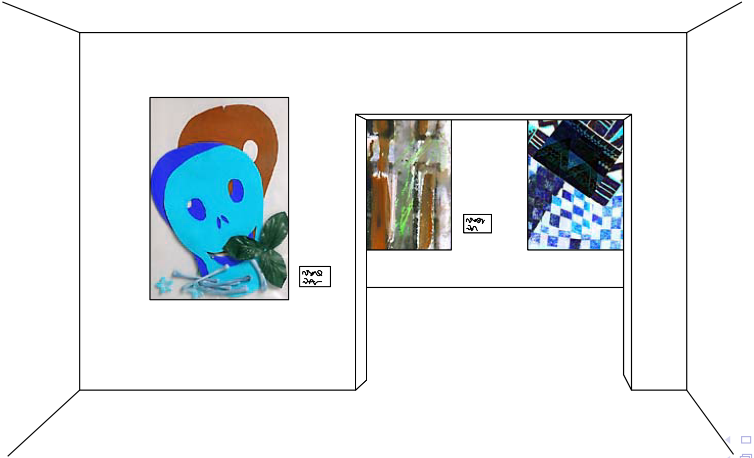
Polygon Triangulation

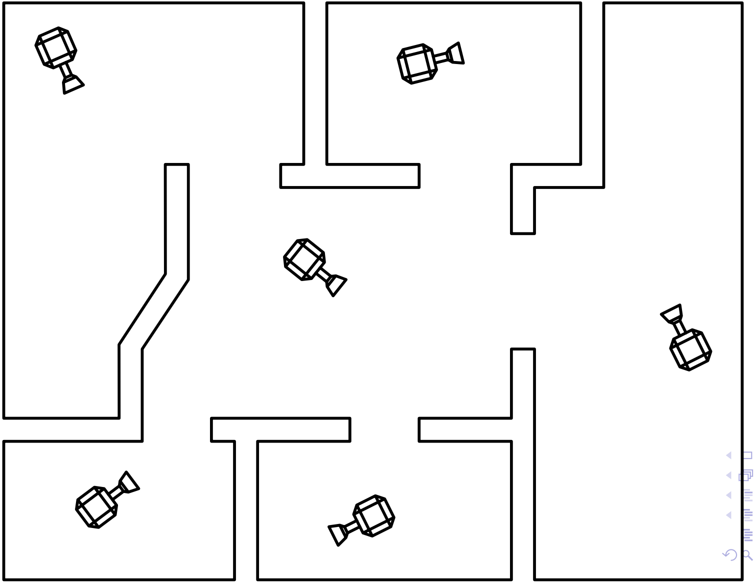
1397-2

Yazd Univ.

Motivation:
The Art Gallery Problem

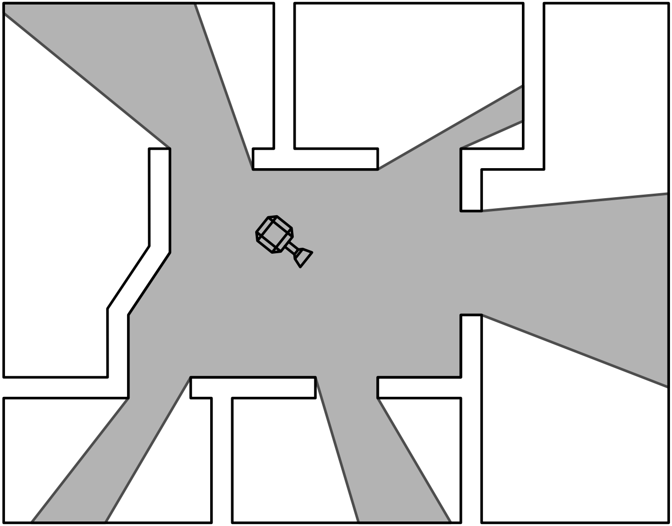


Motivation:
The Art Gallery Problem

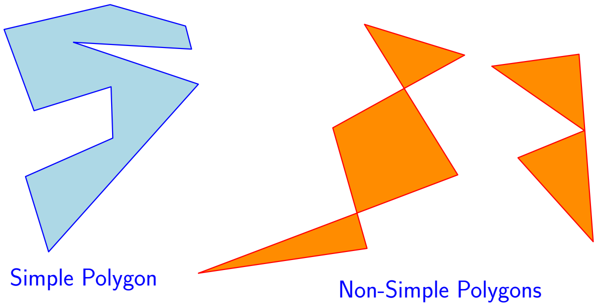


Motivation:

The Art Gallery Problem



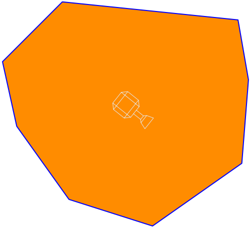
- Simple polygon: Regions enclosed by a single closed polygonal chain that does not intersect itself.
- Question: How many cameras do we need to guard a simple polygon?
  Answer: Depends on the polygon.
- One solution: Decompose the polygon to parts which are simple to guard.



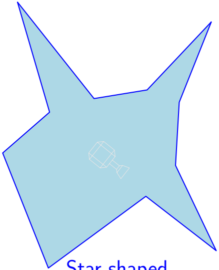
Triangulating Polygons

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.

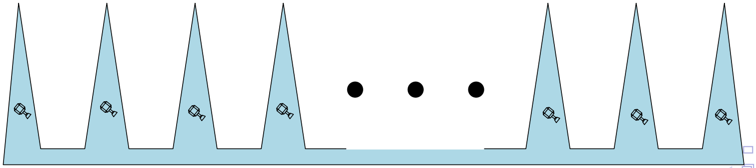


Convex

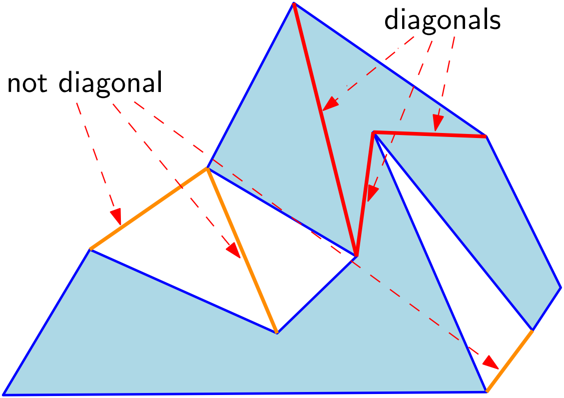


Star-shaped

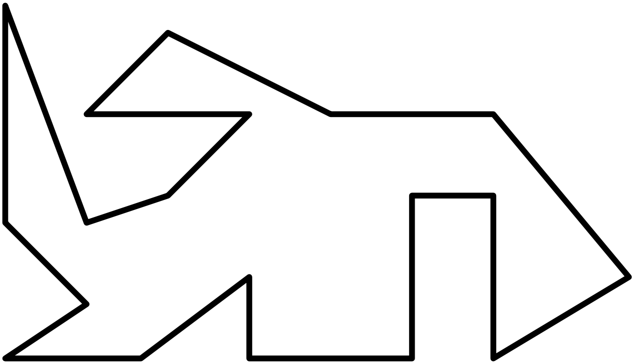
- Simple polygon: Regions enclosed by a single closed polygonal chain that does not intersect itself.
- Question: How many cameras do we need to guard a simple polygon?
  Answer: Depends on the polygon.
- One solution: Decompose the polygon to parts which are simple to guard.



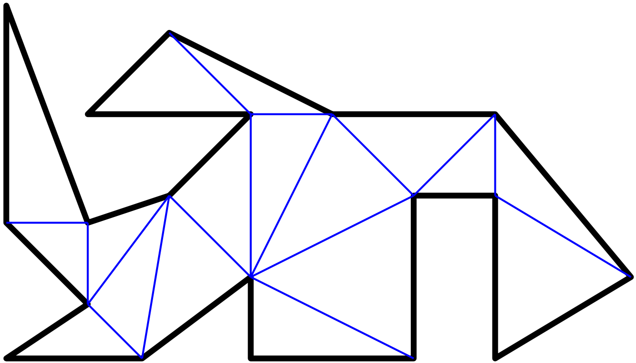
- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



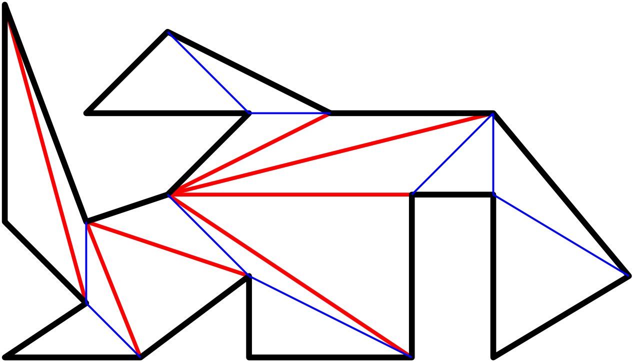
- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



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

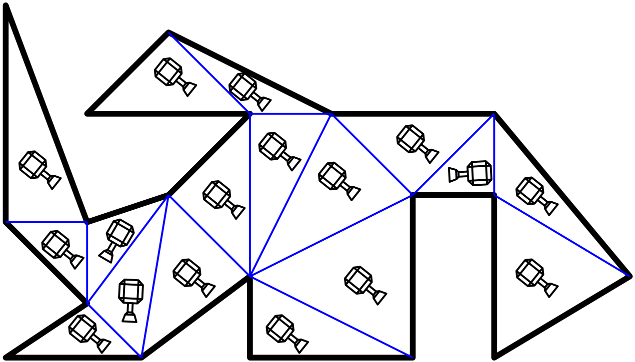


- diagonals:
- Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



- Guarding after triangulation:

- Guarding after triangulation:



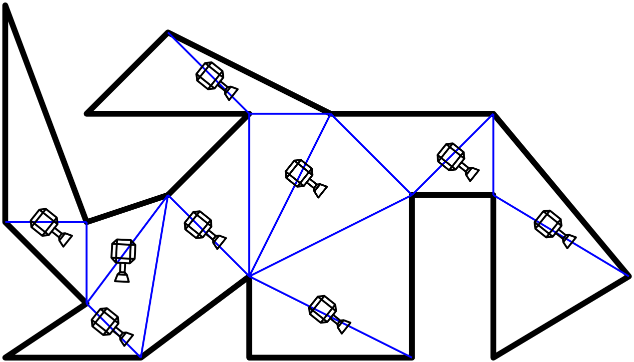
Yazd Univ.

Computational Geometry

The Art Gallery Problem
Guarding and Triangulations

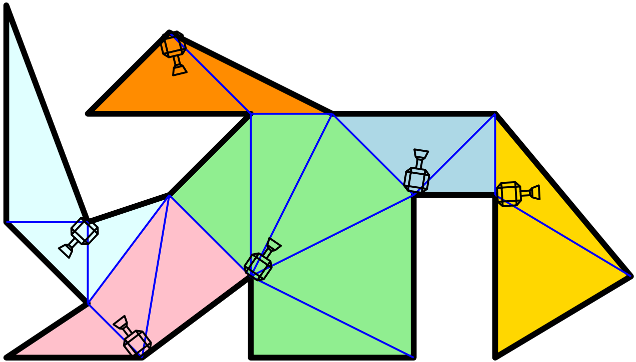
Computing triangulation
Partitioning a Polygon into Monotone Pieces
Triangulating a Monotone Polygon

- Guarding after triangulation:

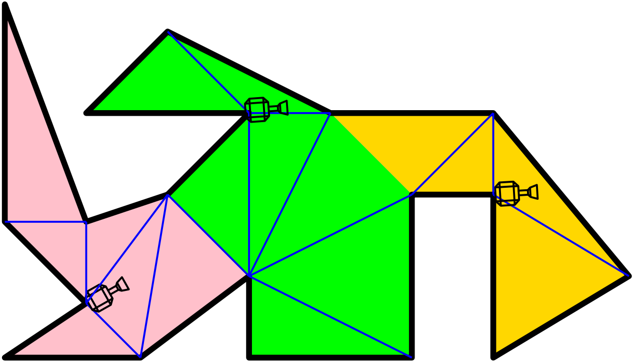


Triangulating Polygons
Definitions

- Guarding after triangulation:



- 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.
Base Case: $n = 3$ (Obvious)
Case 1: Neighbors of v make a diagonal.
Case 2: Otherwise.

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.
Base Case: $n = 3$ (Obvious)
Case 1: Neighbors of v make a diagonal.
Case 2: Otherwise.

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.
Base Case: $n = 3$ (Obvious)
Case 1: Neighbors of v make a diagonal.
Case 2: Otherwise.

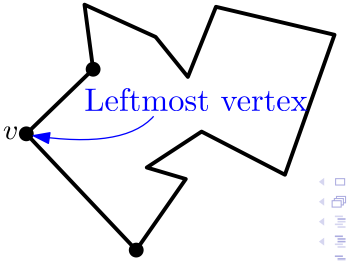
Questions:

- 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.
Base Case: $n = 3$ (Obvious)
Case 1: Neighbors of v make a diagonal.
Case 2: Otherwise.



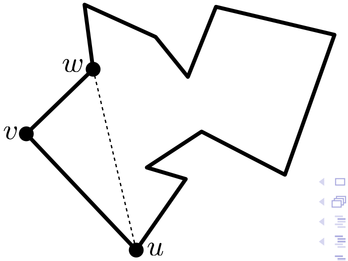
Questions:

- 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.
Base Case: $n = 3$ (Obvious)
Case 1: Neighbors of v make a diagonal.
Case 2: Otherwise.



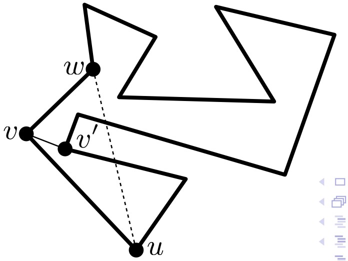
Questions:

- 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.
Base Case: $n = 3$ (Obvious)
Case 1: Neighbors of v make a diagonal.
Case 2: Otherwise.



Guarding a triangulated polygon

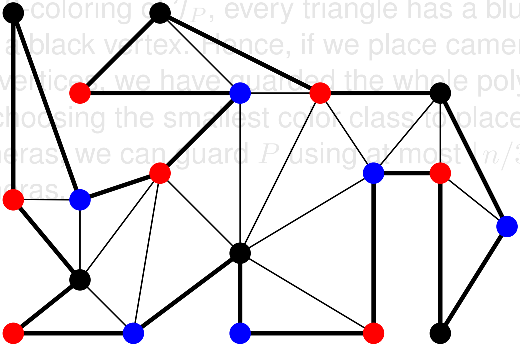
- $\mathcal{T}_P$: A triangulation of a simple polygon P.
- Select $S \subseteq$ the vertices of P, such that any triangle in $\mathcal{T}_P$ has at least one vertex in S, and place the cameras at vertices in S.
- To find such a subset: find a 3-coloring of a triangulated polygon.
- In a 3-coloring of $\mathcal{T}_P$, 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.

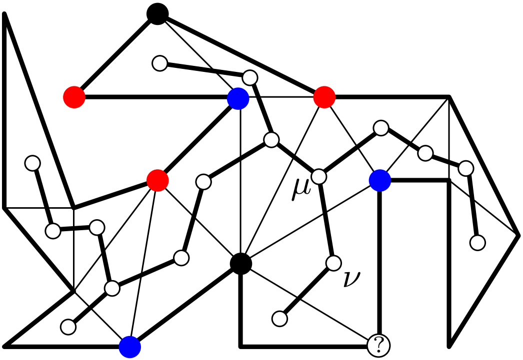
Guarding a triangulated polygon

- $\mathcal{T}_P$: 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 ν and μ if $t(\nu)$ and $t(\mu)$ share a diagonal.
- $\mathcal{G}(\mathcal{T}_P)$ is a tree.



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 ν and μ if $t(\nu)$ and $t(\mu)$ share a diagonal.
- $\mathcal{G}(\mathcal{T}_P)$ is a tree.

Does a 3-coloring always exist?

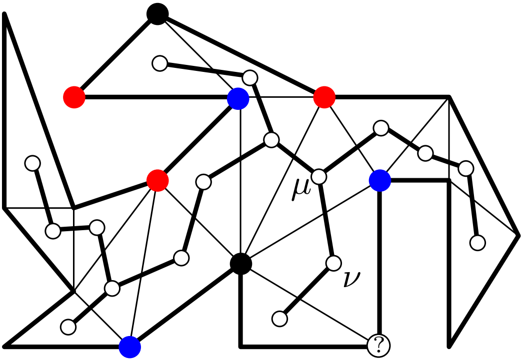
Dual graph:

- This graph $\mathcal{G}(\mathcal{T}_P)$ has a node for every triangle in $\mathcal{T}_P$.
- There is an arc between two nodes ν and μ 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 ν in $\mathcal{G}$, coming from node μ. Only one vertex of $t(\nu)$ remains to be colored.



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 ν in $\mathcal{G}$, coming from node μ. Only one vertex of $t(\nu)$ remains to be colored.

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 ν in $\mathcal{G}$, coming from node μ. Only one vertex of $t(\nu)$ remains to be colored.

Does a 3-coloring always exist?

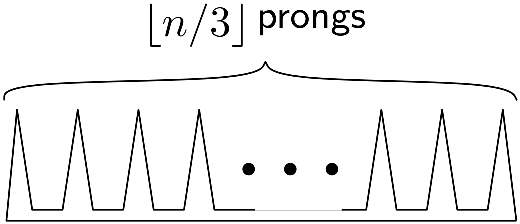
For 3-coloring:

- 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 ν in $\mathcal{G}$, coming from node μ. 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.



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.

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.

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.

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.

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.

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

ℓ-monotone polygon

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

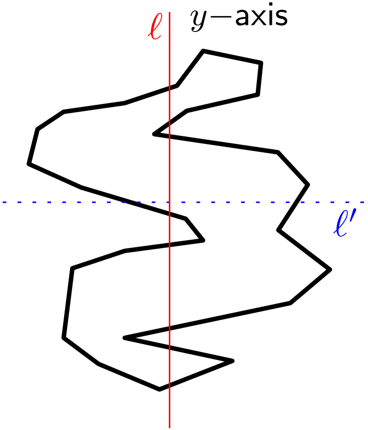
Definition:

- 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 $p_x < q_x$.

Monotone polygon

ℓ-monotone polygon

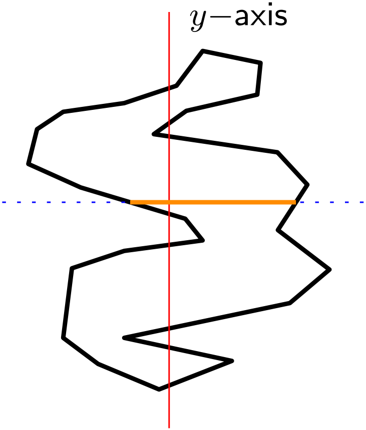
P is called monotone w. r. t. ℓ if $\forall \ell'$ perpendicular to ℓ the intersection of P with ℓ is connected (a line segment, a point, or empty).



Monotone polygon

ℓ-monotone polygon

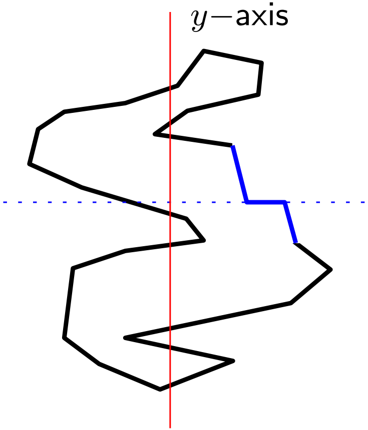
P is called monotone w. r. t. ℓ if $\forall \ell'$ perpendicular to ℓ the intersection of P with ℓ is connected (a line segment, a point, or empty).



Monotone polygon

ℓ-monotone polygon

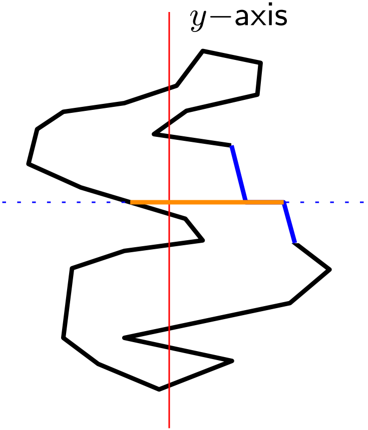
P is called monotone w. r. t. ℓ if $\forall \ell'$ perpendicular to ℓ the intersection of P with ℓ is connected (a line segment, a point, or empty).



Monotone polygon

ℓ-monotone polygon

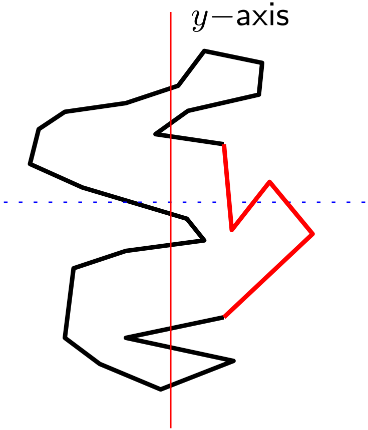
P is called monotone w. r. t. ℓ if $\forall \ell'$ perpendicular to ℓ the intersection of P with ℓ is connected (a line segment, a point, or empty).



Monotone polygon

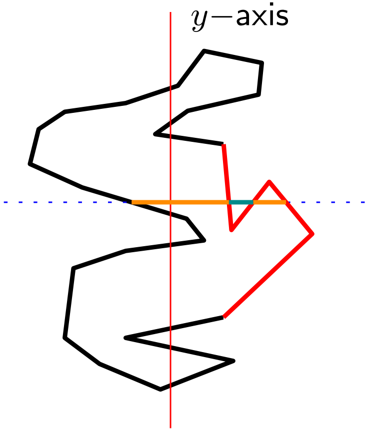
ℓ-monotone polygon

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



ℓ-monotone polygon

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



Monotone polygon

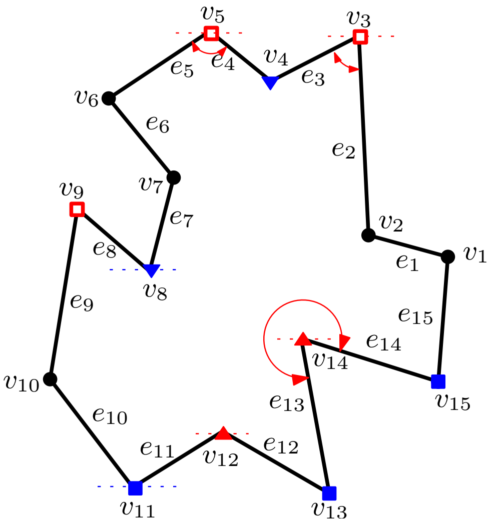
ℓ-monotone polygon

P is called monotone w. r. t. ℓ if $\forall \ell'$ perpendicular to ℓ the intersection of P with ℓ 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 $p_x < q_x$.

Partition P into monotone pieces



=start vertex

=end vertex

=regular vertex

=split vertex

=merge vertex

Partition P into monotone pieces

Lemma 3.4

If P has no split or merge vertices then it is y-monotone.

Proof. Assume P is not y-monotone. We show that it has an split or merge vertex.

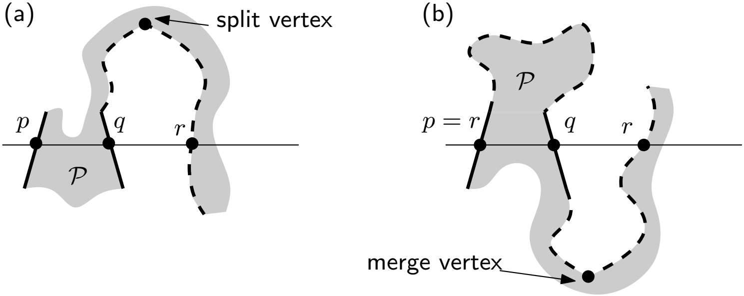
By Lemma 3.4, P has been partitioned into y-monotone pieces once we get rid of its split and merge vertices.

Partition P into monotone pieces

Lemma 3.4

If P has no split or merge vertices then it is y-monotone.

Proof. Assume P is not y-monotone. We show that it has an split or merge vertex.



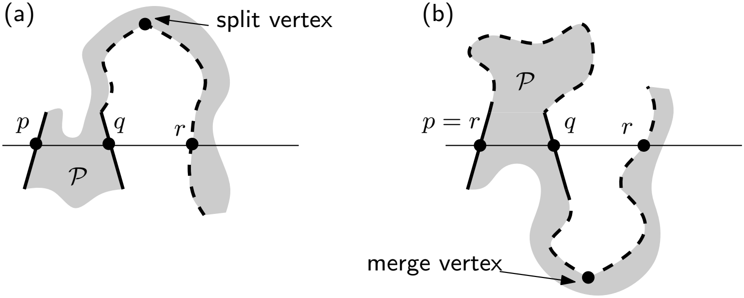
By Lemma 3.4, P has been partitioned into y-monotone pieces once we get rid of its split and merge vertices.

Partition P into monotone pieces

Lemma 3.4

If P has no split or merge vertices then it is y-monotone.

Proof. Assume P is not y-monotone. We show that it has an split or merge vertex.



By Lemma 3.4, P has been partitioned into y-monotone pieces once we get rid of its split and 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.
- $helper(e_j)$: Lowest vertex above the sweep line s. t. the horizontal segment connecting the vertex to e_j lies inside P.
- Connect split vertices to the helper of the edge to their left.

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.
- $helper(e_j)$: Lowest vertex above the sweep line s. t. the horizontal segment connecting the vertex to e_j lies inside P.
- Connect split vertices to the helper of the edge to their left.

Removing split/merge vertices:

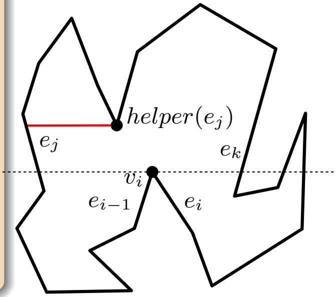
Removing split vertices:

- A sweep line algorithm.
- Events: all the points
- Goal: To add diagonals from each split vertex to a vertex lying above it.
- $helper(e_j)$: Lowest vertex above the sweep line s. t. the horizontal segment connecting the vertex to e_j lies inside P.
- Connect split vertices to the helper of the edge to their left.

Removing split/merge vertices:

Removing split vertices:

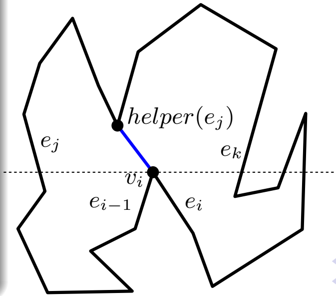
- A sweep line algorithm.
- Events: all the points
- Goal: To add diagonals from each split vertex to a vertex lying above it.
- $helper(e_j)$: Lowest vertex above the sweep line s. t. the horizontal segment connecting the vertex to e_j lies inside P.
- Connect split vertices to the helper of the edge to their left.



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.
- $helper(e_j)$: Lowest vertex above the sweep line s. t. the horizontal segment connecting the vertex to e_j lies inside P.
- Connect split vertices to the helper of the edge to their left.



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.

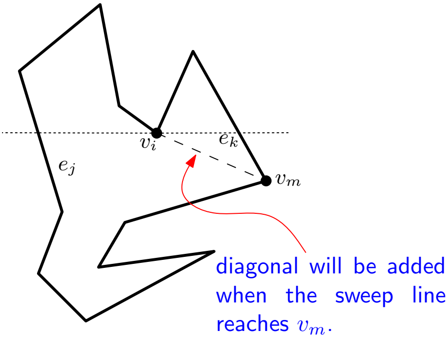
Removing split/merge vertices:

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 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 $\mathcal{D}$.

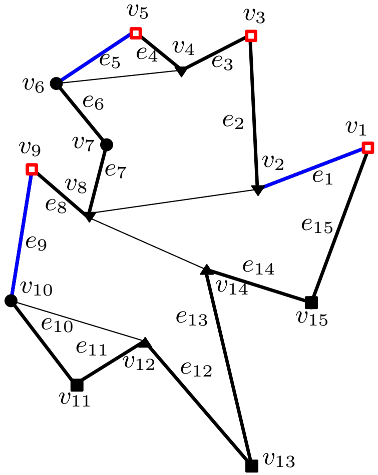
Output: A partitioning of P into monotone subpolygons, stored in $\mathcal{D}$.

1. Construct a priority queue $\mathcal{Q}$ 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** $\mathcal{Q}$ is not empty
4. Remove the vertex v_i with the highest priority from $\mathcal{Q}$.
5. Call the appropriate procedure to handle the vertex, depending on its type.

Make Monotone Algorithm:

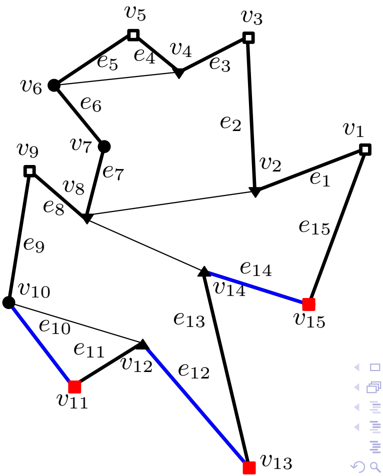
Algorithm HANDLESTARTVERTEX(v_i)

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



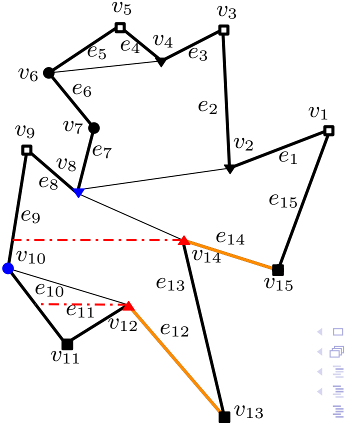
Algorithm HANDLEENDVERTEX(v_i)

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 $\mathcal{T}$.



Algorithm HANDLESPLITVERTEX(v_i)

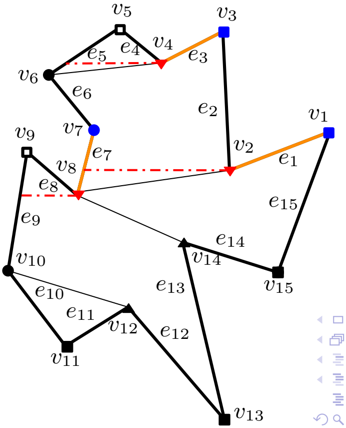
1. Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.
2. Insert the diagonal connecting v_i to $helper(e_j)$ in $\mathcal{D}$.
3. $helper(e_j) \leftarrow v_i$.
4. Insert e_i in $\mathcal{T}$ and set $helper(e_i)$ to v_i.



Make Monotone Algorithm:

Algorithm HANDLEMERGEVERTEX(v_i)

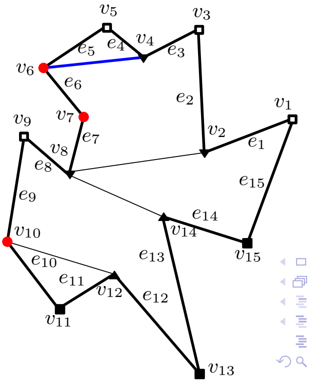
1. **if** $helper(e_{i-1})$ is a merge vertex
2. **then** Insert the diag. v_i to $helper(e_{i-1})$ in $\mathcal{D}$.
3. Delete e_{i-1} from $\mathcal{T}$.
4. Search in $\mathcal{T}$ to find e_j directly left of v_i.
5. **if** $helper(e_j)$ is a merge vertex
6. **then** Insert the diag. v_i to $helper(e_j)$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



Make Monotone Algorithm:

Algorithm HANDLEREGULARVERTEX(v_i)

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



Yazd Univ.

Computational Geometry

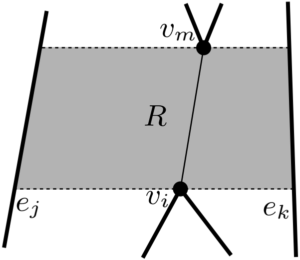
Lemma 3.5

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

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

- No vertex inside R.
- No intersection between $v_i v_m$ and edges of P.
- No intersection between $v_i v_m$ and previous diag.

  R is empty, endpoints of previously added edges: above v_i.

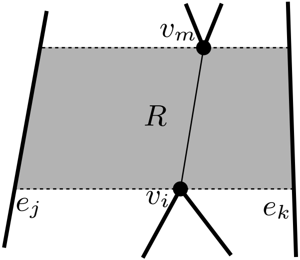


Lemma 3.5

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

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

- No vertex inside R.
- No intersection between $v_i v_m$ and edges of P.
- No intersection between $v_i v_m$ and previous diag. R is empty, endpoints of previously added edges: above v_i.

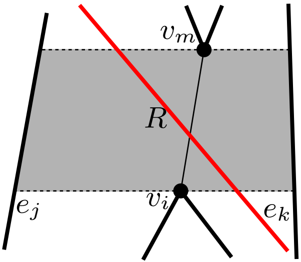


Lemma 3.5

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

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

- No vertex inside R.
- No intersection between $v_i v_m$ and edges of P.
- No intersection between $v_i v_m$ and previous diag. R is empty, endpoints of previously added edges: above v_i.

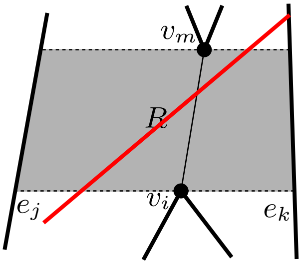


Lemma 3.5

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

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

- No vertex inside R.
- No intersection between $v_i v_m$ and edges of P.
- No intersection between $v_i v_m$ and previous diag. R is empty, endpoints of previously added edges: above v_i.



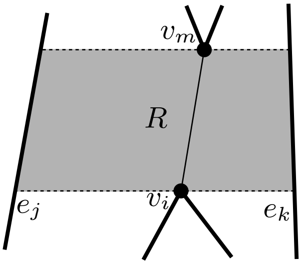
Lemma 3.5

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

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

- No vertex inside R.
- No intersection between $v_i v_m$ and edges of P.
- No intersection between $v_i v_m$ and previous diag.

  R is empty, endpoints of previously added edges: above v_i.



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 $\mathcal{T}$.

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 $\mathcal{T}$.

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 $\mathcal{T}$.

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 $\mathcal{S}$ 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.

Triangulation Algorithm:

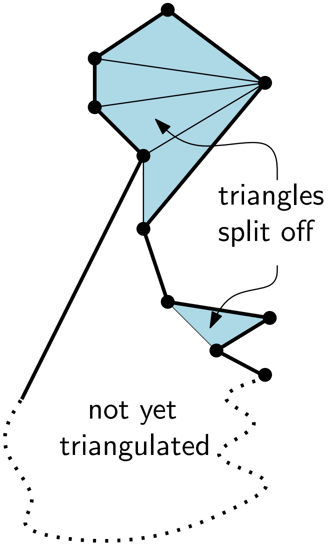
1. The algorithm handles the vertices in order of decreasing y-coordinate. (Left to right for points with same y-coordinate).
2. The algorithm requires a stack $\mathcal{S}$ as auxiliary data structure. It keeps the points that handled but might need more diagonals.
3. When we handle a vertex we add 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 $\mathcal{S}$ 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

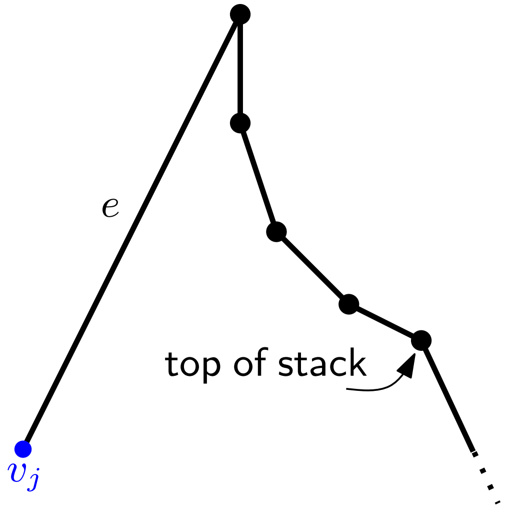


Triangulating a Monotone Polygon

Case 1: v_j and top of stack on different chains

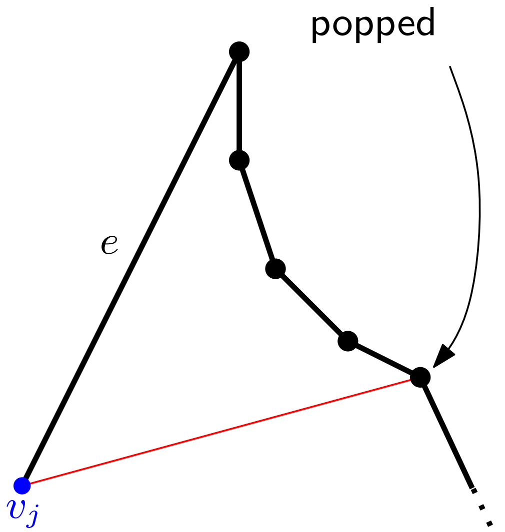
Triangulating a Monotone Polygon

Case 1: v_j and top of stack on different chains



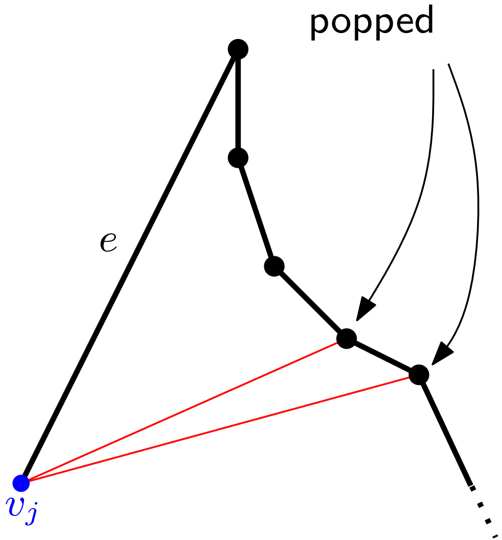
Triangulating a Monotone Polygon

Case 1: v_j and top of stack on different chains



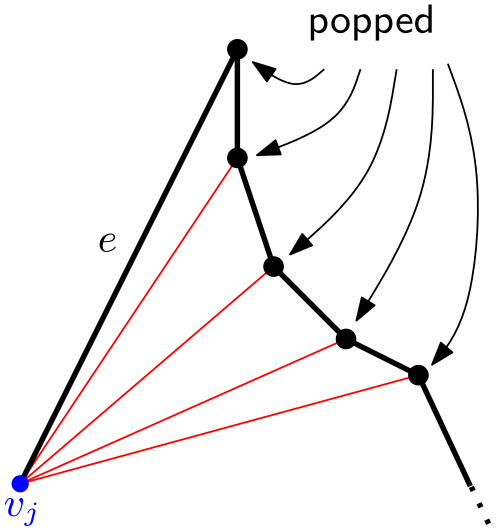
Triangulating a Monotone Polygon

Case 1: v_j and top of stack on different chains



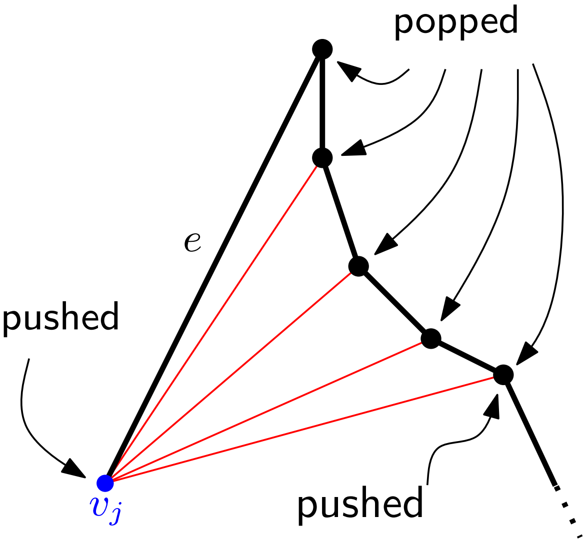
Triangulating a Monotone Polygon

Case 1: v_j and top of stack on different chains



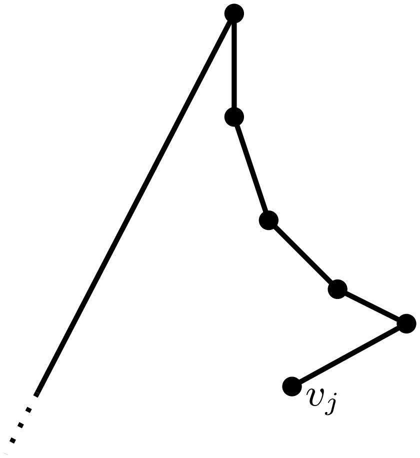
Triangulating a Monotone Polygon

Case 1: v_j and top of stack on different chains



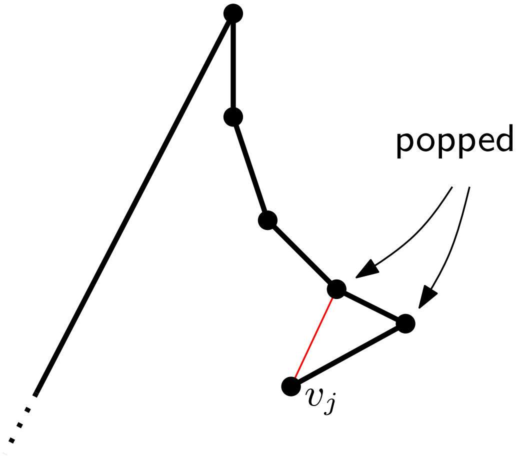
Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



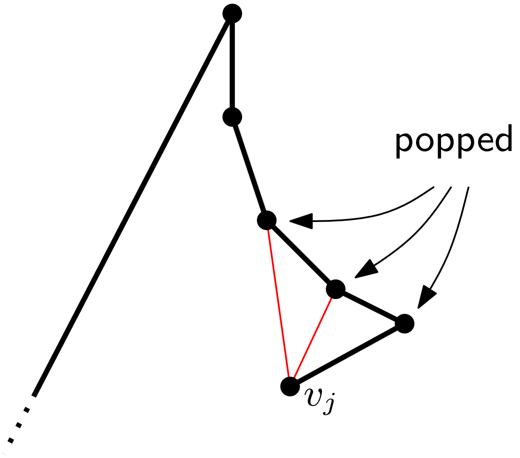
Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



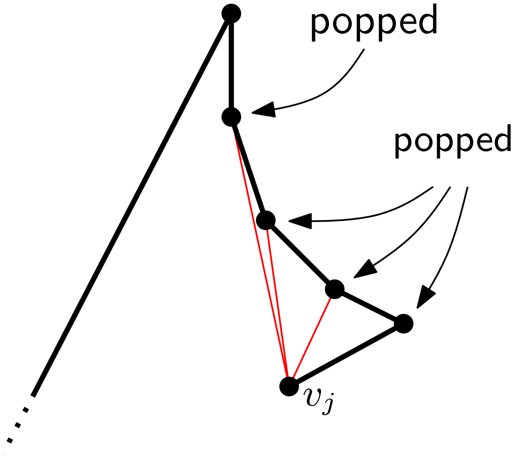
Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



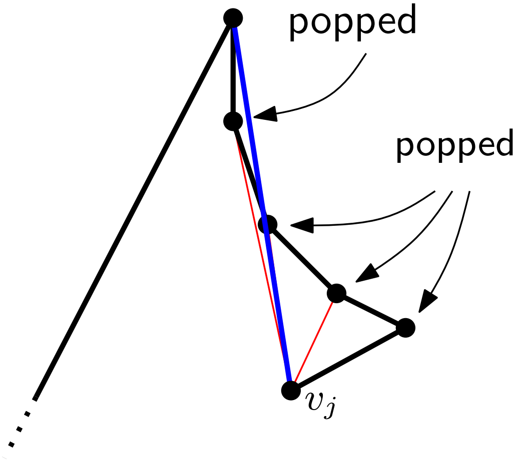
Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



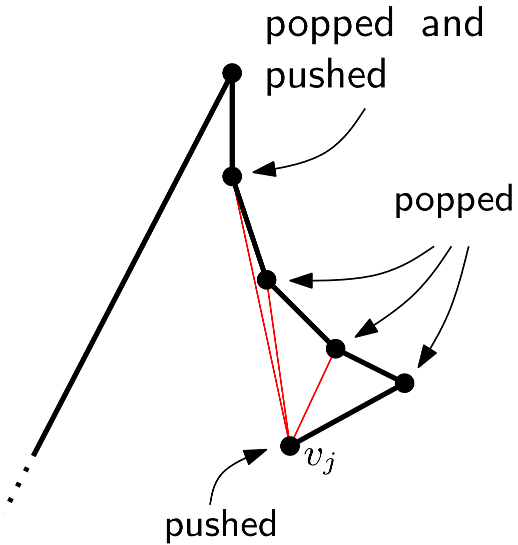
Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



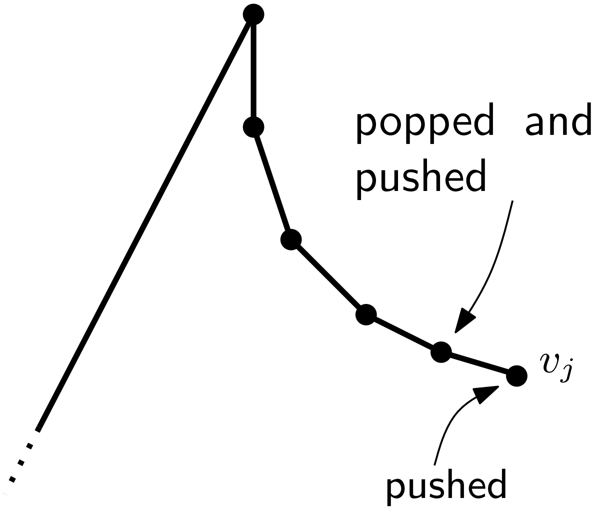
Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



Triangulating a Monotone Polygon

Case 2: v_j and top of stack on same chain



Triangulating a Monotone Polygon

Algorithm TRIANGULATEMONOTONEPOLYGON(P)

1. Merge the vertices on the left &right chain of P into one sequence, sorted on decreasing y-coordinate. Let $u_1, \ldots, u_n$ denote the sorted sequence.
2. Initialize an empty stack $\mathcal{S}$, 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 $\mathcal{S}$ are on different chains
5. **then** Pop all vertices from $\mathcal{S}$.
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 $\mathcal{S}$.
8. **else** Pop one vertex from $\mathcal{S}$.
9. Pop the other vertices from $\mathcal{S}$ 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 $\mathcal{S}$.
10. Push u_j onto $\mathcal{S}$.
11. Add diagonals from u_n to all stack vertices except the first and the last one.

Time Complexity: Step 1: $\mathcal{O}(n)$, Step 2: $\mathcal{O}(1)$.
Steps 3-10: $\mathcal{O}(n)$

Triangulating a Monotone Polygon

Algorithm TRIANGULATEMONOTONEPOLYGON(P)

1. Merge the vertices on the left &right chain of P into one sequence, sorted on decreasing y-coordinate. Let $u_1, \ldots, u_n$ denote the sorted sequence.
2. Initialize an empty stack $\mathcal{S}$, 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 $\mathcal{S}$ are on different chains
5.   **then** Pop all vertices from $\mathcal{S}$.
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 $\mathcal{S}$.
8.   **else** Pop one vertex from $\mathcal{S}$.
9.   Pop the other vertices from $\mathcal{S}$ 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 $\mathcal{S}$.
10.   Push u_j onto $\mathcal{S}$.
11. Add diagonals from u_n to all stack vertices except the first and the last one.

Time Complexity: Step 1: $\mathcal{O}(n)$, Step 2: $\mathcal{O}(1)$.
Steps 3-10: $\mathcal{O}(n)$

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 $O(n)$ storage.

Generalizing to any planar subdivision

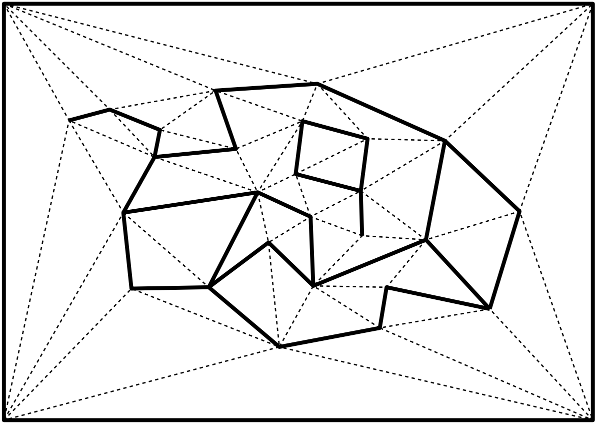
Theorem 3.9

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.

Generalizing to any planar subdivision

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.