

# Polygon Triangulation

1390-2



# Motivation:

## The Art Gallery Problem



Van Gogh

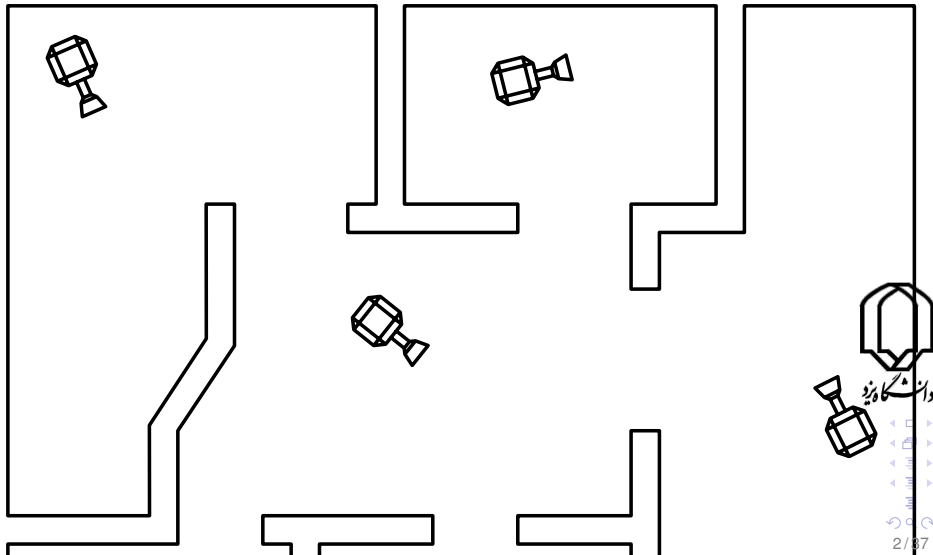


Van Gogh



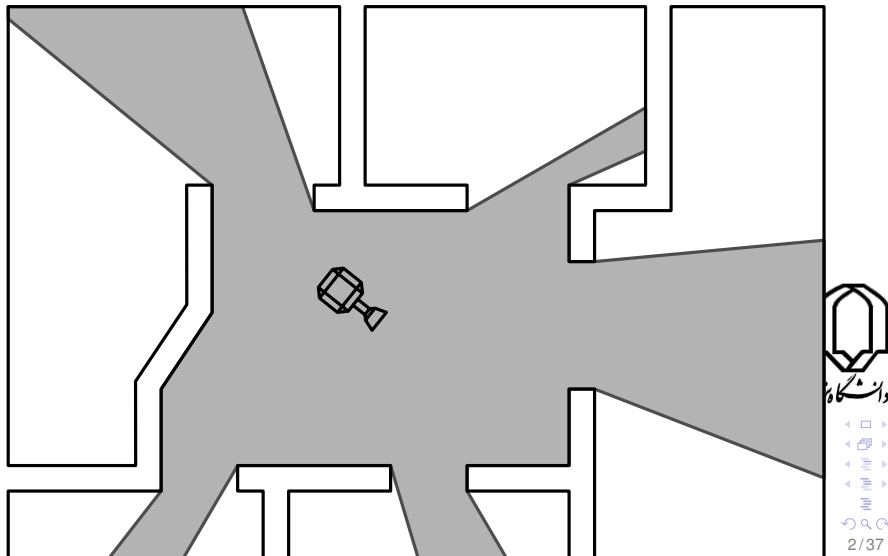
# Motivation:

## The Art Gallery Problem



# Motivation:

## The Art Gallery Problem



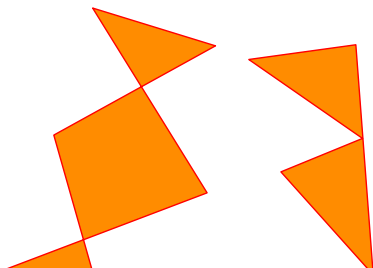
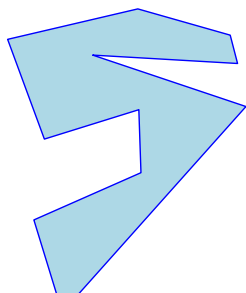
دانشگاه تهران



# Triangulating Polygons

## Definitions

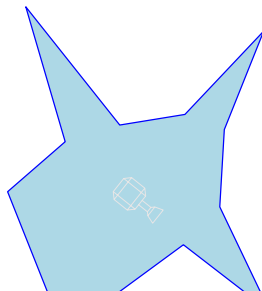
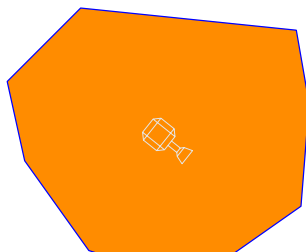
- Simple polygon: Regions enclosed by a single closed polygonal chain that does not intersect itself.
- Question: How many cameras do we need to guard a simple polygon?  
Answer: Depends on the polygon.
- One solution: Decompose the polygon to parts which are simple to guard.



# Triangulating Polygons

## Definitions

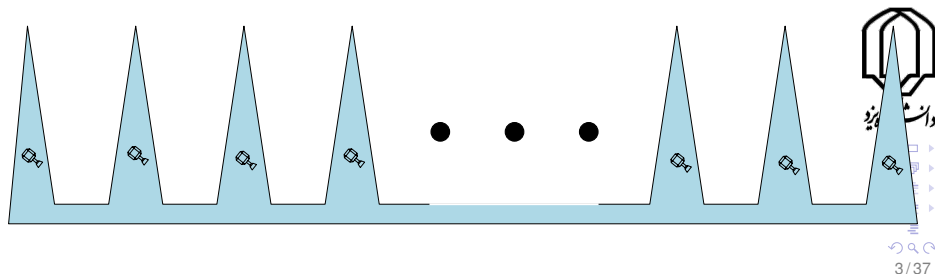
- Simple polygon: Regions enclosed by a single closed polygonal chain that does not intersect itself.
- Question: How many cameras do we need to guard a simple polygon?  
Answer: Depends on the polygon.
- One solution: Decompose the polygon to parts which are simple to guard.



# Triangulating Polygons

## Definitions

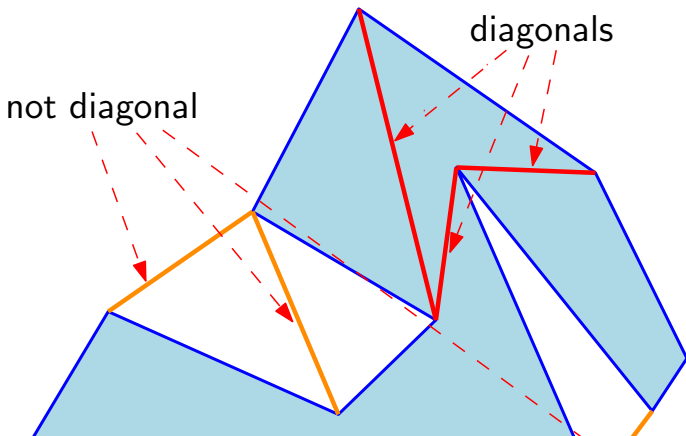
- Simple polygon: Regions enclosed by a single closed polygonal chain that does not intersect itself.
- Question: How many cameras do we need to guard a simple polygon?  
Answer: Depends on the polygon.
- One solution: Decompose the polygon to parts which are simple to guard.



# Triangulating Polygons

## Definitions

- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.

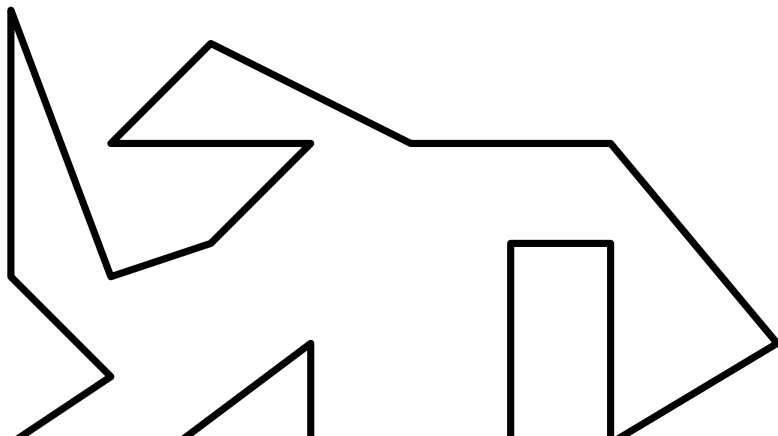




# Triangulating Polygons

## Definitions

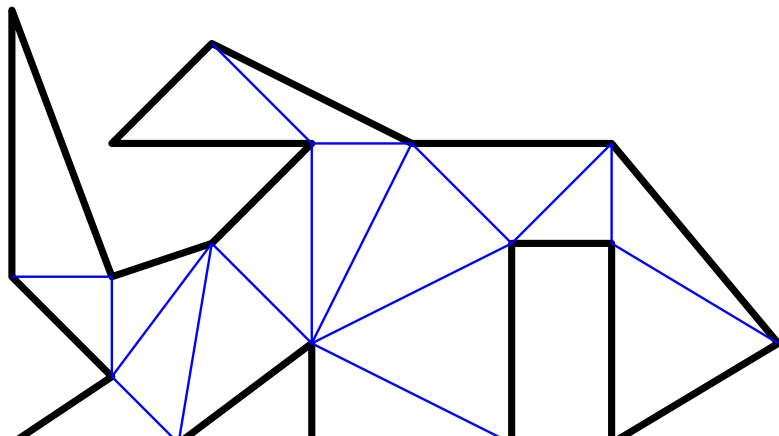
- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



# Triangulating Polygons

## Definitions

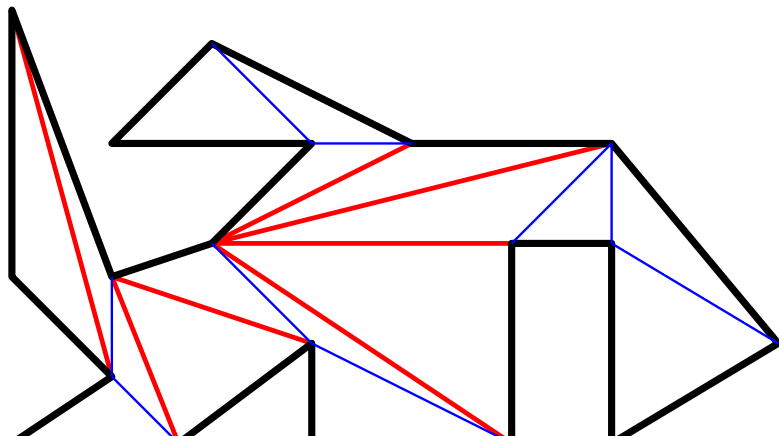
- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



# Triangulating Polygons

## Definitions

- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



# Triangulating Polygons

## Definitions

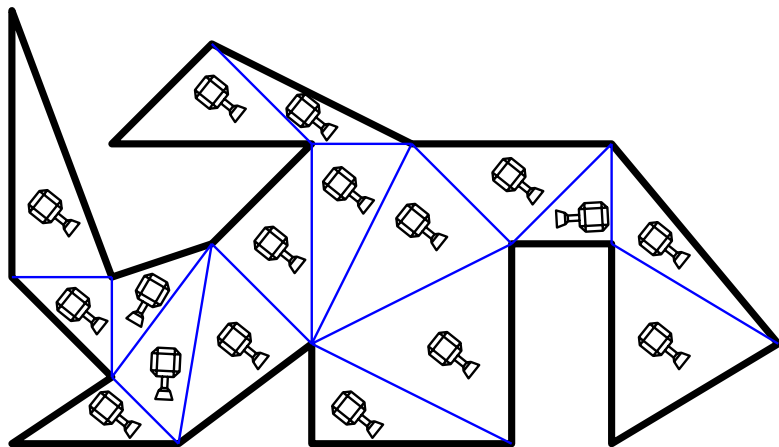
- Guarding after triangulation:



# Triangulating Polygons

## Definitions

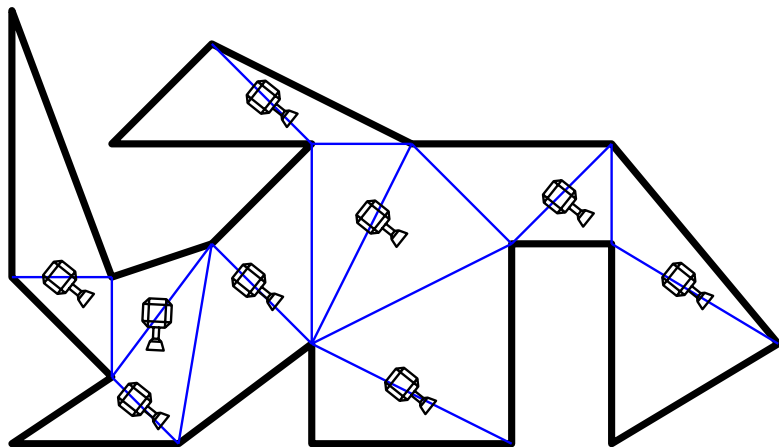
- Guarding after triangulation:



# Triangulating Polygons

## Definitions

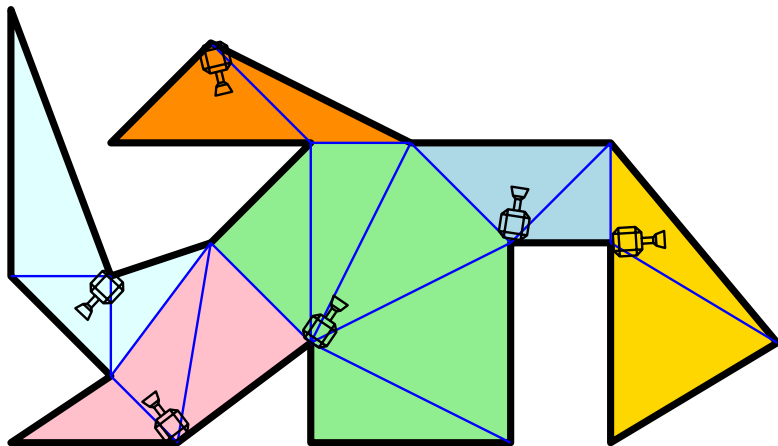
- Guarding after triangulation:



# Triangulating Polygons

## Definitions

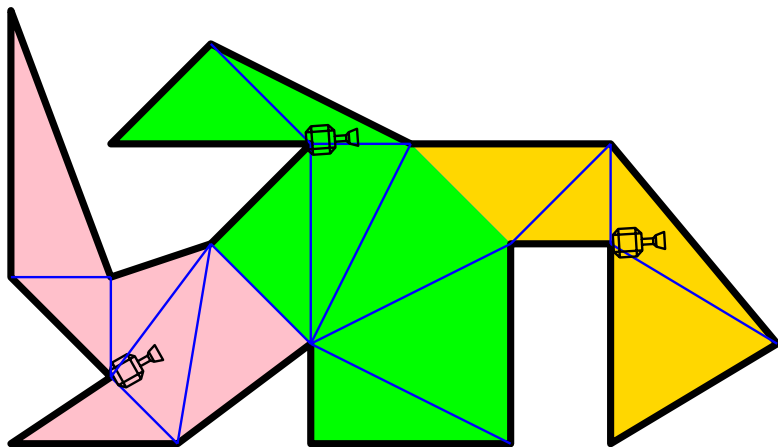
- Guarding after triangulation:



# Triangulating Polygons

## Definitions

- Guarding after triangulation:





## Questions:

- Does a triangulation always exist?
- How many triangles can there be in a triangulation?

## Theorem 3.1

Every simple polygon admits a triangulation, and any triangulation of a simple polygon with  $n$  vertices consists of exactly  $n - 2$  triangles.

**Proof.** By induction.



## Questions:

- Does a triangulation always exist?
- How many triangles can there be in a triangulation?

## Theorem 3.1

Every simple polygon admits a triangulation, and any triangulation of a simple polygon with  $n$  vertices consists of exactly  $n - 2$  triangles.

**Proof.** By induction.



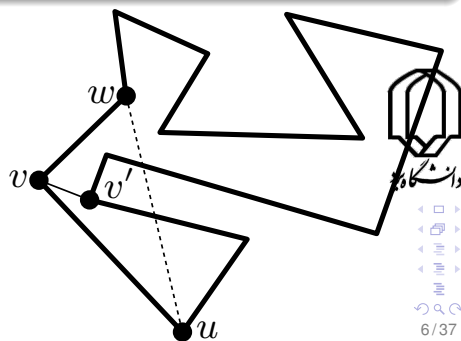
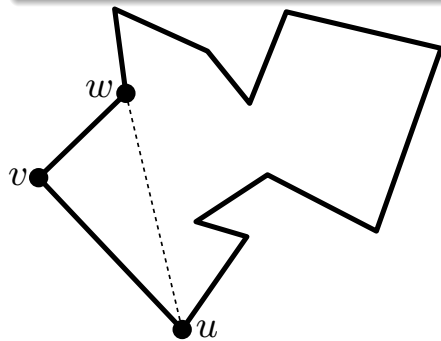
## Questions:

- Does a triangulation always exist?
- How many triangles can there be in a triangulation?

### Theorem 3.1

Every simple polygon admits a triangulation, and any triangulation of a simple polygon with  $n$  vertices consists of exactly  $n - 2$  triangles.

**Proof.** By induction.



# Guarding a triangulated polygon

- $\mathcal{T}_P$ : A triangulation of a simple polygon  $P$ .
- Select  $S \subseteq$  the vertices of  $P$ , such that any triangle in  $\mathcal{T}_P$  has at least one vertex in  $S$ , and place the cameras at vertices in  $S$ .
- To find such a subset: find a 3-coloring of a triangulated polygon.
- In a 3-coloring of  $\mathcal{T}_P$ , every triangle has a blue, a red, and a black vertex. Hence, if we place cameras at all red vertices, we have guarded the whole polygon.
- By choosing the smallest color class to place the cameras, we can guard  $P$  using at most  $\lfloor n/3 \rfloor$  cameras.



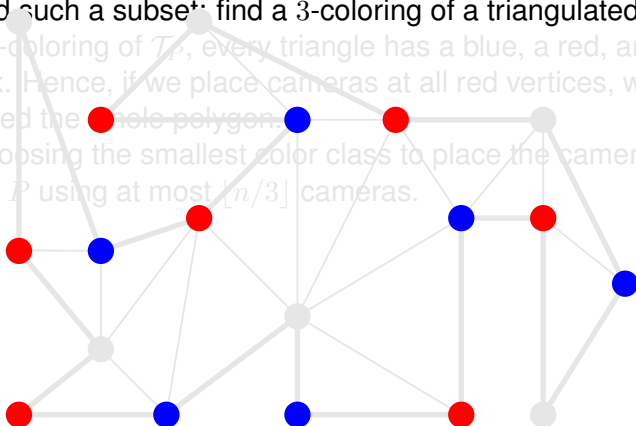
# Guarding a triangulated polygon

- $\mathcal{T}_P$ : A triangulation of a simple polygon  $P$ .
- Select  $S \subseteq$  the vertices of  $P$ , such that any triangle in  $\mathcal{T}_P$  has at least one vertex in  $S$ , and place the cameras at vertices in  $S$ .
- To find such a subset: find a 3-coloring of a triangulated polygon.
- In a 3-coloring of  $\mathcal{T}_P$ , every triangle has a blue, a red, and a black vertex. Hence, if we place cameras at all red vertices, we have guarded the whole polygon.
- By choosing the smallest color class to place the cameras, we can guard  $P$  using at most  $\lfloor n/3 \rfloor$  cameras.



# Guarding a triangulated polygon

- $\mathcal{T}_P$ : A triangulation of a simple polygon  $P$ .
- Select  $S \subseteq$  the vertices of  $P$ , such that any triangle in  $\mathcal{T}_P$  has at least one vertex in  $S$ , and place the cameras at vertices in  $S$ .
- To find such a subset: find a 3-coloring of a triangulated polygon.
- In a 3-coloring of  $\mathcal{T}_P$ , every triangle has a blue, a red, and a black vertex. Hence, if we place cameras at all red vertices, we have guarded the whole polygon.
- By choosing the smallest color class to place the cameras, we can guard  $P$  using at most  $\lfloor n/3 \rfloor$  cameras.



# Guarding a triangulated polygon

- $\mathcal{T}_P$ : A triangulation of a simple polygon  $P$ .
- Select  $S \subseteq$  the vertices of  $P$ , such that any triangle in  $\mathcal{T}_P$  has at least one vertex in  $S$ , and place the cameras at vertices in  $S$ .
- To find such a subset: find a 3-coloring of a triangulated polygon.
- In a 3-coloring of  $\mathcal{T}_P$ , every triangle has a blue, a red, and a black vertex. Hence, if we place cameras at all red vertices, we have guarded the whole polygon.
- By choosing the smallest color class to place the cameras, we can guard  $P$  using at most  $\lfloor n/3 \rfloor$  cameras.



# Guarding a triangulated polygon

- $\mathcal{T}_P$ : A triangulation of a simple polygon  $P$ .
- Select  $S \subseteq$  the vertices of  $P$ , such that any triangle in  $\mathcal{T}_P$  has at least one vertex in  $S$ , and place the cameras at vertices in  $S$ .
- To find such a subset: find a 3-coloring of a triangulated polygon.
- In a 3-coloring of  $\mathcal{T}_P$ , every triangle has a blue, a red, and a black vertex. Hence, if we place cameras at all red vertices, we have guarded the whole polygon.
- By choosing the smallest color class to place the cameras, we can guard  $P$  using at most  $\lfloor n/3 \rfloor$  cameras.

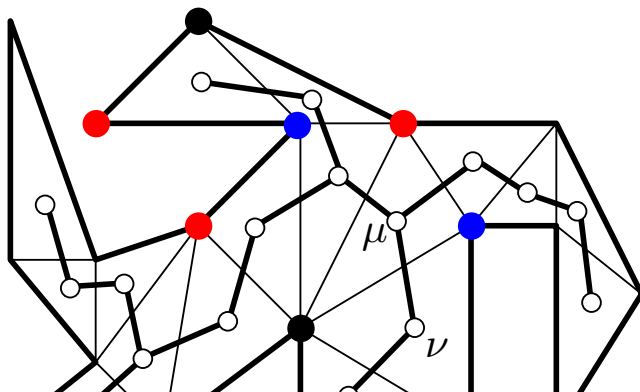




# Does a 3-coloring always exist?

## Dual graph:

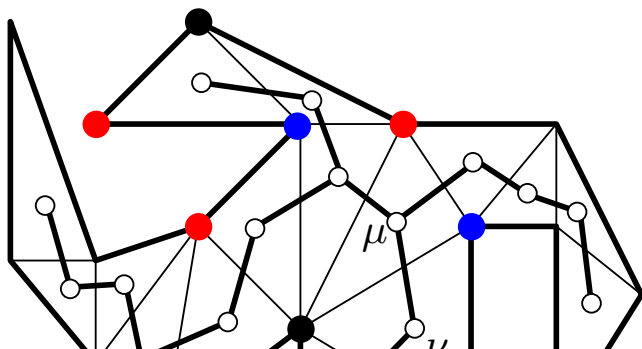
- This graph  $\mathcal{G}(\mathcal{T}_P)$  has a node for every triangle in  $\mathcal{T}_P$ .
- There is an arc between two nodes  $\nu$  and  $\mu$  if  $t(\nu)$  and  $t(\mu)$  share a diagonal.
- $\mathcal{G}(\mathcal{T}_P)$  is a tree.



# Does a 3-coloring always exist?

For 3-coloring:

- Traverse the dual graph (DFS).
- Invariant: so far everything is nice.
- Start from any node of  $\mathcal{G}(\mathcal{T}_P)$ ; color the vertices.
- When we reach a node  $\nu$  in  $\mathcal{G}$ , coming from node  $\mu$ . Only one vertex of  $t(\nu)$  remains to be colored.

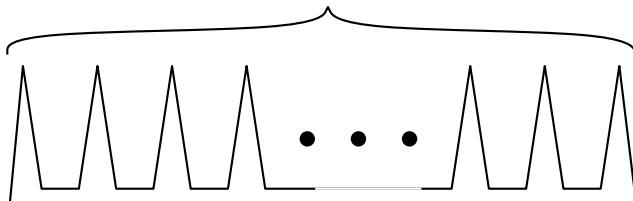


# Art Gallery Theorem

## Theorem 3.2 (Art Gallery Theorem)

For a simple polygon with  $n$  vertices,  $\lfloor n/3 \rfloor$  cameras are occasionally necessary and always sufficient to have every point in the polygon visible from at least one of the cameras.

$\lfloor n/3 \rfloor$  prongs



# Art Gallery Theorem

We will show:

How to compute a triangulation in  $\mathcal{O}(n \log n)$  time.

Therefore:

## Theorem 3.3

Let  $P$  be a simple polygon with  $n$  vertices. A set of  $\lfloor n/3 \rfloor$  camera positions in  $P$  such that any point inside  $P$  is visible from at least one of the cameras can be computed in  $\mathcal{O}(n \log n)$  time.



How can we compute a triangulation of a given polygon?



## Triangulation algorithms

- A really naive algorithm: check all  $\binom{n}{2}$  choices for a diagonal, each takes  $\mathcal{O}(n)$  time. Time complexity:  $\mathcal{O}(n^3)$ .
- A better naive algorithm: find an ear in  $\mathcal{O}(n)$  time, then recurse. Total time:  $\mathcal{O}(n^2)$ .
- First non-trivial algorithm:  $\mathcal{O}(n \log n)$  (1978).
- A long series of papers and algorithms in 80s until Chazelle produced an optimal  $\mathcal{O}(n)$  algorithm in 1991.
- Linear time algorithm insanely complicated; there are randomized, expected linear time that are more accessible.
- Here we present a  $\mathcal{O}(n \log n)$  algorithm.



# Algorithm Outline

## Algorithm Outline

- 1 Partition polygon into monotone polygons.
- 2 Triangulate each monotone piece.



# Monotone polygon

## $\ell$ -monotone polygon

$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).

### Definition:

- A point  $p$  is **below** another point  $q$  if  $p_y < q_y$  or  $p_y = q_y$  and  $p_x > q_x$ .
- $p$  is **above**  $q$  if  $p_y > q_y$  or  $p_y = q_y$  and  $p_x < q_x$ .

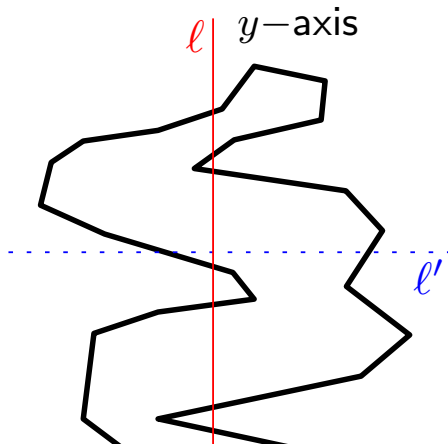
دانشگاه



# Monotone polygon

## $\ell$ -monotone polygon

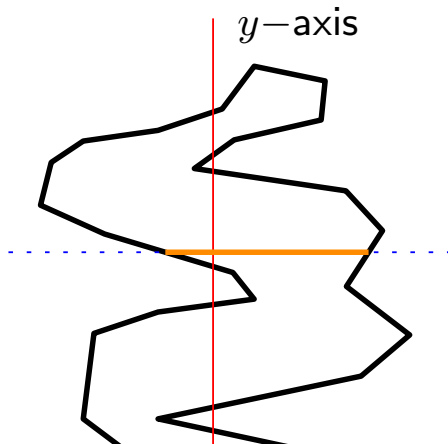
$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell$  is connected (a line segment, a point, or empty).



# Monotone polygon

## $\ell$ -monotone polygon

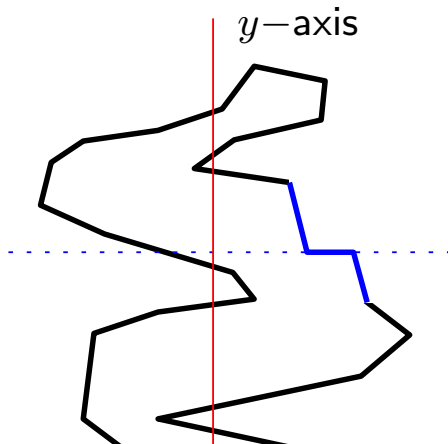
$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).



# Monotone polygon

## $\ell$ -monotone polygon

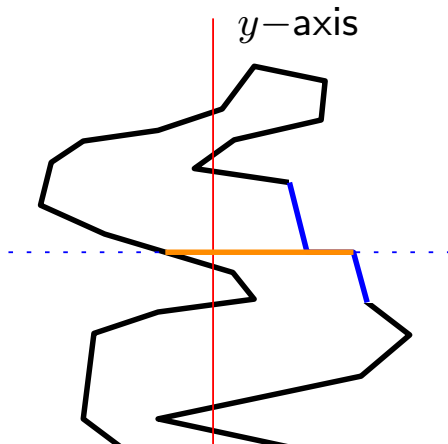
$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).



# Monotone polygon

## $\ell$ -monotone polygon

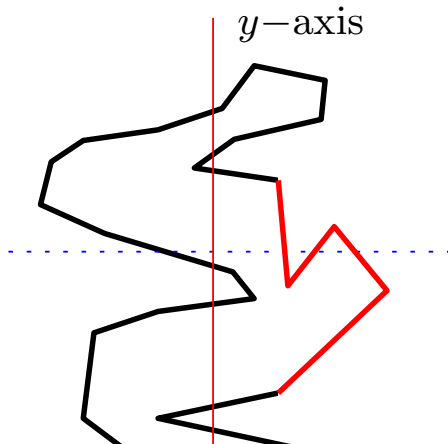
$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).



# Monotone polygon

## $\ell$ -monotone polygon

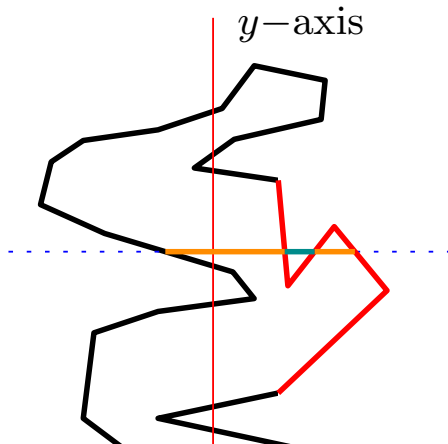
$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).



# Monotone polygon

## $\ell$ -monotone polygon

$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).



# Monotone polygon

## $\ell$ -monotone polygon

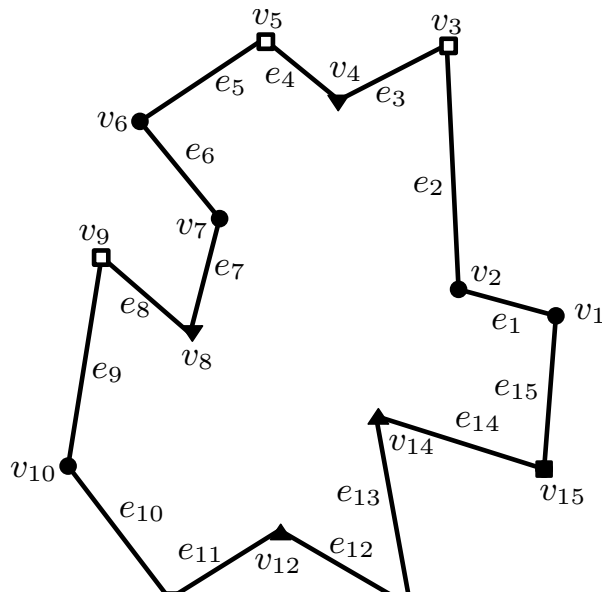
$P$  is called monotone w. r. t.  $\ell$  if  $\forall \ell'$  perpendicular to  $\ell$  the intersection of  $P$  with  $\ell'$  is connected (a line segment, a point, or empty).

## Definition:

- A point  $p$  is **below** another point  $q$  if  $p_y < q_y$  or  $p_y = q_y$  and  $p_x > q_x$ .
- $p$  is **above**  $q$  if  $p_y > q_y$  or  $p_y = q_y$  and  $p_x < q_x$ .

دانشگاه

# Partition $P$ into monotone pieces



□ = start vertex

■ = end vertex

● = regular vertex

▲ = split vertex

▼ = merge vertex





# Partition $\mathcal{P}$ into monotone pieces

## Lemma 3.4

$P$  is  $y$ -monotone if it has no split or merge vertices.

**Proof.** Assume  $\mathcal{P}$  is not  $y$ -monotone.

$P$  has been partitioned into  $y$ -monotone pieces once we get rid of its split and merge vertices.



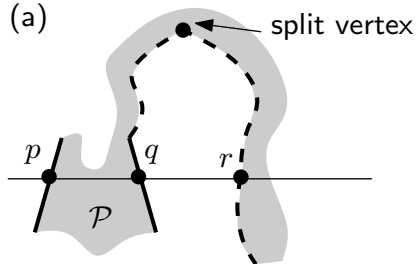
# Partition $\mathcal{P}$ into monotone pieces

## Lemma 3.4

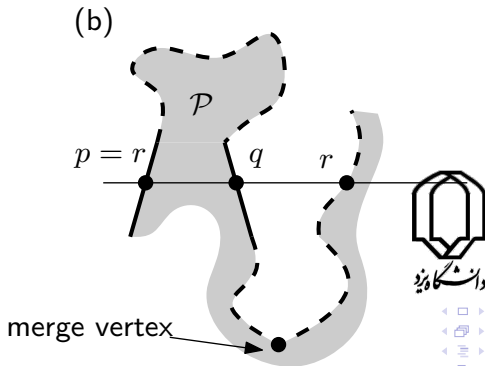
$P$  is  $y$ -monotone if it has no split or merge vertices.

**Proof.** Assume  $\mathcal{P}$  is not  $y$ -monotone.

(a)



(b)



$P$  has been partitioned into  $y$ -monotone pieces once we get rid of its

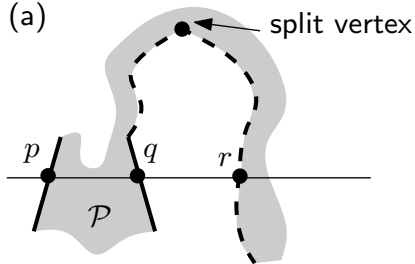
# Partition $\mathcal{P}$ into monotone pieces

## Lemma 3.4

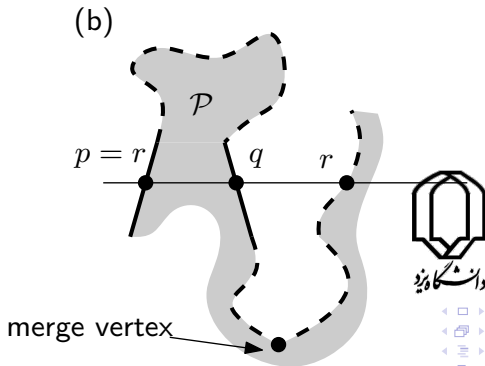
$P$  is  $y$ -monotone if it has no split or merge vertices.

**Proof.** Assume  $\mathcal{P}$  is not  $y$ -monotone.

(a)



(b)



$P$  has been partitioned into  $y$ -monotone pieces once we get rid of its

# Removing split/merge vertices:

## Removing split vertices:

- A sweep line algorithm. Events: all the points
- Goal: To add diagonals from each split vertex to a vertex lying above it.
- $helper(e_j)$ : The lowest vertex above the sweep line such that the horizontal segment connecting the vertex to  $e_j$  lies inside  $P$ .



# Removing split/merge vertices:

## Removing split vertices:

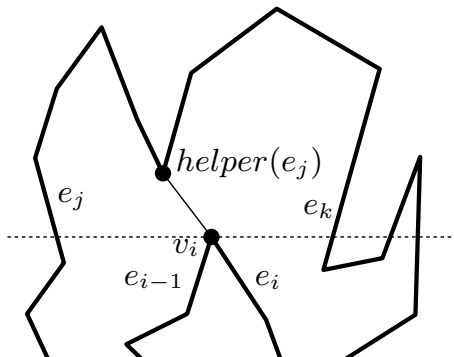
- A sweep line algorithm. Events: all the points
- Goal: To add diagonals from each split vertex to a vertex lying above it.
- $helper(e_j)$ : The lowest vertex above the sweep line such that the horizontal segment connecting the vertex to  $e_j$  lies inside  $P$ .



# Removing split/merge vertices:

## Removing split vertices:

- A sweep line algorithm. Events: all the points
- Goal: To add diagonals from each split vertex to a vertex lying above it.
- $helper(e_j)$ : The lowest vertex above the sweep line such that the horizontal segment connecting the vertex to  $e_j$  lies inside  $P$ .



# Removing split/merge vertices:

## Removing merge vertices:

- Connect each merge vertex to the highest vertex below the sweep line in between  $e_j$  and  $e_k$ .
- But we do not know the point.
- When we reach a vertex  $v_m$  that replaces the helper of  $e_j$ , then this is the vertex we are looking for.



# Removing split/merge vertices:

## Removing merge vertices:

- Connect each merge vertex to the highest vertex below the sweep line in between  $e_j$  and  $e_k$ .
- But we do not know the point.
- When we reach a vertex  $v_m$  that replaces the helper of  $e_j$ , then this is the vertex we are looking for.

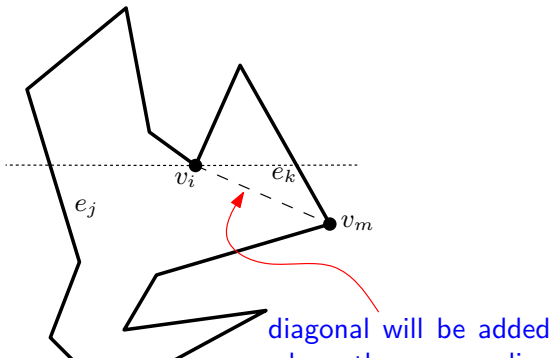




# Removing split/merge vertices:

## Removing merge vertices:

- Connect each merge vertex to the highest vertex below the sweep line in between  $e_j$  and  $e_k$ .
- But we do not know the point.
- When we reach a vertex  $v_m$  that replaces the helper of  $e_j$ , then this is the vertex we are looking for.



# Data Structure:

For this approach, we need to find the edge to the left of each vertex.  
To do that:

- 1 We store the edges of  $P$  intersecting the sweep line in the leaves of a dynamic binary search tree  $\mathcal{T}$ .
- 2 Because we are only interested in edges to the left of split and merge vertices we only need to store edges in  $\mathcal{T}$  that have the interior of  $P$  to their right.
- 3 With each edge in  $\mathcal{T}$  we store its helper.
- 4 We store  $P$  in DCEL form and make changes such that it remains valid.



# Data Structure:

For this approach, we need to find the edge to the left of each vertex.  
To do that:

- 1 We store the edges of  $P$  intersecting the sweep line in the leaves of a dynamic binary search tree  $\mathcal{T}$ .
- 2 Because we are only interested in edges to the left of split and merge vertices we only need to store edges in  $\mathcal{T}$  that have the interior of  $P$  to their right.
- 3 With each edge in  $\mathcal{T}$  we store its helper.
- 4 We store  $P$  in DCEL form and make changes such that it remains valid.



# Data Structure:

For this approach, we need to find the edge to the left of each vertex.  
To do that:

- 1 We store the edges of  $P$  intersecting the sweep line in the leaves of a dynamic binary search tree  $\mathcal{T}$ .
- 2 Because we are only interested in edges to the left of split and merge vertices we only need to store edges in  $\mathcal{T}$  that have the interior of  $P$  to their right.
- 3 With each edge in  $\mathcal{T}$  we store its helper.
- 4 We store  $P$  in DCEL form and make changes such that it remains valid.



# Data Structure:

For this approach, we need to find the edge to the left of each vertex.  
To do that:

- 1 We store the edges of  $P$  intersecting the sweep line in the leaves of a dynamic binary search tree  $\mathcal{T}$ .
- 2 Because we are only interested in edges to the left of split and merge vertices we only need to store edges in  $\mathcal{T}$  that have the interior of  $P$  to their right.
- 3 With each edge in  $\mathcal{T}$  we store its helper.
- 4 We store  $P$  in DCEL form and make changes such that it remains valid.



# Make Monotone Algorithm:

**Algorithm** MAKEMONOTONE( $P$ )

**Input:** A simple polygon  $P$  stored in a DCEL  $\mathcal{D}$ .

**Output:** A partitioning of  $P$  into monotone subpolygons, stored in  $\mathcal{D}$ .

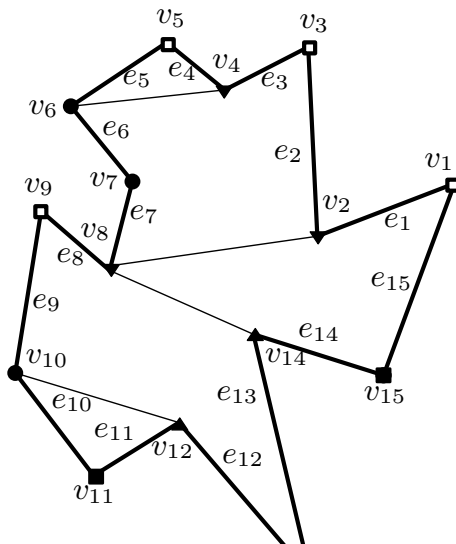
1. Construct a priority queue  $Q$  on the vertices of  $P$ , using their  $y$ -coordinates as priority. If two points have the same  $y$ -coordinate, the one with smaller  $x$ -coordinate has higher priority.
2. Initialize an empty binary search tree  $\mathcal{T}$ .
3. **while**  $Q$  is not empty
4.     Remove the vertex  $v_i$  with the highest priority from  $Q$ .
5.     Call the appropriate procedure to handle the vertex, depending on its type.



# Make Monotone Algorithm:

**Algorithm** HANDLESTARTVERTEX( $v_i$ )

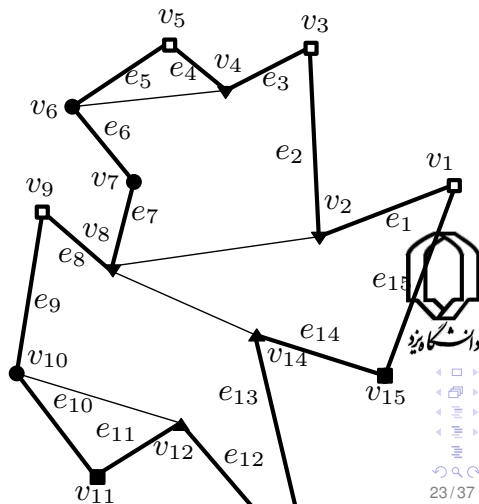
1. Insert  $e_i$  in  $\mathcal{T}$  and set  $helper(e_i)$  to  $v_i$ .



# Make Monotone Algorithm:

**Algorithm** HANDLEENDVERTEX( $v_i$ )

1. **if**  $helper(e_{i-1})$  is a merge vertex
2.     **then** Insert the diagonal connecting  $v_i$  to  $helper(e_{i-1})$  in  $\mathcal{D}$ .
3.     Delete  $e_{i-1}$  from  $\mathcal{T}$ .

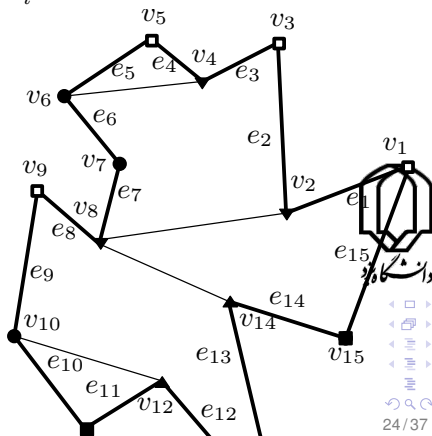




# Make Monotone Algorithm:

## Algorithm HANDLE\_SPLITVERTEX( $v_i$ )

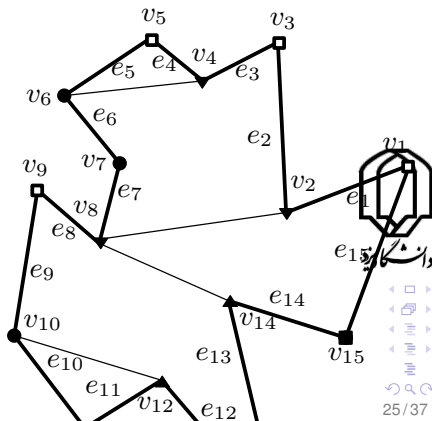
1. Search in  $\mathcal{T}$  to find the edge  $e_j$  directly left of  $v_i$ .
2. Insert the diagonal connecting  $v_i$  to  $helper(e_j)$  in  $\mathcal{D}$ .
3.  $helper(e_j) \leftarrow v_i$ .
4. Insert  $e_i$  in  $\mathcal{T}$  and set  $helper(e_i)$  to  $v_i$ .



# Make Monotone Algorithm:

**Algorithm** HANDLEMERGEVERTEX( $v_i$ )

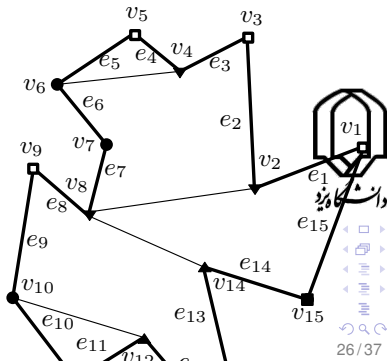
1. **if**  $\text{helper}(e_{i-1})$  is a merge vertex
2.     **then** Insert the diag.  $v_i$  to  $\text{helper}(e_{i-1})$  in  $\mathcal{D}$ .
3. Delete  $e_{i-1}$  from  $\mathcal{T}$ .
4. Search in  $\mathcal{T}$  to find  $e_j$  directly left of  $v_i$ .
5. **if**  $\text{helper}(e_j)$  is a merge vertex
6.     **then** Insert the diag.  $v_i$  to  $\text{helper}(e_j)$  in  $\mathcal{D}$ .
7.  $\text{helper}(e_j) \leftarrow v_i$ .



# Make Monotone Algorithm:

**Algorithm** HANDLEREGULARVERTEX( $v_i$ )

1. **if** the interior of  $P$  lies to the right of  $v_i$
2.     **then if**  $\text{helper}(e_{i-1})$  is a merge vertex
3.         **then** Insert the diag.  $v_i$  to  $\text{helper}(e_{i-1})$  in  $\mathcal{D}$ .
4.         Delete  $e_{i-1}$  from  $\mathcal{T}$ .
5.         Insert  $e_i$  in  $\mathcal{T}$  and set  $\text{helper}(e_i)$  to  $v_i$ .
6.     **else** Search in  $\mathcal{T}$  to find  $e_j$  directly left of  $v_i$ .
7.         **if**  $\text{helper}(e_j)$  is a merge vertex
8.             **then** Insert the diag.  $v_i$  to  $\text{helper}(e_j)$  in  $\mathcal{D}$ .
9.          $\text{helper}(e_j) \leftarrow v_i$

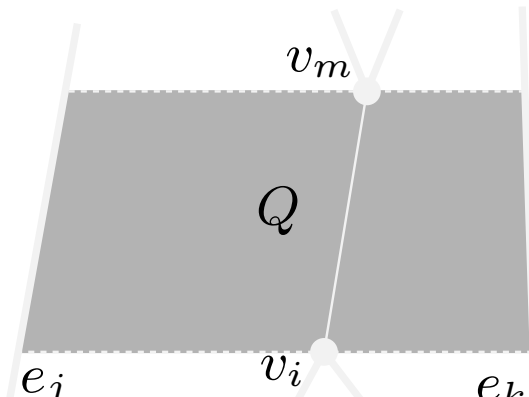


### Lemma 3.5

Algorithm MAKEMONOTONE adds a set of non-intersecting diagonals that partitions  $P$  into monotone subpolygons.

**Proof.** (For split vertices) (other cases are similar)

- No intersection between  $v_i v_m$  and edges of  $P$ .
- No intersection between  $v_i v_m$  and previous edges.

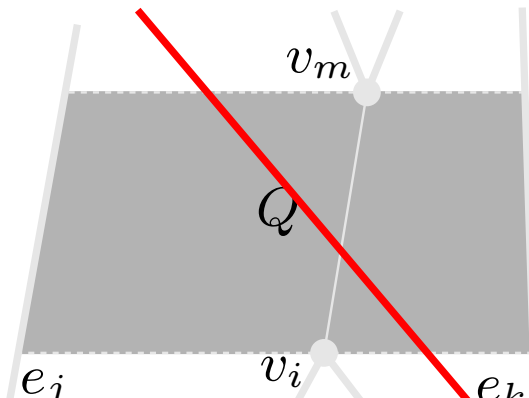


### Lemma 3.5

Algorithm MAKEMONOTONE adds a set of non-intersecting diagonals that partitions  $P$  into monotone subpolygons.

**Proof.** (For split vertices) (other cases are similar)

- No intersection between  $v_i v_m$  and edges of  $P$ .
- No intersection between  $v_i v_m$  and previous edges.

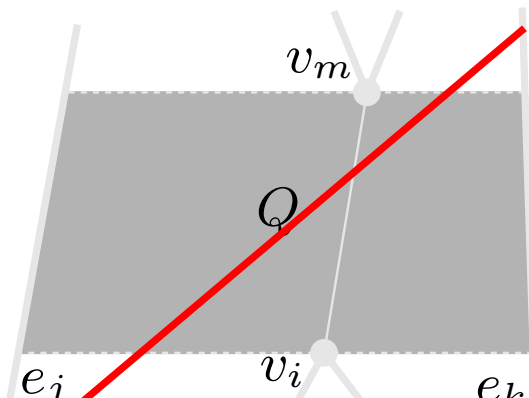


### Lemma 3.5

Algorithm MAKEMONOTONE adds a set of non-intersecting diagonals that partitions  $P$  into monotone subpolygons.

**Proof.** (For split vertices) (other cases are similar)

- No intersection between  $v_i v_m$  and edges of  $P$ .
- No intersection between  $v_i v_m$  and previous edges.

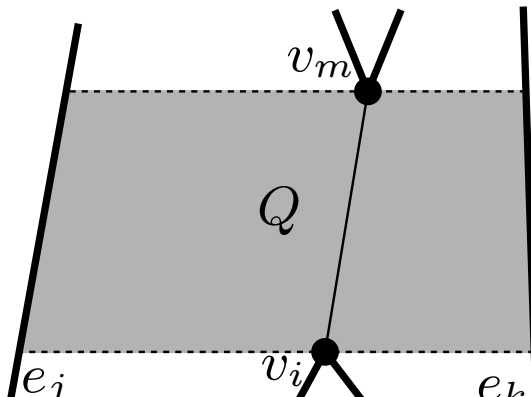


### Lemma 3.5

Algorithm MAKEMONOTONE adds a set of non-intersecting diagonals that partitions  $P$  into monotone subpolygons.

**Proof.** (For split vertices) (other cases are similar)

- No intersection between  $v_i v_m$  and edges of  $P$ .
- No intersection between  $v_i v_m$  and previous edges.



# Running time/ Space complexity

## Running time:

- Constructing the priority queue  $Q$ :  $\mathcal{O}(n)$  time.
- Initializing  $\mathcal{T}$ :  $\mathcal{O}(1)$  time.
- To handle an event, we perform:
  - 1 one operation on  $Q$ :  $\mathcal{O}(\log n)$  time.
  - 2 at most one query on  $\mathcal{T}$ :  $\mathcal{O}(\log n)$  time.
  - 3 one insertion, and one deletion on  $\mathcal{T}$ :  $\mathcal{O}(\log n)$  time.
  - 4 we insert at most two diagonals into  $\mathcal{D}$ :  $\mathcal{O}(1)$  time.

## Space Complexity:

The amount of storage used by the algorithm is clearly linear: every vertex is stored at most once in  $Q$ , and every edge is stored at most once in  $\mathcal{T}$ .



# Running time/ Space complexity

## Running time:

- Constructing the priority queue  $Q$ :  $\mathcal{O}(n)$  time.
- Initializing  $\mathcal{T}$ :  $\mathcal{O}(1)$  time.
- To handle an event, we perform:
  - 1 one operation on  $Q$ :  $\mathcal{O}(\log n)$  time.
  - 2 at most one query on  $\mathcal{T}$ :  $\mathcal{O}(\log n)$  time.
  - 3 one insertion, and one deletion on  $\mathcal{T}$ :  $\mathcal{O}(\log n)$  time.
  - 4 we insert at most two diagonals into  $\mathcal{D}$ :  $\mathcal{O}(1)$  time.

## Space Complexity:

The amount of storage used by the algorithm is clearly linear: every vertex is stored at most once in  $Q$ , and every edge is stored at most once in  $\mathcal{T}$ .

# Monotone Decomposition:

## Theorem 3.6

A simple polygon with  $n$  vertices can be partitioned into  $y$ -monotone polygons in  $\mathcal{O}(n \log n)$  time with an algorithm that uses  $\mathcal{O}(n)$  storage.



## Triangulating a Monotone Polygon



# Triangulating a Monotone Polygon

## Triangulation Algorithm:

- 1 The algorithm handles the vertices in order of decreasing  $y$ -coordinate. (Left to right for points with same  $y$ -coordinate).
- 2 The algorithm requires a stack  $S$  as auxiliary data structure. It keeps the points that handled but might need more diagonals.
- 3 When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible.
- 4 Algorithm invariant: the part of  $P$  that still needs to be triangulated, and lies above the last vertex that has been encountered so far, looks like a funnel turned upside down.



# Triangulating a Monotone Polygon

## Triangulation Algorithm:

- 1 The algorithm handles the vertices in order of decreasing  $y$ -coordinate. (Left to right for points with same  $y$ -coordinate).
- 2 The algorithm requires a stack  $S$  as auxiliary data structure. It keeps the points that handled but might need more diagonals.
- 3 When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible.
- 4 Algorithm invariant: the part of  $P$  that still needs to be triangulated, and lies above the last vertex that has been encountered so far, looks like a funnel turned upside down.



# Triangulating a Monotone Polygon

## Triangulation Algorithm:

- 1 The algorithm handles the vertices in order of decreasing  $y$ -coordinate. (Left to right for points with same  $y$ -coordinate).
- 2 The algorithm requires a stack  $S$  as auxiliary data structure. It keeps the points that handled but might need more diagonals.
- 3 When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible.
- 4 Algorithm invariant: the part of  $P$  that still needs to be triangulated, and lies above the last vertex that has been encountered so far, looks like a funnel turned upside down.



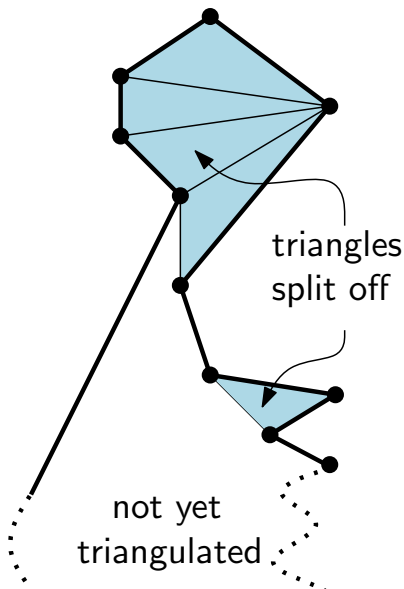
# Triangulating a Monotone Polygon

## Triangulation Algorithm:

- 1 The algorithm handles the vertices in order of decreasing  $y$ -coordinate. (Left to right for points with same  $y$ -coordinate).
- 2 The algorithm requires a stack  $S$  as auxiliary data structure. It keeps the points that handled but might need more diagonals.
- 3 When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible.
- 4 Algorithm invariant: the part of  $P$  that still needs to be triangulated, and lies above the last vertex that has been encountered so far, looks like a funnel turned upside down.



# Triangulating a Monotone Polygon





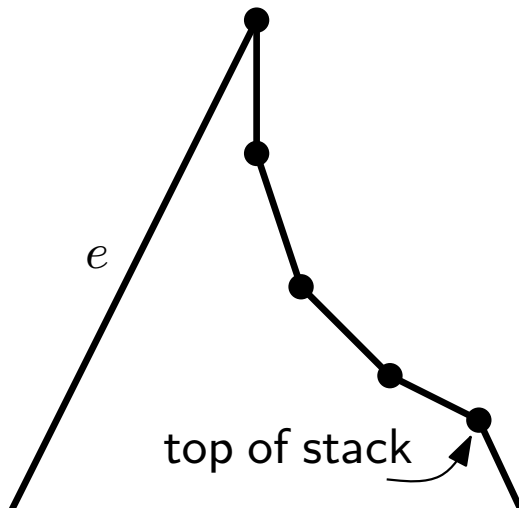
# Triangulating a Monotone Polygon

Case 1:  $v_j$  and top of stack on different chains



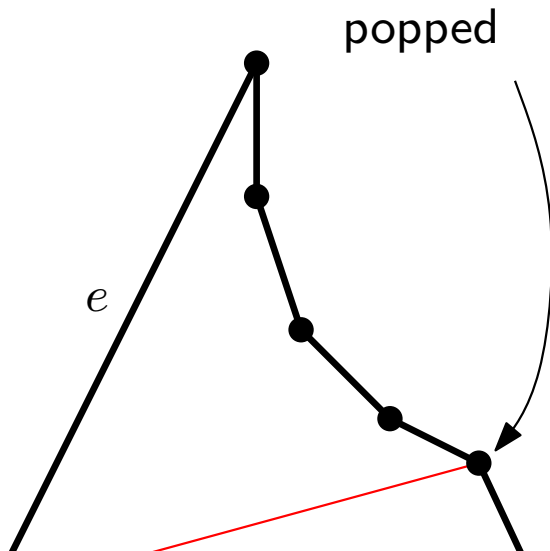
# Triangulating a Monotone Polygon

Case 1:  $v_j$  and top of stack on different chains



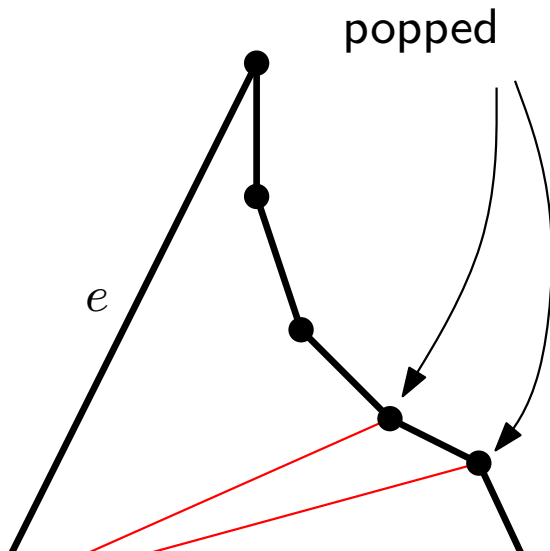
# Triangulating a Monotone Polygon

Case 1:  $v_j$  and top of stack on different chains



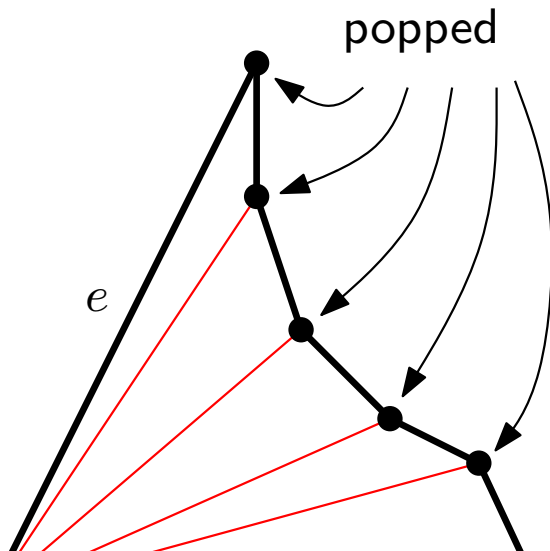
# Triangulating a Monotone Polygon

Case 1:  $v_j$  and top of stack on different chains



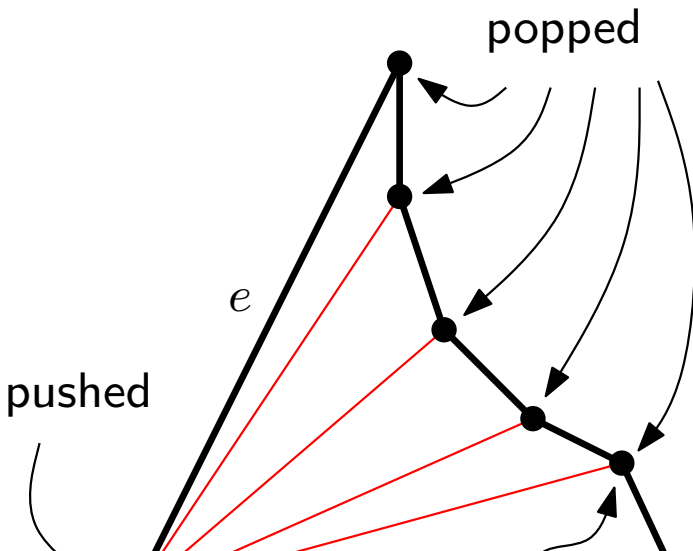
# Triangulating a Monotone Polygon

Case 1:  $v_j$  and top of stack on different chains



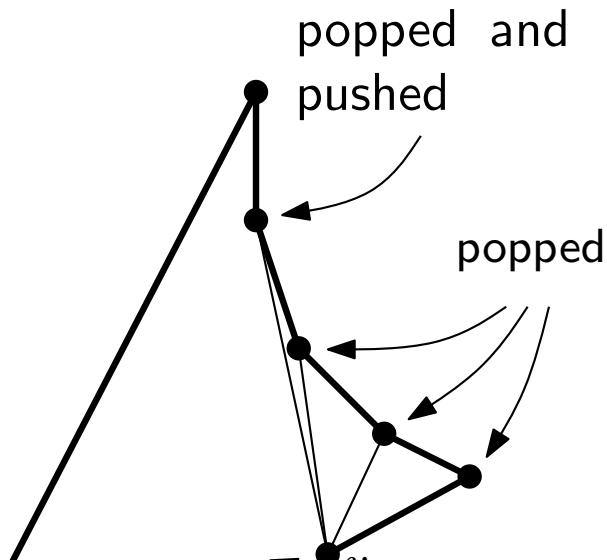
# Triangulating a Monotone Polygon

Case 1:  $v_j$  and top of stack on different chains



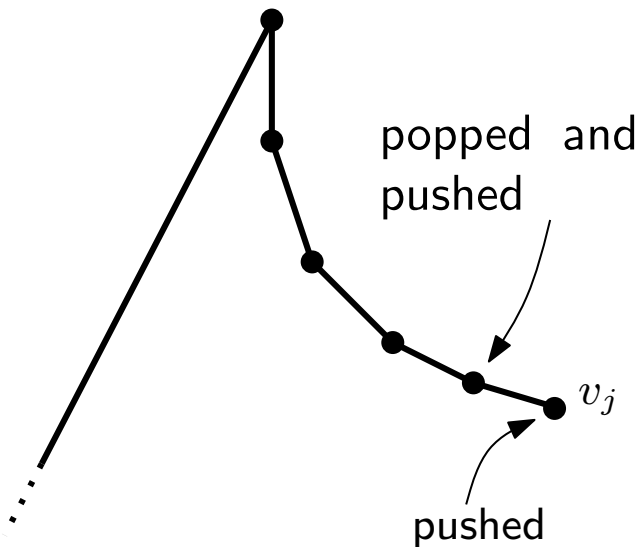
# Triangulating a Monotone Polygon

Case 2:  $v_j$  and top of stack on same chain



# Triangulating a Monotone Polygon

Case 2:  $v_j$  and top of stack on same chain





# Triangulating a Monotone Polygon

**Algorithm** TRIANGULATEMONOTONEPOLYGON( $P$ )

**Input:** A strictly  $y$ -monotone polygon  $P$  stored in  $\mathcal{D}$ .

**Output:** A triangulation of  $P$  stored in  $\mathcal{D}$ .

1. Merge the vertices on the left chain and the vertices on the right chain of  $P$  into one sequence, sorted on decreasing  $y$ -coordinate. Let  $u_1, \dots, u_n$  denote the sorted sequence.
2. Initialize an empty stack  $S$ , and push  $u_1$  and  $u_2$  onto it.
3. **for**  $j \leftarrow 3$  **to**  $n - 1$
4.     **if**  $u_j$  and the vertex on top of  $S$  are on different chains
5.         **then** Pop all vertices from  $S$ .
6.         Insert into  $\mathcal{D}$  a diagonal from  $u_j$  to each popped vertex, except the last one.
7.         Push  $u_{j-1}$  and  $u_j$  onto  $S$ .
8.     **else** Pop one vertex from  $S$ .
9.         Pop the other vertices from  $S$  as long as the diagonals from  $u_j$  to them are inside  $P$ . Insert these diagonals into  $\mathcal{D}$ . Push the last vertex that has been popped back onto  $S$ .
10.         Push  $u_j$  onto  $S$ .
11. Add diagonals from  $u_n$  to all stack vertices except the first and the last one.



# Polygon Triangulation

## Theorem 3.8

A simple polygon with  $n$  vertices can be triangulated in  $\mathcal{O}(n \log n)$  time with an algorithm that uses  $\mathcal{O}(n)$  storage.

## Theorem 3.9

A planar subdivision with  $n$  vertices in total can be triangulated in  $\mathcal{O}(n \log n)$  time with an algorithm that uses  $\mathcal{O}(n)$  storage.



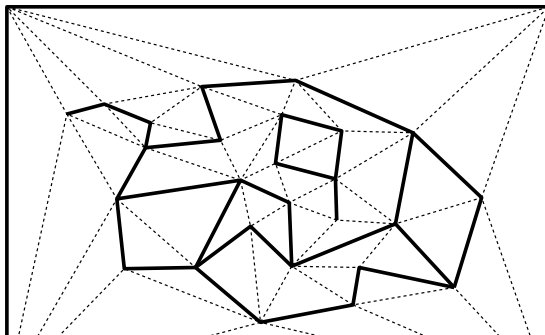
# Polygon Triangulation

## Theorem 3.8

A simple polygon with  $n$  vertices can be triangulated in  $\mathcal{O}(n \log n)$  time with an algorithm that uses  $\mathcal{O}(n)$  storage.

## Theorem 3.9

A planar subdivision with  $n$  vertices in total can be triangulated in  $\mathcal{O}(n \log n)$  time with an algorithm that uses  $\mathcal{O}(n)$  storage.



END.

