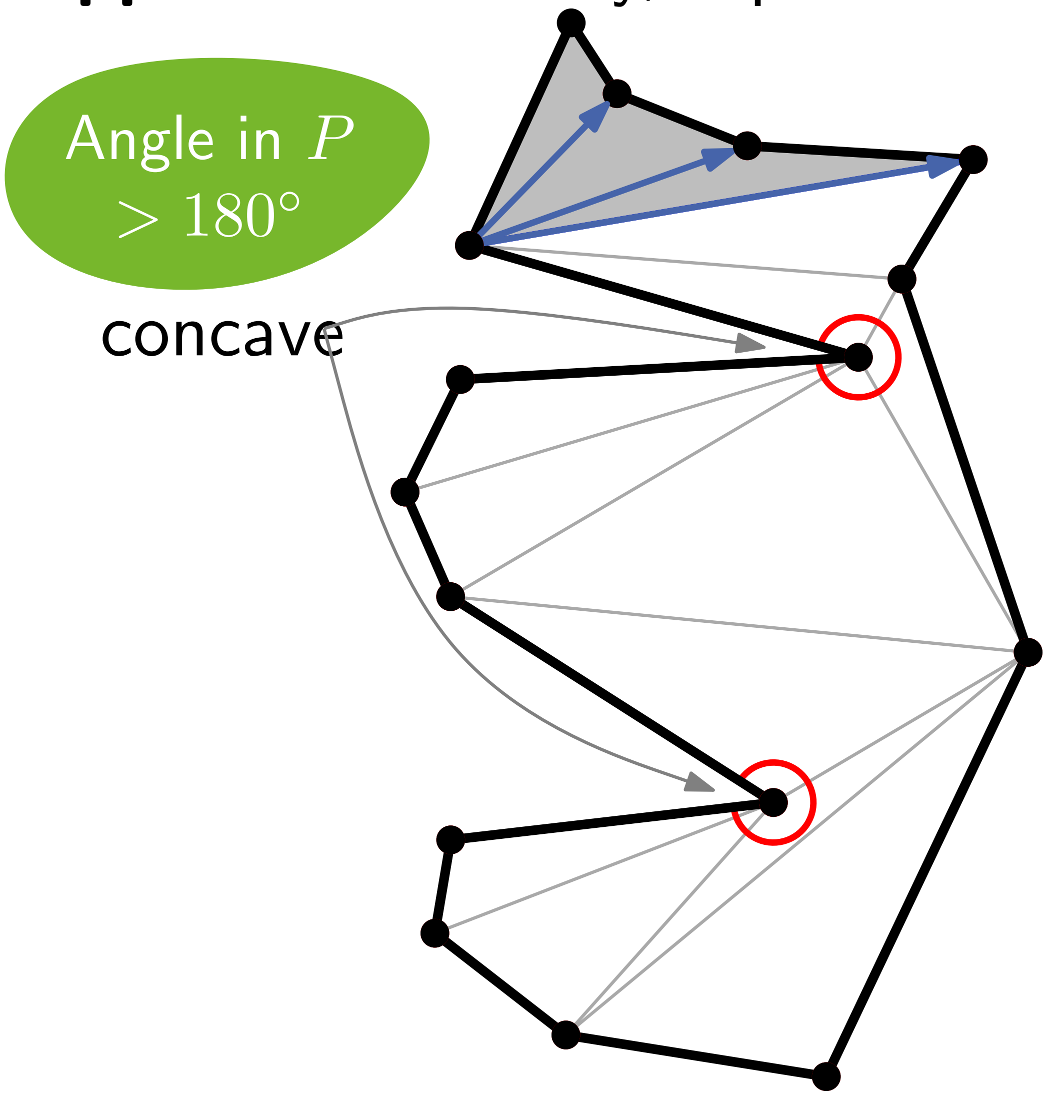
The already visited but not triangulated polygon has the shape of a *funnel* (trichter).



# Triangulate $y$-monotone Polygon

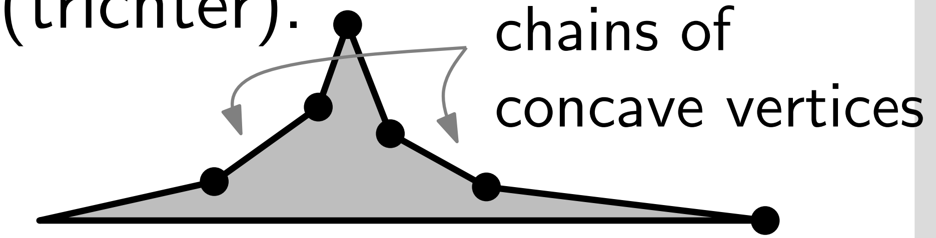
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides




**Invariant?**

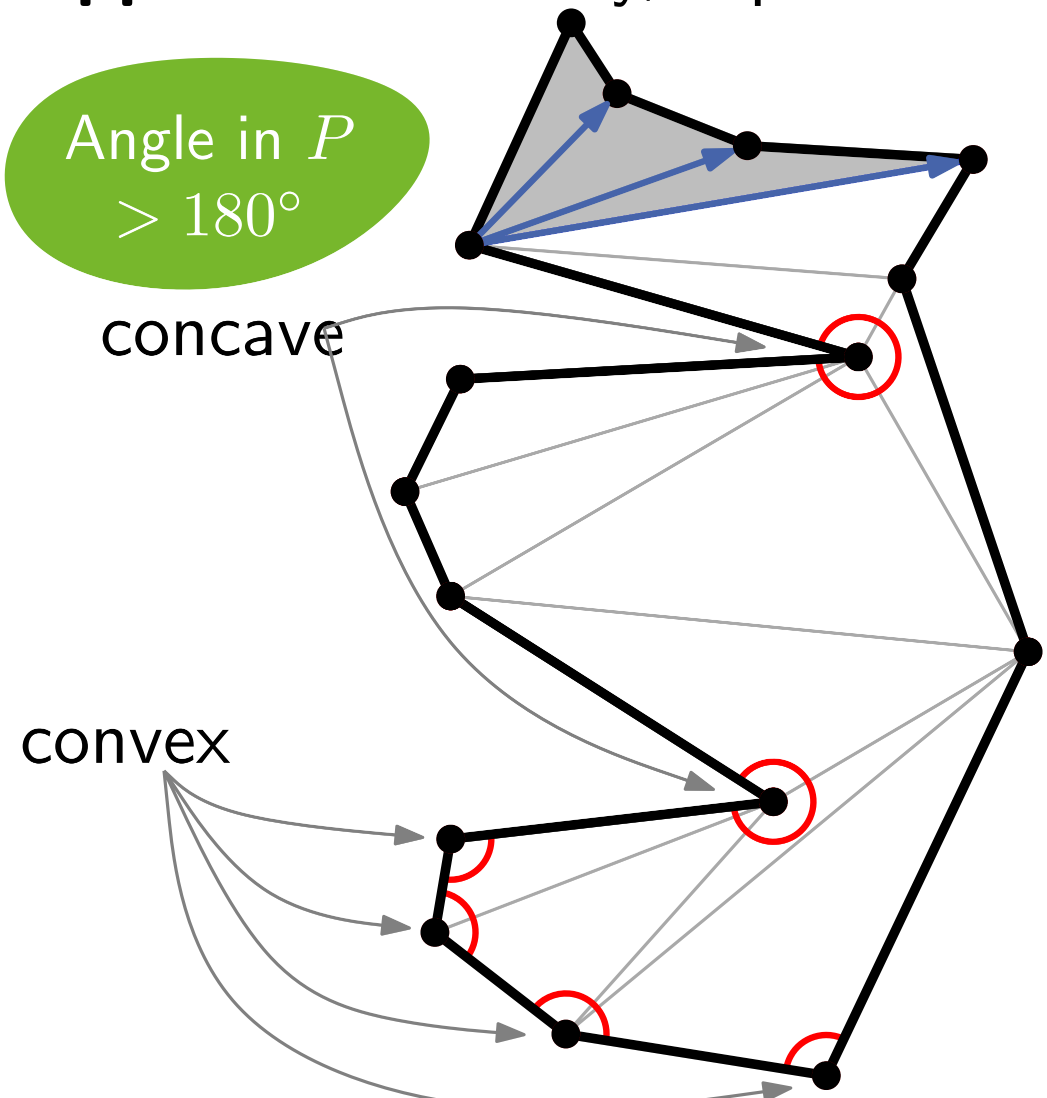
The already visited but not triangulated polygon has the shape of a *funnel* (trichter).



# Triangulate $y$-monotone Polygon

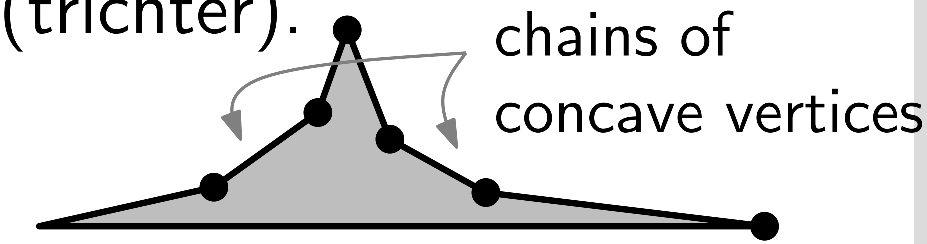
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides




**Invariant?**

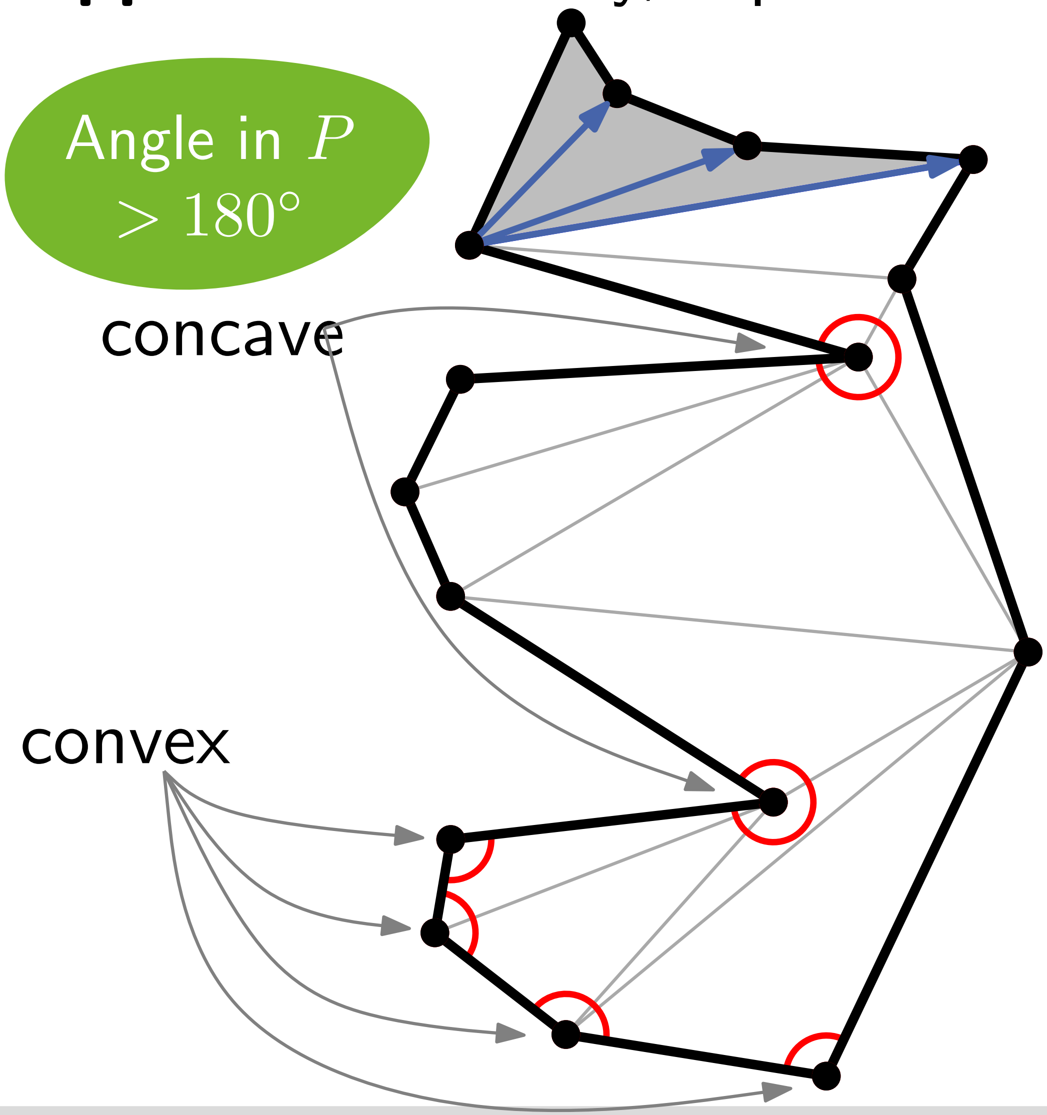
The already visited but not triangulated polygon has the shape of a *funnel* (trichter).



# Triangulate $y$-monotone Polygon

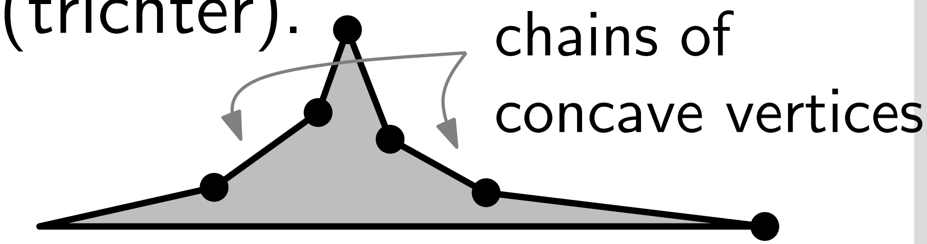
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

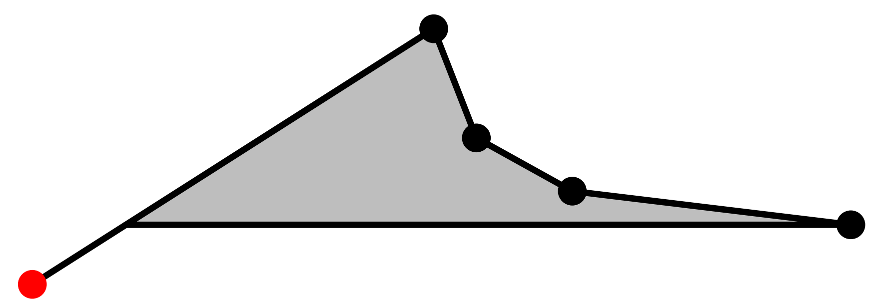



**Invariant?**

The already visited but not triangulated polygon has the shape of a *funnel* (trichter).



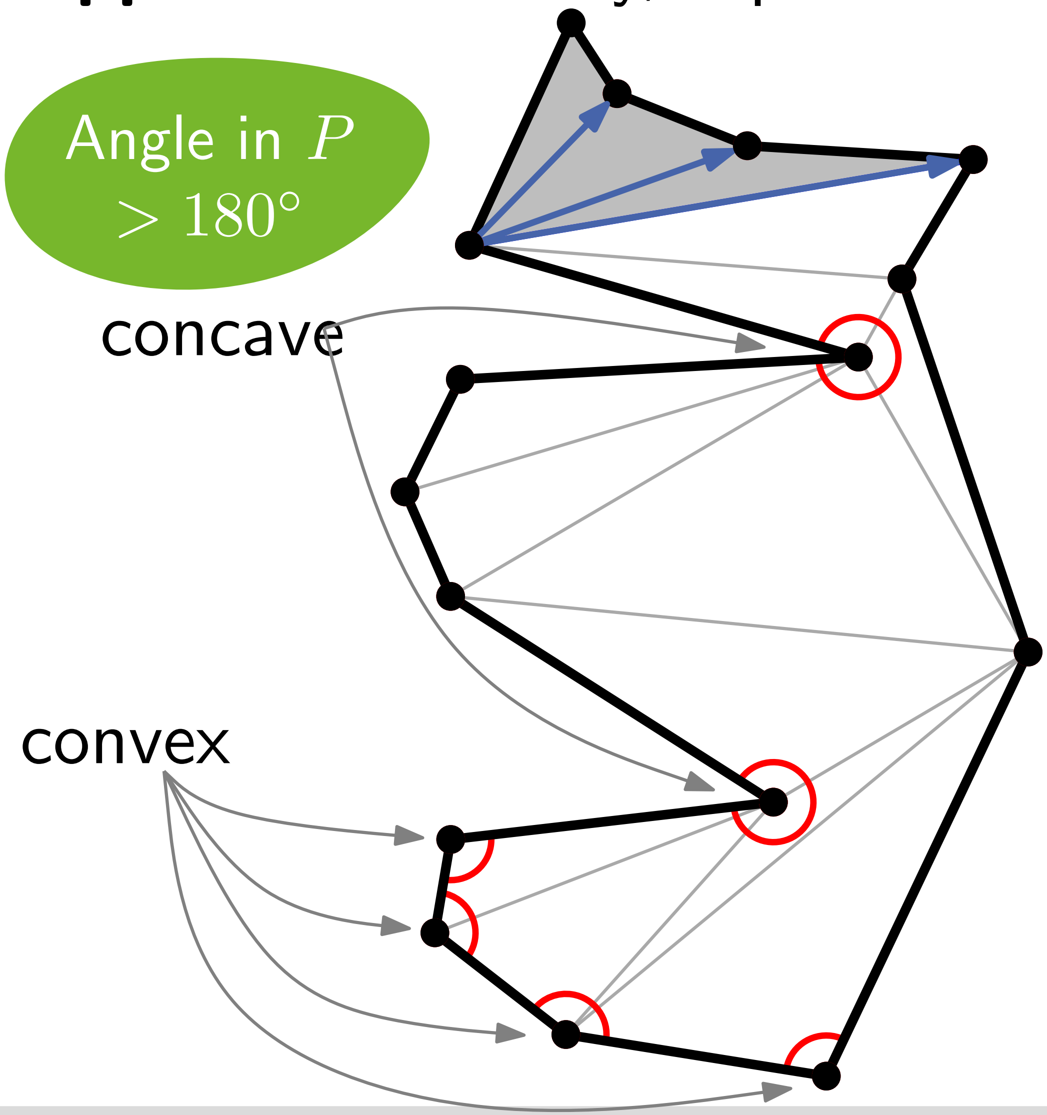

In our case:



# Triangulate $y$-monotone Polygon

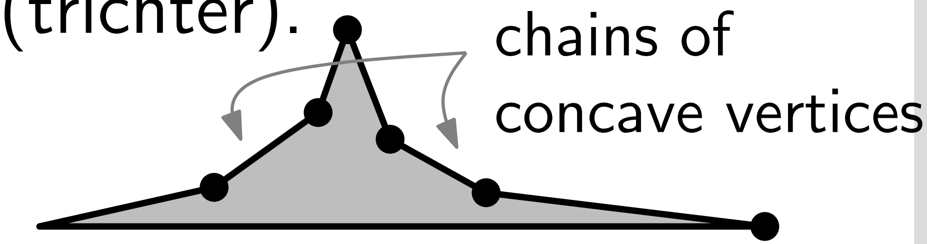
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

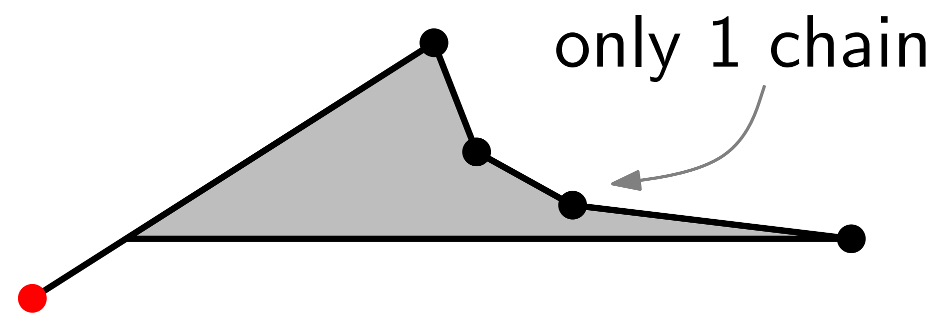



**Invariant?**

The already visited but not triangulated polygon has the shape of a *funnel* (trichter).



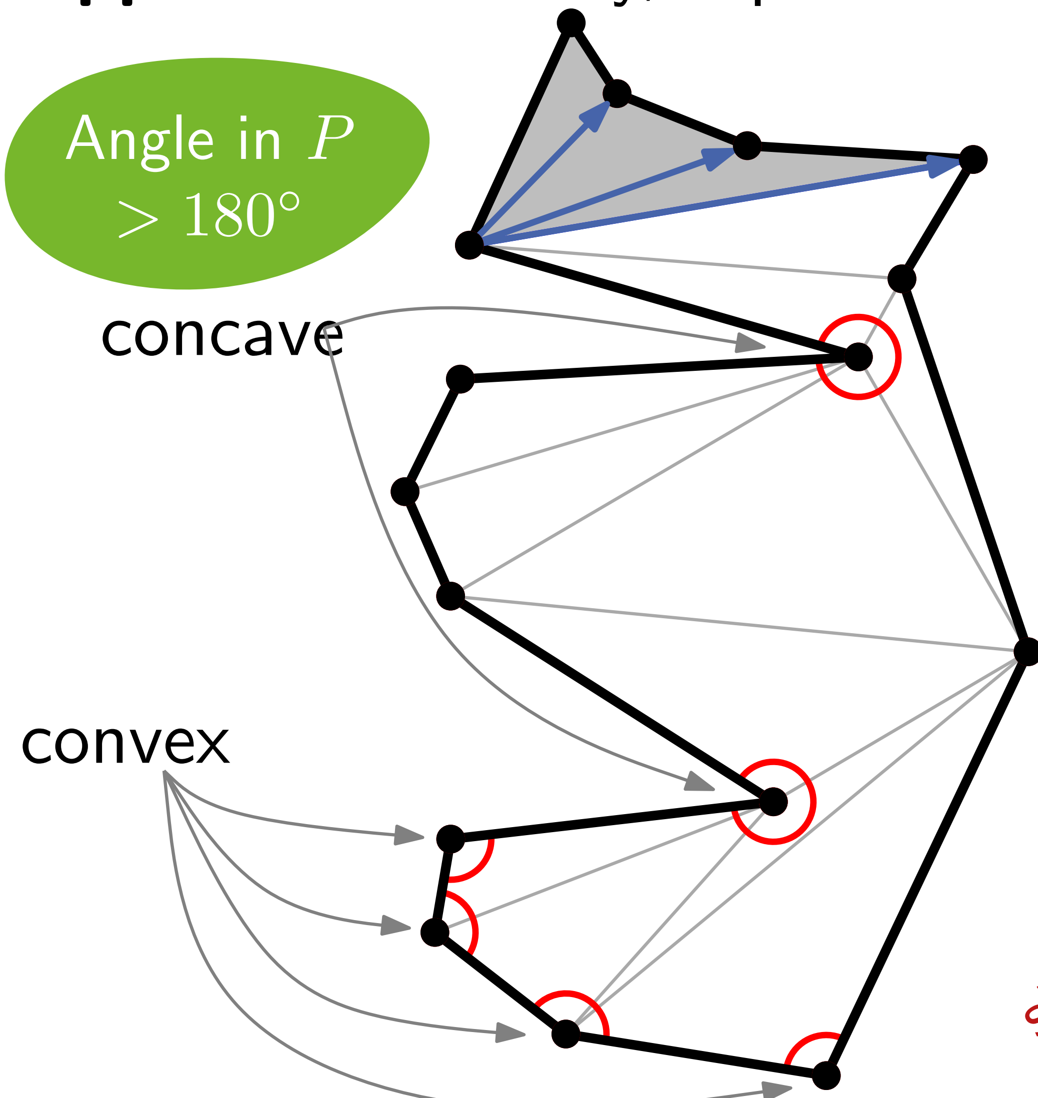

In our case:



# Triangulate $y$-monotone Polygon

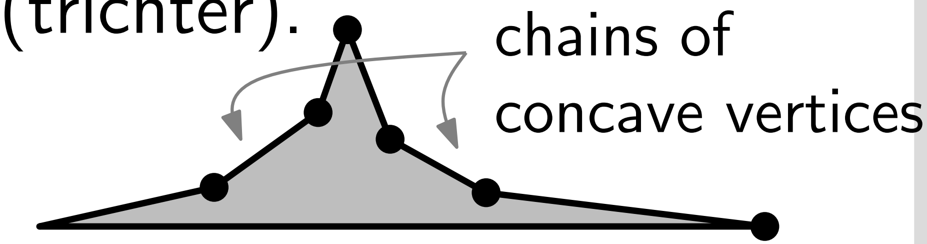
**Reminder:** The left and the right boundary of the polygon have decreasing $y$-coordinates

**Approach:** Greedy, top down traversal of both sides

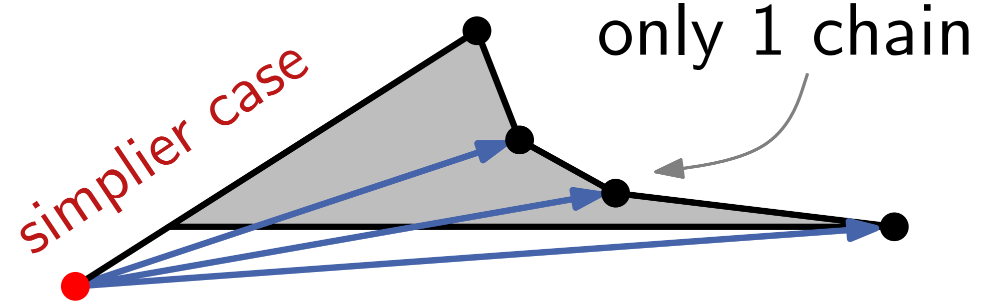



**Invariant?**

The already visited but not triangulated polygon has the shape of a *funnel* (trichter).




In our case:



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$

# Algorithm TriangulateMonotonePolygon

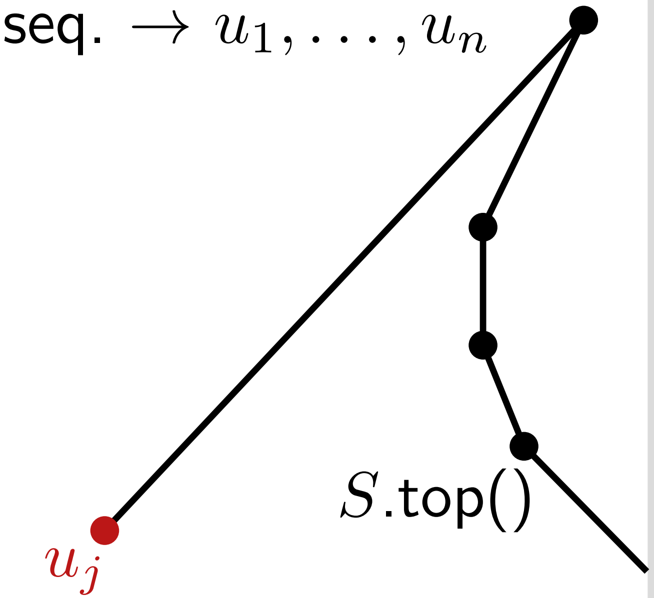
TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

```
Merge vertices on left and right chains into desc. 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() from different paths then
        while not S.empty() do
            v ← S.pop()
            if not S.empty() then draw (u_j, v)
        S.push(u_{j−1}); S.push(u_j)
```

# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

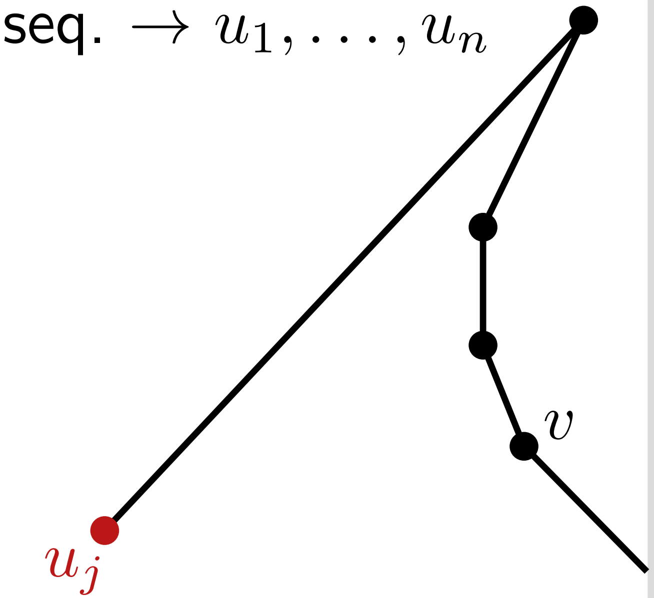
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
  **if** $u_j$ and $S$.top() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push($u_{j-1}$); $S$.push($u_j$)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

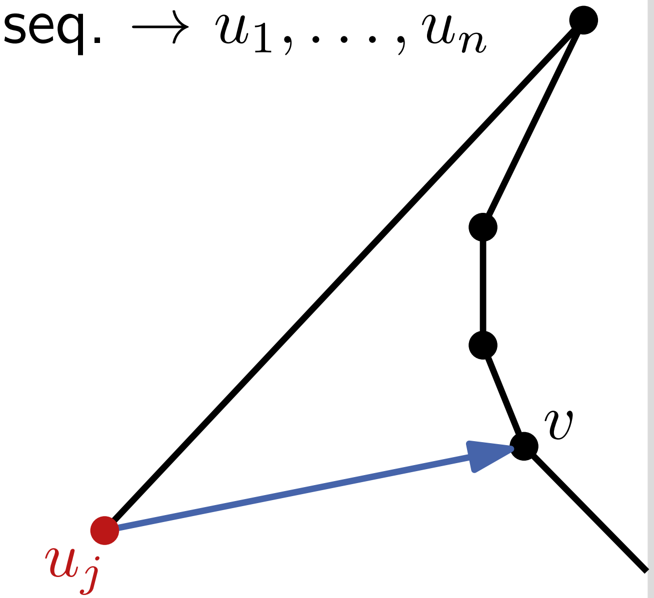
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push$(u_1)$; $S$.push$(u_2)$
**for** $j \leftarrow 3$ **to** $n-1$ **do**
  **if** $u_j$ and $S$.top() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push$(u_{j-1})$; $S$.push$(u_j)$



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

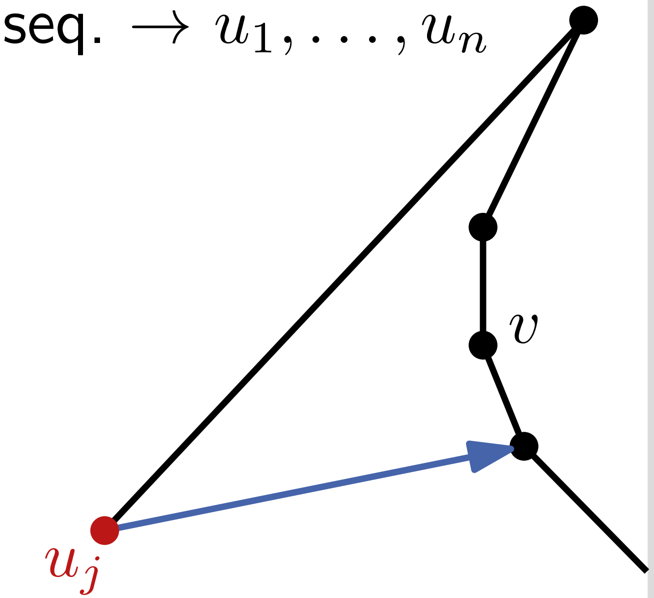
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
  **if** $u_j$ and $S$.top() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push($u_{j-1}$); $S$.push($u_j$)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

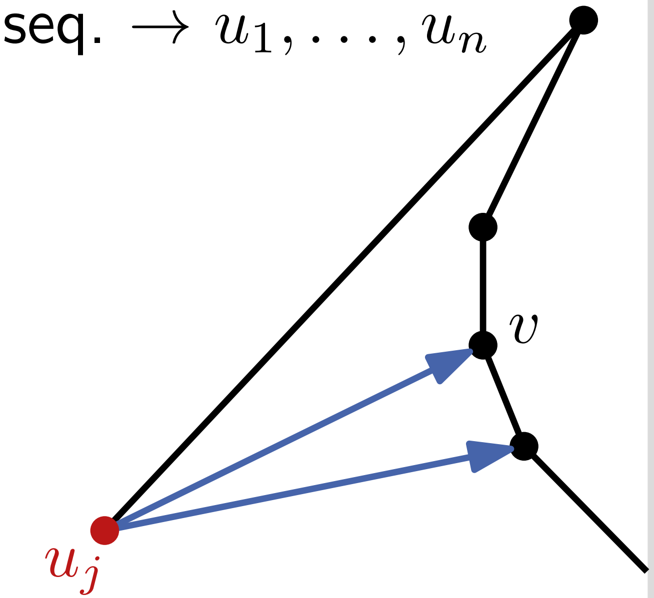
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
  **if** $u_j$ and $S$.top() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push($u_{j-1}$); $S$.push($u_j$)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

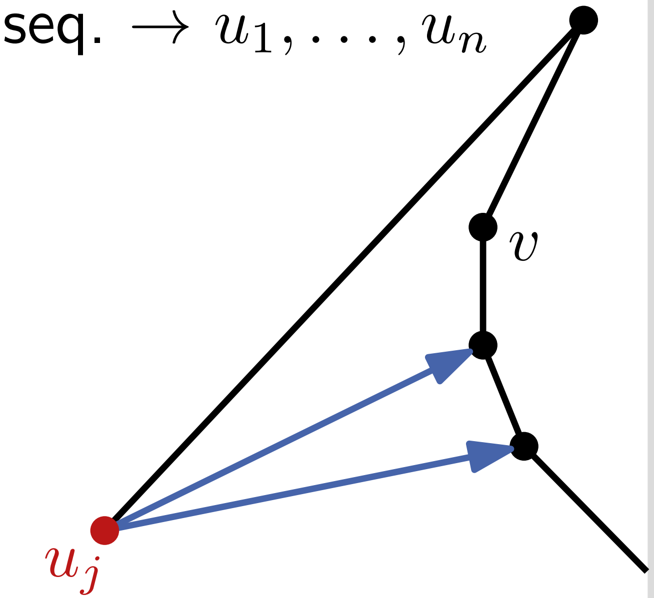
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push$(u_1)$; $S$.push$(u_2)$
**for** $j \leftarrow 3$ **to** $n-1$ **do**
    **if** $u_j$ and $S$.top() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push$(u_{j-1})$; $S$.push$(u_j)$



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

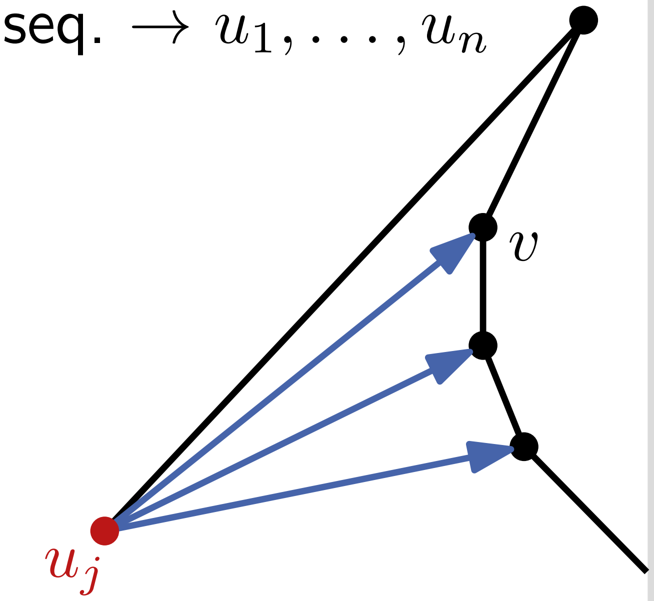
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
    **if** $u_j$ and $S$.top() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push($u_{j-1}$); $S$.push($u_j$)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

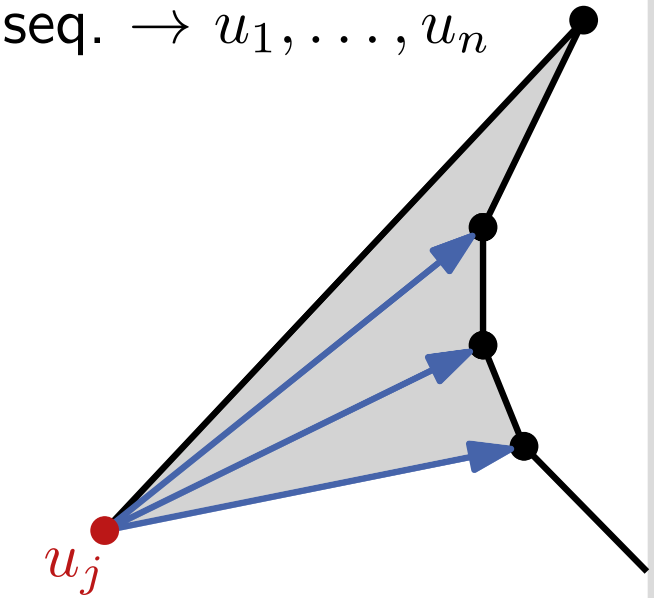
Merge vertices on left and right chains into desc. seq. $\to u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
    **if** $u_j$ and $S$.top() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push($u_{j-1}$); $S$.push($u_j$)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

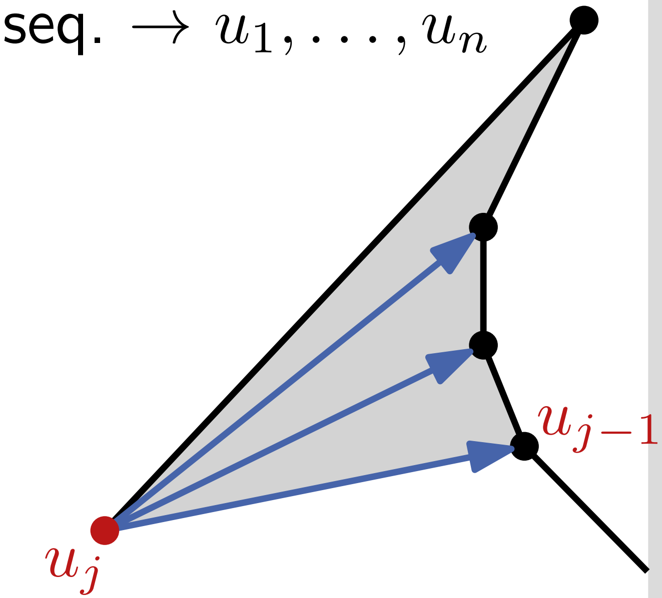
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
  **if** $u_j$ and $S$.top() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push($u_{j-1}$); $S$.push($u_j$)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

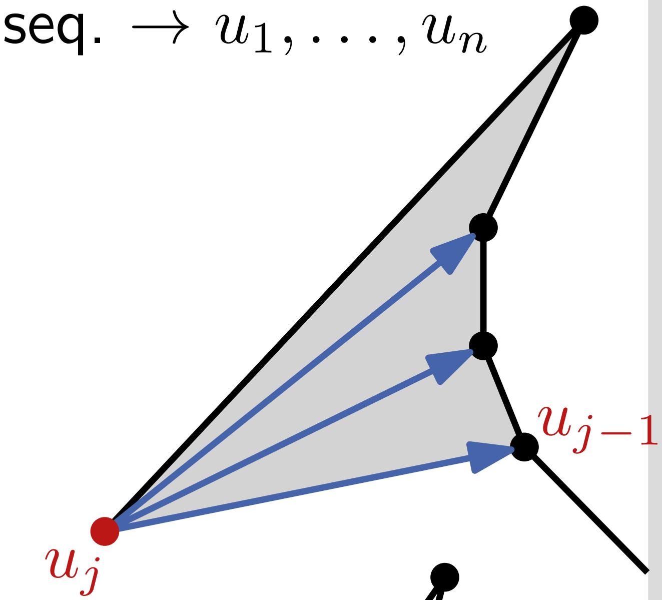
Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
Stack $S \leftarrow \emptyset$; $S$.push($u_1$); $S$.push($u_2$)
**for** $j \leftarrow 3$ **to** $n-1$ **do**
  **if** $u_j$ and $S$.top() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push($u_{j-1}$); $S$.push($u_j$)

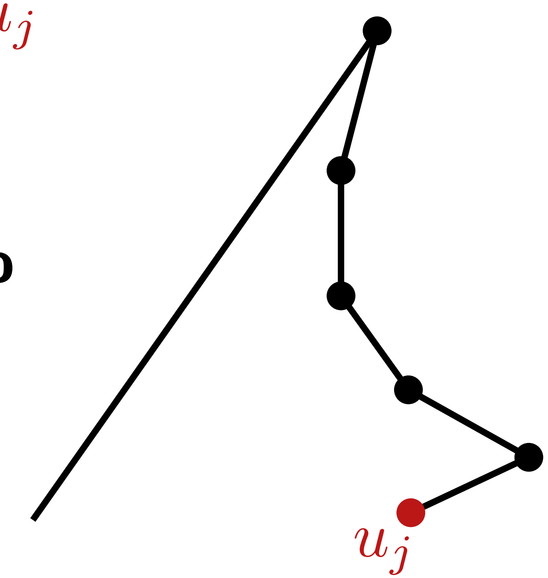


# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push$(u_{j-1})$; $S$.push$(u_j)$
    **else**
        $v \leftarrow S$.pop()
        **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
            $v \leftarrow S$.pop()
            draw diagonal $(u_j, v)$
        $S$.push$(v)$; $S$.push$(u_j)$

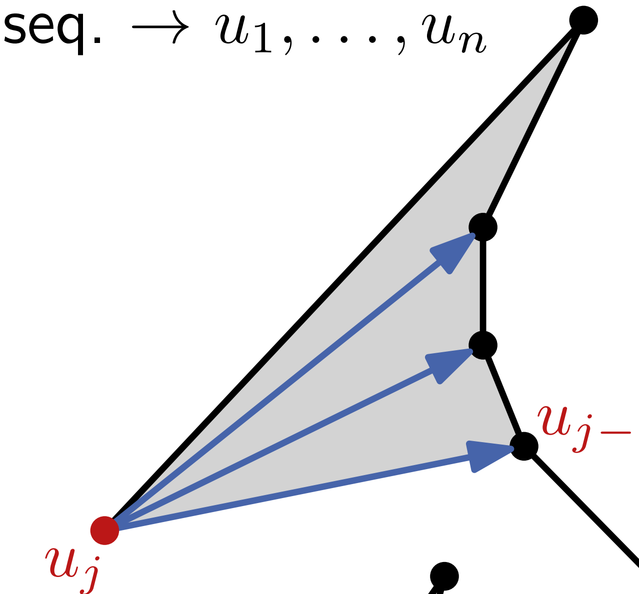


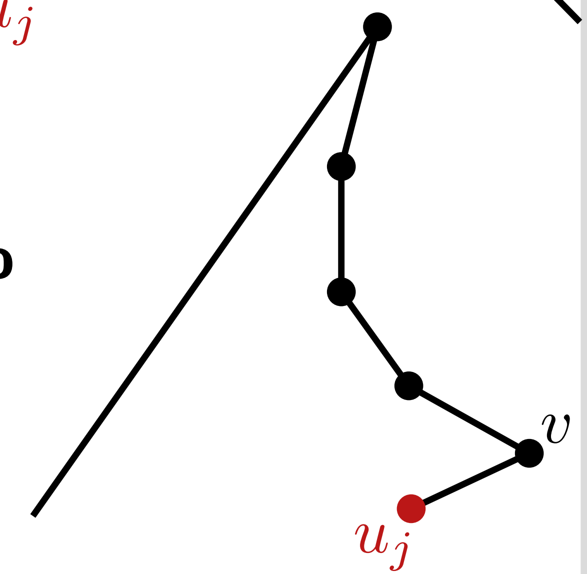



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push$(u_{j-1})$; $S$.push$(u_j)$
    **else**
        $v \leftarrow S$.pop()
        **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
            $v \leftarrow S$.pop()
            draw diagonal $(u_j, v)$
        $S$.push$(v)$; $S$.push$(u_j)$

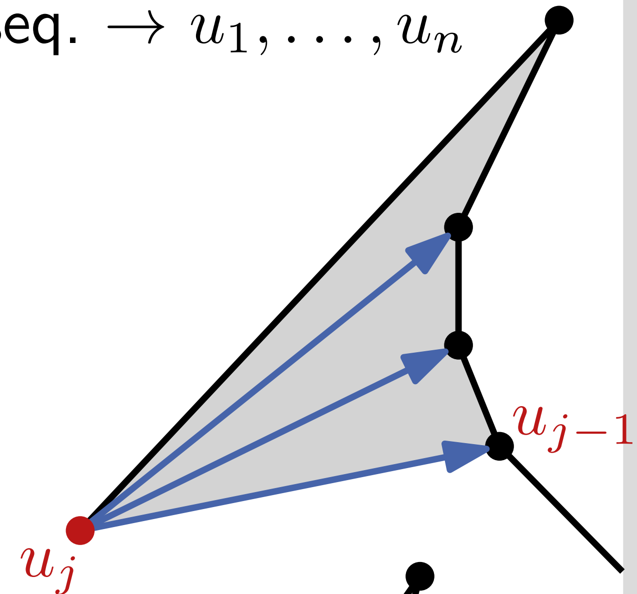


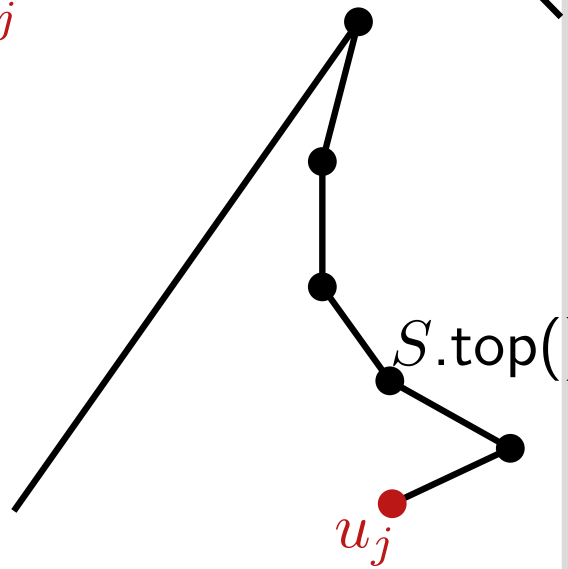



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push$(u_{j-1})$; $S$.push$(u_j)$
  **else**
    $v \leftarrow S$.pop()
    **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
      $v \leftarrow S$.pop()
      draw diagonal $(u_j, v)$
    $S$.push$(v)$; $S$.push$(u_j)$

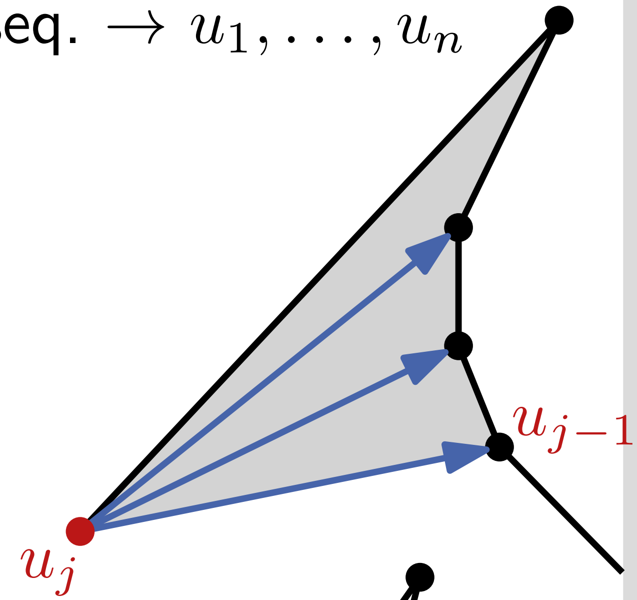


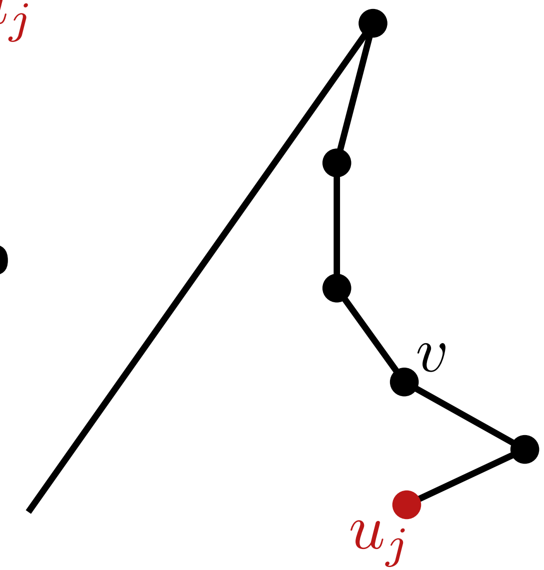



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

> Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
> 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() from different paths **then**
>         **while not** $S$.empty() **do**
>             $v \leftarrow S$.pop()
>             **if not** $S$.empty() **then** draw $(u_j, v)$
>         $S$.push$(u_{j-1})$; $S$.push$(u_j)$
>     **else**
>         $v \leftarrow S$.pop()
>         **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
>             $v \leftarrow S$.pop()
>             draw diagonal $(u_j, v)$
>         $S$.push$(v)$; $S$.push$(u_j)$

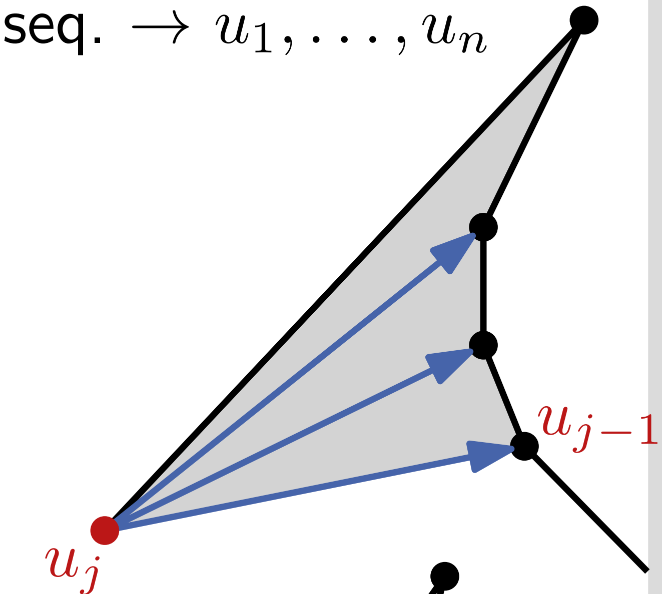


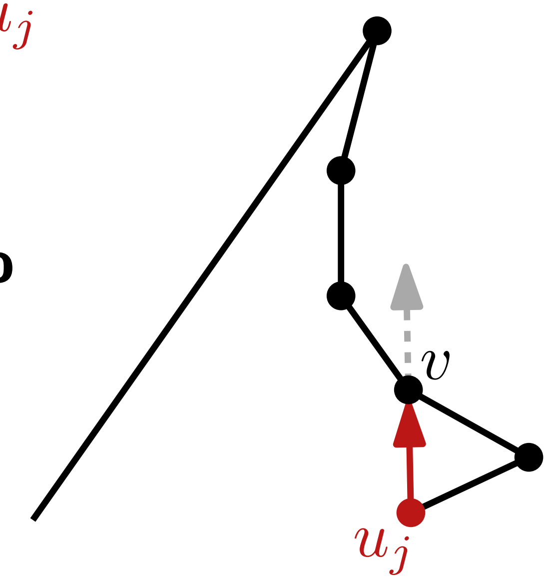



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push$(u_{j-1})$; $S$.push$(u_j)$
  **else**
    $v \leftarrow S$.pop()
    **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
      $v \leftarrow S$.pop()
      draw diagonal $(u_j, v)$
    $S$.push$(v)$; $S$.push$(u_j)$

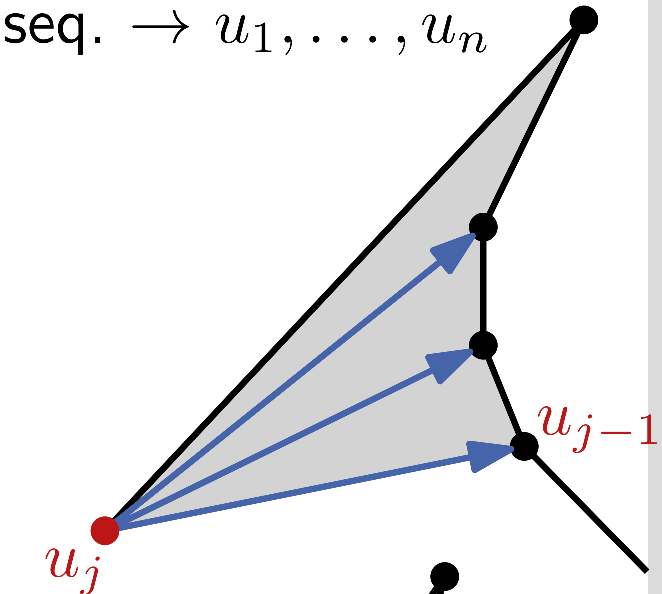


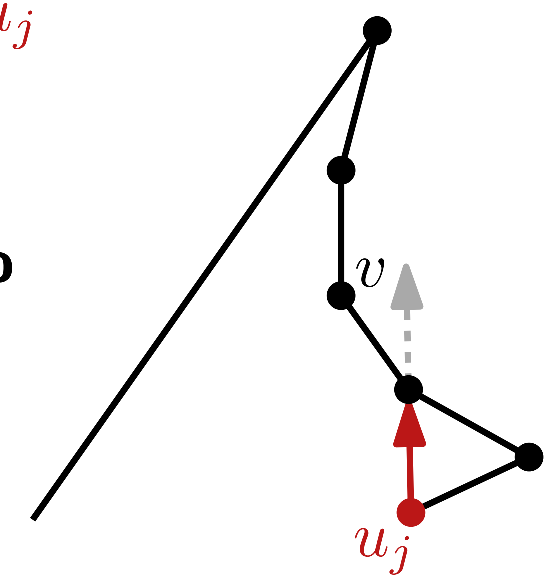



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

- Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \dots, u_n$
- 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() from different paths **then**
    - **while not** $S$.empty() **do**
      - $v \leftarrow S$.pop()
      - **if not** $S$.empty() **then** draw $(u_j, v)$
    - $S$.push$(u_{j-1})$; $S$.push$(u_j)$
  - **else**
    - $v \leftarrow S$.pop()
    - **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
      - $v \leftarrow S$.pop()
      - draw diagonal $(u_j, v)$
    - $S$.push$(v)$; $S$.push$(u_j)$

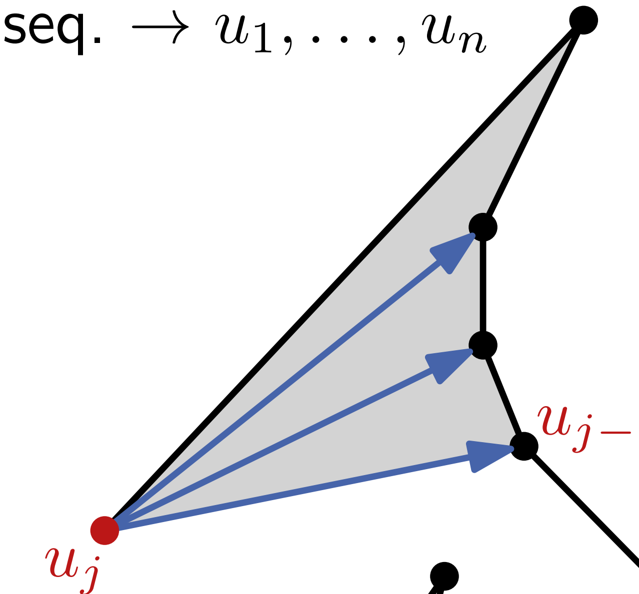


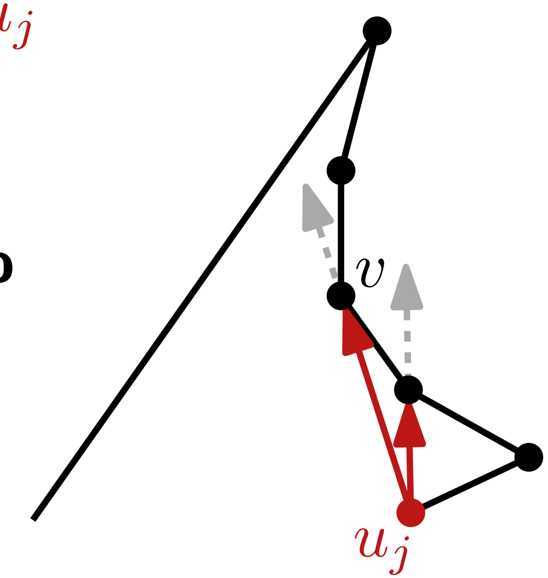



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
    **while not** $S$.empty() **do**
      $v \leftarrow S$.pop()
      **if not** $S$.empty() **then** draw $(u_j, v)$
    $S$.push($u_{j-1}$); $S$.push($u_j$)
  **else**
    $v \leftarrow S$.pop()
    **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
      $v \leftarrow S$.pop()
      draw diagonal $(u_j, v)$
    $S$.push($v$); $S$.push($u_j$)

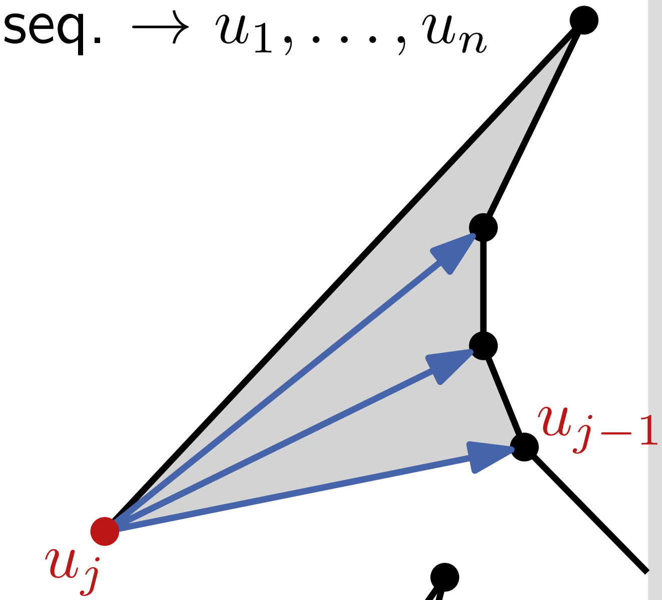


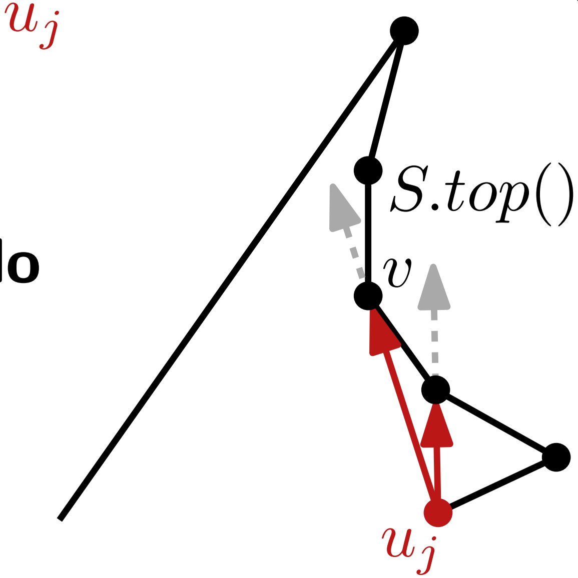



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
  - **while not** $S$.empty() **do**
    - $v \leftarrow S$.pop()
    - **if not** $S$.empty() **then** draw $(u_j, v)$
  - $S$.push$(u_{j-1})$; $S$.push$(u_j)$
- **else**
  - $v \leftarrow S$.pop()
  - **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
    - $v \leftarrow S$.pop()
    - draw diagonal $(u_j, v)$
  - $S$.push$(v)$; $S$.push$(u_j)$

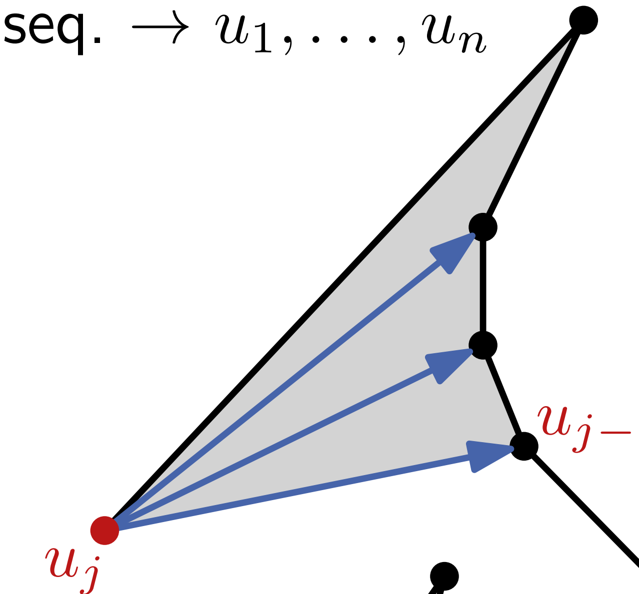


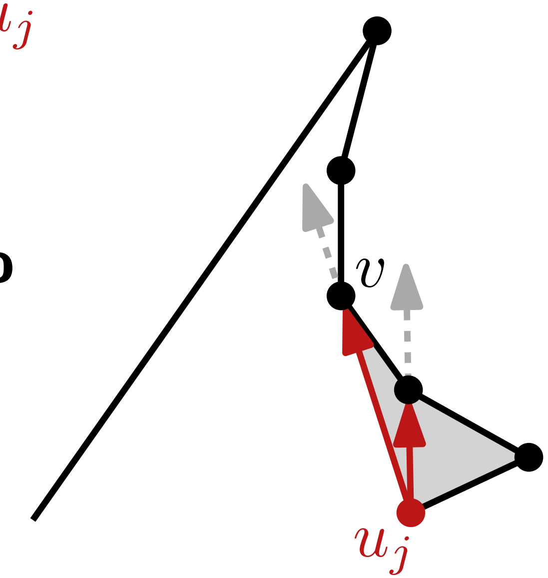



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\to u_1, \ldots, u_n$
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() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push($u_{j-1}$); $S$.push($u_j$)
    **else**
        $v \leftarrow S$.pop()
        **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
            $v \leftarrow S$.pop()
            draw diagonal $(u_j, v)$
        $S$.push($v$); $S$.push($u_j$)

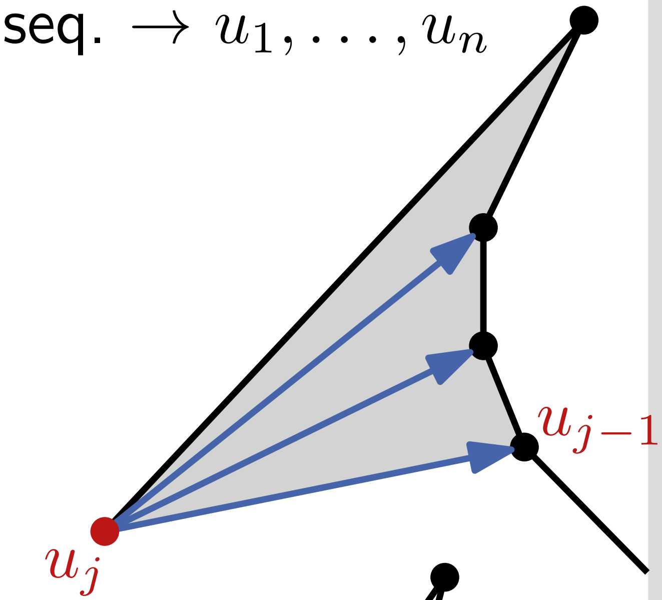


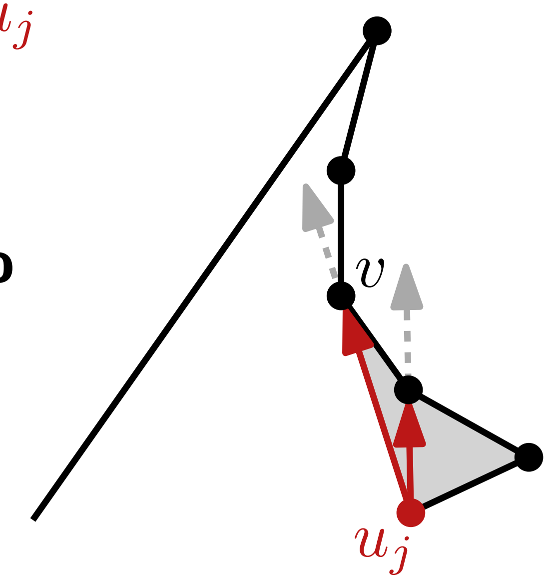



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push($u_{j-1}$); $S$.push($u_j$)
    **else**
        $v \leftarrow S$.pop()
        **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
            $v \leftarrow S$.pop()
            draw diagonal $(u_j, v)$
        $S$.push($v$); $S$.push($u_j$)
Connect $u_n$ to all the vertices in $S$ (except for the first and the last)



# Algorithm TriangulateMonotonePolygon

TriangulateMonotonePolygon(Polygon $P$ as doubly-connected list of edges)

Merge vertices on left and right chains into desc. seq. $\rightarrow u_1, \ldots, u_n$
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() from different paths **then**
        **while not** $S$.empty() **do**
            $v \leftarrow S$.pop()
            **if not** $S$.empty() **then** draw $(u_j, v)$
        $S$.push$(u_{j-1})$; $S$.push$(u_j)$
    **else**
        $v \leftarrow S$.pop()
        **while not** $S$.empty() **and** $u_j$ sees $S$.top() **do**
            $v \leftarrow S$.pop()
            draw diagonal $(u_j, v)$
        $S$.push$(v)$; $S$.push$(u_j)$
Connect $u_n$ to all the vertices in $S$ (except for the first and the last)

**Task:**
What is the running time?

# Summary

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

# Summary

**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 into $y$-monotone polygons in $O(n \log n)$ time and $O(n)$ space.

recall

# Summary

**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 into $y$-monotone polygons in $O(n \log n)$ time and $O(n)$ space.

recall

$\Downarrow$

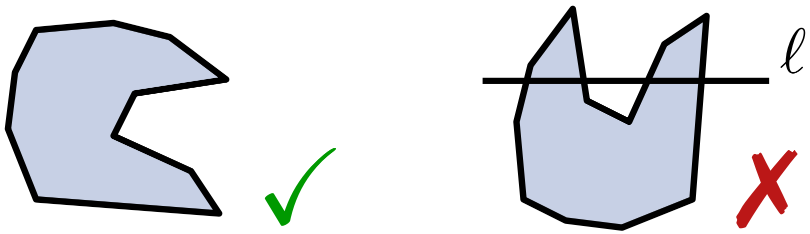
**Theorem 5:** A simple polygon with $n$ vertices can be triangulated in $O(n \log n)$ time and $O(n)$ space.

# Proof of Art-Gallery-Theorem: Overview

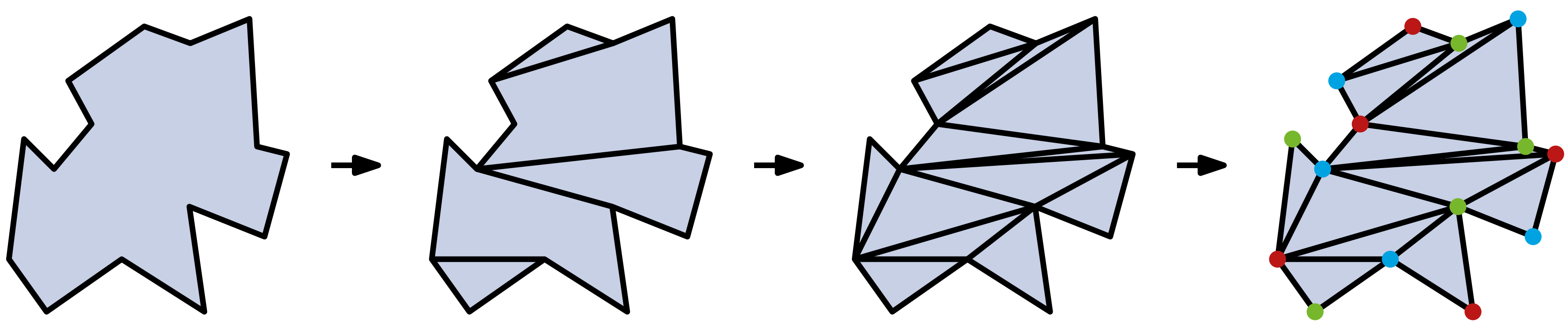
Three-step procedure:

- Step 1: Decompose $P$ in $y$-monotone polygons ✓

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




- Step 2: Triangulate $y$-monotone polygons ✓
- Step 3: use DFS to color the triangulated polygon ✓



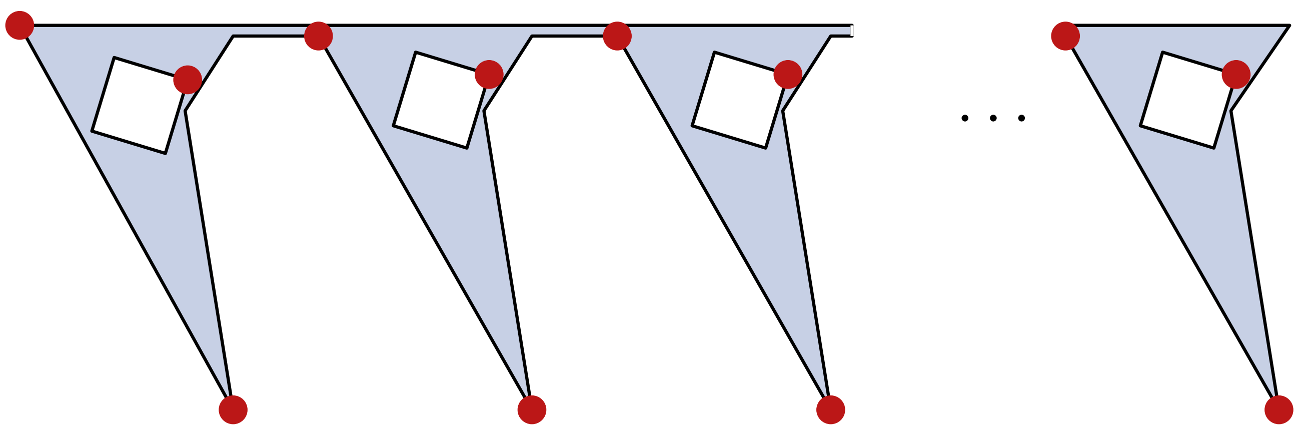
# Discussion

**Can the triangulation algorithm be expanded to work with polygons with holes?**

# Discussion

**Can the triangulation algorithm be expanded to work with polygons with holes?**

- Triangulation: yes
- But are $\lfloor n/3 \rfloor$ cameras still sufficient to guard it?
  No, a generalization of Art-Gallery-Theorems says that $\lfloor (n+h)/3 \rfloor$ cameras are sometimes necessary, and always sufficient, where $h$ is the number of holes. [Hoffmann et al., 1991]



# Discussion

**Can the triangulation algorithm be expanded to work with polygons with holes?**

- Triangulation: yes
- But are $\lfloor n/3 \rfloor$ cameras still sufficient to guard it?
  No, a generalization of Art-Gallery-Theorems says that $\lfloor (n+h)/3 \rfloor$ cameras are sometimes necessary, and always sufficient, where $h$ is the number of holes. [Hoffmann et al., 1991]

**Can we solve the triangulation problem faster for simple polygons?**

# Discussion

## Can the triangulation algorithm be expanded to work with polygons with holes?

- Triangulation: yes
- But are $\lfloor n/3 \rfloor$ cameras still sufficient to guard it?
  No, a generalization of Art-Gallery-Theorems says that $\lfloor (n+h)/3 \rfloor$ cameras are sometimes necessary, and always sufficient, where $h$ is the number of holes. [Hoffmann et al., 1991]

## Can we solve the triangulation problem faster for simple polygons?

Yes. The question whether it is possible was open for more than a decade. In the end of 80's a faster randomized algorithm was given, and in 1990 Chazelle presented a deterministic linear-time algorithm (complicated).