Computational Geometry

Triangulating Polygons

or

Guarding Art Galleries

Lecture #3

[Comp. Geom A&A : Chapter 3]
Guarding an Art Gallery

Given a \textit{simple} polygon $P$ (i.e., no holes, no self-intersection)...
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$. 
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras!
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! *But minimizing them is NP-hard…*
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $qc \subseteq P$.

**Aim:** Use few cameras! *But minimizing them is NP-hard…*
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ "sees" a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $qc \subseteq P$.

**Aim:** Use few cameras! But minimizing them is NP-hard…
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! *But minimizing them is NP-hard...*
Guarding an Art Gallery

Given a simple polygon $P$ (i.e., no holes, no self-intersection)...

Observation. Camera $c$ “sees” a star-shaped region

Definition. A pt $q \in P$ is visible from $c \in P$ if $\overline{qc} \subseteq P$.

Aim: Use few cameras! But minimizing them is NP-hard…
Guarding an Art Gallery

Given a simple polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is visible from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! But minimizing them is NP-hard…
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

*Definition.* A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

*Observation.* Camera $c$ “sees” a star-shaped region

*Aim:* Use few cameras! *But minimizing them is NP-hard...*
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! *But minimizing them is NP-hard...*
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! *But minimizing them is NP-hard…*
Guarding an Art Gallery

Given a \textit{simple} polygon \( P \) (i.e., no holes, no self-intersection)...

\begin{itemize}
  \item \textbf{Observation.} Camera \( c \) “sees” a star-shaped region
  \item \textbf{Definition.} A pt \( q \in P \) is \textit{visible} from \( c \in P \) if \( \overline{qc} \subseteq P \).
  \item \textbf{Aim:} Use few cameras! \textit{But minimizing them is NP-hard…}
\end{itemize}
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! *But minimizing them is NP-hard…*
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $qc \subseteq P$.

**Aim:** Use few cameras! But minimizing them is NP-hard…

**Theorem.** 1. Every simple polygon can be triangulated.
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! 

**Theorem.** 1. Every simple polygon can be triangulated.
2. Any triangulation of a simple polygon with $n$ vertices consists of $n - 2$ triangles.

But minimizing them is NP-hard...
Guarding an Art Gallery

Given a *simple* polygon $P$ (i.e., no holes, no self-intersection)...

**Observation.** Camera $c$ “sees” a star-shaped region

**Definition.** A pt $q \in P$ is *visible* from $c \in P$ if $\overline{qc} \subseteq P$.

**Aim:** Use few cameras! 

**Theorem.**

1. Every simple polygon can be triangulated.
2. Any triangulation of a simple polygon with $n$ vertices consists of $n - 2$ triangles.
## The Art Gallery Theorem

[Chvátal ’75]

**Theorem.** For surveilling a simple polygon with $n$ vertices, $\lfloor n/3 \rfloor$ cameras are sometimes necessary and always sufficient.
The Art Gallery Theorem

Theorem. For surveilling a simple polygon with $n$ vertices, $\left\lfloor n/3 \right\rfloor$ cameras are sometimes necessary and always sufficient.

Exercise. Find, for arbitrarily large $n$, a polygon with $n$ vertices, where $\approx n/3$ cameras are necessary.

[2 minutes]
The Art Gallery Theorem

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

**Exercise.** Find, for arbitrarily large $n$, a polygon with $n$ vertices, where $\approx n/3$ cameras are necessary.

[dBCvKO'08]
The Art Gallery Theorem

[Chvátal ’75]

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

**Exercise.** Find, for arbitrarily large \( n \), a polygon with \( n \) vertices, where \( \approx n/3 \) cameras are necessary.

[2 minutes]

[DBCvKO’08]
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lceil n/3 \rceil \) cameras are sometimes necessary and always sufficient.

**Exercise.** Find, for arbitrarily large \( n \), a polygon with \( n \) vertices, where \( \approx n/3 \) cameras are necessary.
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lceil n/3 \rceil \) cameras are sometimes necessary and always sufficient.

**Exercise.** Find, for arbitrarily large \( n \), a polygon with \( n \) vertices, where \( \approx n/3 \) cameras are necessary.
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with $n$ vertices, \(\lfloor n/3 \rfloor\) cameras are sometimes necessary and always sufficient.
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with $n$ vertices, $\left\lfloor n/3 \right\rfloor$ cameras are sometimes necessary and always sufficient.

[Chvátal ’75]
The Art Gallery Theorem

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

**To do:** Find algorithm for triangulating a simple polygon!
The Art Gallery Theorem

Theorem.

For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

To do:

Find algorithm for triangulating a simple polygon!

Brute force:
The Art Gallery Theorem

[Chvátal ’75]

**Theorem.** For surveilling a simple polygon with $n$ vertices, ⌊$n/3$⌋ cameras are sometimes necessary and always sufficient.

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion
The Art Gallery Theorem

<table>
<thead>
<tr>
<th>Theorem.</th>
<th>For surveilling a simple polygon with ( n ) vertices, ( \lceil n/3 \rceil ) cameras are sometimes necessary and always sufficient.</th>
</tr>
</thead>
</table>

- **To do:** Find algorithm for triangulating a simple polygon!

- **Brute force:** follow existence proof, using recursion

  running time:
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion

running time: \( O(n^2) \)
The Art Gallery Theorem

[Chvátal '75]

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lceil n/3 \rceil \) cameras are sometimes necessary and always sufficient.

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion
running time: \( O(n^2) \)

**Faster triangulation in two steps:**
Theorem. For surveilling a simple polygon with $n$ vertices, \( \lceil n/3 \rceil \) cameras are sometimes necessary and always sufficient.

To do: Find algorithm for triangulating a simple polygon!

Brute force: follow existence proof, using recursion
running time: $O(n^2)$

Faster triangulation in two steps:

$n$-vtx polygon
The Art Gallery Theorem

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

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion
running time: $O(n^2)$

**Faster triangulation in two steps:**

$n$-vtx polygon $\rightarrow$ “nice” pieces, $n'$ vtc $\rightarrow$
The Art Gallery Theorem

Theorem. For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

To do: Find algorithm for triangulating a simple polygon!

Brute force: follow existence proof, using recursion running time: \( O(n^2) \)

Faster triangulation in two steps:

| \( n \)-vtx polygon | “nice” pieces, \( n' \) vtc | \( n'' \) triangles |
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with $n$ vertices, \( \lceil n/3 \rceil \) cameras are sometimes necessary and always sufficient.

[Chvátal '75]

[black board]

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion
running time: \( O(n^2) \)

**Faster triangulation in two steps:**

- \( n \)-vtx polygon $\rightarrow$ “nice” pieces, \( n' \) vtc $\rightarrow$ \( n'' \) triangles
- \( O(n \log n) \)
The Art Gallery Theorem

[Chvátal ’75]

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion

running time: \( O(n^2) \)

**Faster triangulation in two steps:**

<table>
<thead>
<tr>
<th>( n )-vtx polygon</th>
<th>“nice” pieces, ( n' ) vtc</th>
<th>( n'' ) triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(n \log n) )</td>
<td>( O(n') )</td>
<td></td>
</tr>
</tbody>
</table>

[black board]
The Art Gallery Theorem

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion
running time: \( O(n^2) \)

**Faster triangulation in two steps:**

- \( n \)-vtx polygon → “nice” pieces, \( n' \) vtc → \( n'' \) triangles
- \( O(n \log n) \) \( O(n') \)

**Definition.** A polygon \( P \) is \( y \)-monotone
if, for any horizontal line \( \ell \), \( \ell \cap P \) is connected.
The Art Gallery Theorem

Theorem. For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

To do: Find algorithm for triangulating a simple polygon!

Brute force: follow existence proof, using recursion
running time: \( O(n^2) \)

Faster triangulation in two steps:

- \( n \)-vtx polygon → “nice” pieces, \( n' \) vtc → \( n'' \) triangles
  - \( O(n \log n) \)
  - \( O(n') \)

Definition. A polygon \( P \) is \( y \)-monotone if, for any horizontal line \( \ell \), \( \ell \cap P \) is connected.
The Art Gallery Theorem [Chvátal ’75]

**Theorem.** For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient.

**To do:** Find algorithm for triangulating a simple polygon!

**Brute force:** follow existence proof, using recursion
running time: \( O(n^2) \)

**Faster triangulation in two steps:**
\[
\begin{align*}
&\text{n-vtx polygon} \quad \text{“nice” pieces, } n' \text{ vtc} \quad \text{n” triangles} \\
&O(n \log n) \quad O(n')
\end{align*}
\]

**Definition.** A polygon \( P \) is \( y \)-monotone if, for any horizontal line \( \ell \), \( \ell \cap P \) is connected.
The Art Gallery Theorem

Theorem. For surveilling a simple polygon with \( n \) vertices, \( \lfloor n/3 \rfloor \) cameras are sometimes necessary and always sufficient. [Chvátal ’75]

To do: Find algorithm for triangulating a simple polygon!

Brute force: follow existence proof, using recursion
running time: \( O(n^2) \)

Faster triangulation in two steps:

- \( n \)-vtx polygon \( \rightarrow \) “nice” pieces, \( n' \) vtc
- \( n'' \) triangles

Running time:
- \( O(n \log n) \)
- \( O(n') \)

Definition. A polygon \( P \) is \( y\)-monotone if, for any horizontal line \( \ell \), \( \ell \cap P \) is connected.
The Art Gallery Theorem

Theorem. For surveilling a simple polygon with \( n \) vertices, \( \lceil n/3 \rceil \) cameras are sometimes necessary and always sufficient.

[Chvátal ’75]

To do: Find algorithm for triangulating a simple polygon!

Brute force: follow existence proof, using recursion running time: \( O(n^2) \)

Faster triangulation in two steps:

- \( n \)-vtx polygon → “nice” pieces, \( n' \) vtc → \( n'' \) triangles
- \( O(n \log n) \) \( O(n') \)

Definition. A polygon \( P \) is \( y \)-monotone if, for any horizontal line \( \ell \), \( \ell \cap P \) is connected.
Partitioning a Polygon into Monotone Pieces

Idea: Classify vertices of given simple polygon $P$
Idea: Classify vertices of given simple polygon $P$

– *turn* vertices:

– *regular* vertices
Idea: Classify vertices of given simple polygon $P$

- turn vertices:
  vertical component of walking direction changes

- regular vertices
Partitioning a Polygon into Monotone Pieces

**Idea:** Classify vertices of given simple polygon $P$

- **turn** vertices:
  - vertical component of walking direction changes
  - $\text{start vertex}$
  - if $\alpha < 180^\circ$

- **regular** vertices
Partitioning a Polygon into Monotone Pieces

**Idea:**
Classify vertices of given simple polygon $P$

- *turn* vertices:
  - vertical component of walking direction changes

  - **start** vertex

  - **split** vertex

- *regular* vertices

  - if $\alpha < 180^\circ$

  - if $\beta > 180^\circ$
**Partitioning a Polygon into Monotone Pieces**

**Idea:** Classify vertices of given simple polygon $P$

- **turn** vertices:
  - vertical component of walking direction changes
  - start vertex
  - split vertex
  - end vertex

- **regular** vertices

  - if $\alpha < 180^\circ$
  - if $\beta > 180^\circ$
  - if $\gamma < 180^\circ$
Partitioning a Polygon into Monotone Pieces

**Idea:** Classify vertices of given simple polygon $P$

- *turn* vertices:
  - vertical component of walking direction changes
    - *start* vertex
    - *split* vertex
    - *end* vertex
    - *merge* vertex
- *regular* vertices

- if $\alpha < 180^\circ$
- if $\beta > 180^\circ$
- if $\gamma < 180^\circ$
- if $\delta > 180^\circ$
Partitioning a Polygon into Monotone Pieces

**Idea:** Classify vertices of given simple polygon $P$

- **turn vertices:**
  - vertical component of walking direction changes

  *start vertex*

  *split vertex*

  *end vertex*

  *merge vertex*

- **regular vertices**

**Lemma:** Let $P$ be a simple polygon. Then $P$ is $y$-monotone $\iff$ $P$ has neither split vertices nor merge vertices.
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross
- each other
- edges of $P$
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross – each other – edges of $P$

1) Treating split vertices
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross – each other – edges of $P$

1) Treating split vertices
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross each other – edges of $P$

1) Treating split vertices
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross each other – edges of $P$

1) Treating split vertices
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross
- each other
- edges of $P$

1) Treating split vertices
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross — each other — edges of $P$

1) Treating split vertices

Connect $v$ to vertex $w^*$ having minimum $y$-coordinate among all vertices $w$ above $v$ and with $\text{left}(w) = \text{left}(v)$. 
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross
- each other
- edges of $P$

1) Treating split vertices

Connect $v$ to vertex $w^*$ having minimum $y$-coordinate among all vertices $w$ above $v$ and with $\text{left}(w) = \text{left}(v)$. 
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross — each other — edges of $P$

1) Treating split vertices

Connect $v$ to vertex $w^*$ having minimum $y$-coordinate among all vertices $w$ above $v$ and with $\text{left}(w) = \text{left}(v)$. 
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross each other – edges of \( P \)

1) Treating split vertices

Connect \( v \) to vertex \( w^\star \) having minimum \( y \)-coordinate among all vertices \( w \) above \( v \) and with \( \text{left}(w) = \text{left}(v) \).

Think of a sweep-line algorithm:
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross – each other – edges of $P$

1) Treating split vertices

Connect $v$ to vertex $w^*$ having minimum $y$-coordinate among all vertices $w$ above $v$ and with $\text{left}(w) = \text{left}(v)$.

Think of a sweep-line algorithm:
Towards an Algorithm

Idea: Add diagonals to “destroy” split and merge vertices.

Problem: Diagonals must not cross – each other – edges of $P$

1) Treating split vertices

Connect $v$ to vertex $w^*$ having minimum $y$-coordinate among all vertices $w$ above $v$ and with $\text{left}(w) = \text{left}(v)$.

Think of a sweep-line algorithm:
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross
- each other
- edges of \( P \)

1) Treating split vertices

Connect \( v \) to vertex \( w^* \) having minimum \( y \)-coordinate among all vertices \( w \) above \( v \) and with \( \text{left}(w) = \text{left}(v) \).

Think of a sweep-line algorithm:

Connect \( v \) to \( \text{helper} (\text{left}(v)) \).
Towards an Algorithm

**Idea:** Add diagonals to “destroy” split and merge vertices.

**Problem:** Diagonals must not cross
- each other
- edges of \( P \)

1) Treating split vertices

Connect \( v \) to vertex \( w^* \) having minimum \( y \)-coordinate among all vertices \( w \) above \( v \) and with \( \text{left}(w) = \text{left}(v) \).

Think of a sweep-line algorithm:

Connect \( v \) to \( \text{helper}(\text{left}(v)) \).
An Algorithm

2) Treating merge vertices
An Algorithm

2) Treating merge vertices
An Algorithm

2) Treating merge vertices
An Algorithm

2) Treating merge vertices
An Algorithm

2) Treating merge vertices
An Algorithm

2) Treating merge vertices
An Algorithm

2) Treating merge vertices

\[
\text{makeMonotone}(\text{polygon } P)
\]
\[
\mathcal{D} \leftarrow \text{DCEL}(V(P), E(P))
\]
\[
\mathcal{Q} \leftarrow \text{priority queue on } V(P)
\]
\[
\mathcal{T} \leftarrow \text{empty bin. search tree}
\]
An Algorithm

2) Treating merge vertices

\[ \text{makeMonotone}(\text{polygon } P) \]
\[ D \leftarrow \text{DCEL}(V(P), E(P)) \]
\[ Q \leftarrow \text{priority queue on } V(P) \]
\[ T \leftarrow \text{empty bin. search tree} \]

\{ \text{doubly-connected edge list:} \]
\{ \text{data structure for planar subdivisions} \}
An Algorithm

2) Treating merge vertices

makeMonotone(polygon \( P \))
\[ D \leftarrow \text{DCEL}(V(P), E(P)) \]
\[ Q \leftarrow \text{priority queue on } V(P) \]
\[ T \leftarrow \text{empty bin. search tree} \]

\( (x, y) \prec (x', y') : \iff y > y' \lor (y = y' \land x < x') \)

\text{doubly-connected edge list: data structure for planar subdivisions}
An Algorithm

2) Treating merge vertices

```
makeMonotone(polygon P)
D ← DCEL(V(P), E(P))
Q ← priority queue on V(P)
T ← empty bin. search tree
while Q ≠ ∅ do
    v ← Q.extractMax()
    type ← type of vertex v
    handleTypeVertex(v)
return DCEL D
```

```
doubly-connected edge list:
data structure for planar subdivisions
(x, y) ≺ (x', y') ⇔
y > y' ∨ (y = y' ∧ x < x')

∈ start, split, end, merge, regular
```
An Algorithm

2) Treating merge vertices

\[
\text{makeMonotone}(\text{polygon } P) \\
\quad \mathcal{D} \gets \text{DCEL}(V(P), E(P)) \\
\quad \mathcal{Q} \gets \text{priority queue on } V(P) \\
\quad \mathcal{T} \gets \text{empty bin. search tree} \\
\text{while } \mathcal{Q} \neq \emptyset \text{ do} \\
\quad \quad v \gets \mathcal{Q}.\text{extractMax}() \\
\quad \quad \text{type} \gets \text{type of vertex } v \\
\quad \quad \text{handleTypeVertex}(v) \\
\quad \text{return } \text{DCEL } \mathcal{D}
\]

\[
\text{handleMergeVertex}(\text{vertex } v) \\
\quad e \gets \text{edge following } v \text{ cw} \\
\quad \text{if } \text{helper}(e) \text{ merge vtx then} \\
\quad \quad \mathcal{D}.\text{insert}((v, \text{helper}(e))) \\
\quad \mathcal{T}.\text{delete}(e) \\
\quad e' \gets \mathcal{T}.\text{edgeLeftOf}(v) \\
\quad \text{if } \text{helper}(e') \text{ merge vtx then} \\
\quad \quad \mathcal{D}.\text{insert}((v, \text{helper}(e'))) \\
\quad \text{helper}(e') \gets v
\]
An Algorithm

2) Treating merge vertices

\[
\text{makeMonotone}(\text{polygon } P) \\
D \leftarrow \text{DCEL}(V(P), E(P)) \\
Q \leftarrow \text{priority queue on } V(P) \\
T \leftarrow \text{empty bin. search tree} \\
\text{while } Q \neq \emptyset \text{ do} \\
\quad v \leftarrow Q.\text{extractMax()} \\
\quad \text{type } \leftarrow \text{type of vertex } v \\
\quad \text{handleTypeVertex}(v) \\
\text{return DCEL } D
\]

\[
\text{handleMergeVertex}(\text{vertex } v) \\
e \leftarrow \text{edge following } v \text{ cw} \\
\text{if } \text{helper}(e) \text{ merge vtx then} \\
\quad D.\text{insert(diag}(v, \text{helper}(e))) \\
\quad T.\text{delete}(e) \\
e' \leftarrow T.\text{edgeLeftOf}(v) \\
\text{if } \text{helper}(e') \text{ merge vtx then} \\
\quad D.\text{insert(diag}(v, \text{helper}(e'))) \\
\quad \text{helper}(e') \leftarrow v
\]
An Algorithm

2) Treating merge vertices

```latex
\textbf{makeMonotone}(\text{polygon } P) \\
D \leftarrow \text{DCEL}(V(P), E(P)) \\
Q \leftarrow \text{priority queue on } V(P) \\
T \leftarrow \text{empty bin. search tree} \\
\text{while } Q \neq \emptyset \text{ do} \\
\hspace{1em} v \leftarrow Q.\text{extractMax}() \\
\hspace{1em} \text{type } \leftarrow \text{type of vertex } v \\
\hspace{1em} \text{handleTypeVertex}(v) \\
\text{return } \text{DCEL } D
```

```latex
\textbf{handleMergeVertex}(\text{vertex } v) \\
e \leftarrow \text{edge following } v \text{ cw} \\
\text{if } \text{helper}(e) \text{ merge vtx then} \\
\hspace{1em} D.\text{insert}(\text{diag}(v, \text{helper}(e))) \\
T.\text{delete}(e) \\
e' \leftarrow T.\text{edgeLeftOf}(v) \\
\text{if } \text{helper}(e') \text{ merge vtx then} \\
\hspace{1em} D.\text{insert}(\text{diag}(v, \text{helper}(e'))) \\
\text{helper}(e') \leftarrow v
```
An Algorithm

2) Treating merge vertices

\[
\text{makeMonotone}(\text{polygon } P) \\
D \leftarrow \text{DCEL}(V(P), E(P)) \\
Q \leftarrow \text{priority queue on } V(P) \\
T \leftarrow \text{empty bin. search tree} \\
\text{while } Q \neq \emptyset \text{ do} \\
\quad \text{v} \leftarrow Q.\text{extractMax}() \\
\quad \text{type} \leftarrow \text{type of vertex } v \\
\quad \text{handleTypeVertex}(v) \\
\text{return } \text{DCEL } D
\]

\[
\text{handleMergeVertex}(\text{vertex } v) \\
\quad e \leftarrow \text{edge following } v \text{ cw} \\
\quad \text{if } \text{helper}(e) \text{ merge vtx then} \\
\quad \quad D.\text{insert(diag}(v, \text{helper}(e))) \\
\quad \quad T.\text{delete}(e) \\
\quad \quad e' \leftarrow T.\text{edgeLeftOf}(v) \\
\quad \quad \text{if } \text{helper}(e') \text{ merge vtx then} \\
\quad \quad \quad D.\text{insert(diag}(v, \text{helper}(e'))) \\
\quad \quad \text{helper}(e') \leftarrow v
\]
An Algorithm

2) Treating merge vertices

```
makeMonotone(polygon P)
D ← DCEL(V(P), E(P))
Q ← priority queue on V(P)
T ← empty bin. search tree
while Q ≠ ∅ do
    v ← Q.extractMax()
    type ← type of vertex v
    handleTypeVertex(v)
return DCEL D
```

```
handleMergeVertex(vertex v)
e ← edge following v cw
if helper(e) merge vtx then
    D.insert(diag(v, helper(e)))
    T.delete(e)
e′ ← T.edgeLeftOf(v)
if helper(e′) merge vtx then
    D.insert(diag(v, helper(e′)))
    helper(e′) ← v
```

An Algorithm

2) Treating merge vertices

\[\text{makeMonotone}(\text{polygon } P)\]
\[\begin{align*}
D & \leftarrow \text{DCEL}(V(P), E(P)) \\
Q & \leftarrow \text{priority queue on } V(P) \\
T & \leftarrow \text{empty bin. search tree}
\end{align*}\]

\[\text{while } Q \neq \emptyset \text{ do}
\begin{align*}
\text{\quad } v & \leftarrow Q.\text{extractMax()} \\
\text{\quad } \text{type} & \leftarrow \text{type of vertex } v \\
\text{\quad } \text{handleTypeVertex}(v)
\end{align*}\]

\[\text{return } \text{DCEL } D\]

\[\text{handleMergeVertex}(\text{vertex } v)\]
\[\begin{align*}
e & \leftarrow \text{edge following } v \text{ cw} \\
\text{if } \text{helper}(e) \text{ merge vtx } \text{ then}
\begin{align*}
\quad & \text{D.insert(diag}(v, \text{helper}(e))) \\
\quad & T.\text{delete}(e) \\
\quad & e' \leftarrow T.\text{edgeLeftOf}(v) \\
\text{if } \text{helper}(e') \text{ merge vtx } \text{ then}
\begin{align*}
\quad & \text{D.insert(diag}(v, \text{helper}(e'))) \\
\quad & \text{helper}(e') \leftarrow v
\end{align*}
\end{align*}\]
An Algorithm

2) Treating merge vertices

makeMonotone(polygon $P$)

$D \leftarrow \text{DCEL}(V(P), E(P))$

$Q \leftarrow \text{priority queue on } V(P)$

$T \leftarrow \text{empty bin. search tree}$

while $Q \neq \emptyset$ do

$\quad v \leftarrow Q.\text{extractMax}()$

$\quad \text{type} \leftarrow \text{type of vertex } v$

handleTypeVertex($v$)

return DCEL $D$

handleMergeVertex(vertex $v$)

$e \leftarrow \text{edge following } v \text{ cw}$

if helper($e$) merge vtx then

$\quad D.\text{insert}($\text{diag}(v, \text{helper}(e)))$

$\quad T.\text{delete}($e$)$

$e' \leftarrow T.\text{edgeLeftOf}(v)$

if helper($e'$) merge vtx then

$\quad D.\text{insert}($\text{diag}(v, \text{helper}(e'))$)$

helper($e'$) $\leftarrow v$
An Algorithm

2) Treating merge vertices

makeMonotone(polygon P)
\[ D \leftarrow DCEL(V(P), E(P)) \]
\[ Q \leftarrow \text{priority queue on } V(P) \]
\[ T \leftarrow \text{empty bin. search tree} \]

while \( Q \neq \emptyset \) do
\[ v \leftarrow Q\text{.extractMax()} \]
\[ \text{type} \leftarrow \text{type of vertex } v \]
\[ \text{handleTypeVertex}(v) \]

return DCEL \( D \)

handleMergeVertex(vertex v)
\[ e \leftarrow \text{edge following } v \text{ cw} \]
if helper(e) merge vtx then
\[ D\text{.insert(diag}(v, \text{helper}(e))) \]
\[ T\text{.delete}(e) \]
\[ e' \leftarrow T\text{.edgeLeftOf}(v) \]
if helper(e') merge vtx then
\[ D\text{.insert(diag}(v, \text{helper}(e'))) \]
\[ \text{helper}(e') \leftarrow v \]
An Algorithm

2) Treating merge vertices

makeMonotone(polygon \( P \))
\[
D \leftarrow \text{DCEL}(V(P), E(P))
\]
\[
Q \leftarrow \text{priority queue on } V(P)
\]
\[
T \leftarrow \text{empty bin. search tree}
\]
while \( Q \neq \emptyset \) do
\[
\begin{align*}
&v \leftarrow Q.\text{extractMax}() \\
&type \leftarrow \text{type of vertex } v \\
&\text{handleTypeVertex}(v)
\end{align*}
\]
return \( \text{DCEL } D \)

handleMergeVertex(vertex \( v \))
\[
e \leftarrow \text{edge following } v \text{ cw}
\]
if helper(\( e \)) merge vtx then
\[
D.\text{insert}(\text{diag}(v, \text{helper}(e)))
\]
\[
T.\text{delete}(e)
\]
e′ \leftarrow T.\text{edgeLeftOf}(v)
if helper(\( e' \)) merge vtx then
\[
D.\text{insert}(\text{diag}(v, \text{helper}(e')))
\]
helper(\( e' \)) \leftarrow v
An Algorithm

2) Treating merge vertices

```plaintext
makeMonotone(polygon P)
D ← DCEL(V(P), E(P))
Q ← priority queue on V(P)
T ← empty bin. search tree
while Q ≠ ∅ do
    v ← Q.extractMax()
    type ← type of vertex v
    handleTypeVertex(v)
return DCEL D

handleMergeVertex(vertex v)
e ← edge following v cw
if helper(e) merge vtx then
    D.insert(diag(v, helper(e)))
    T.delete(e)
e' ← T.edgeLeftOf(v)
if helper(e') merge vtx then
    D.insert(diag(v, helper(e')))
    helper(e') ← v
```

Diagram:
- Polygon with vertices and edges labeled.
- Vertex v and edges e and e' highlighted.
- Priority queue and search tree representations.
An Algorithm

2) Treating merge vertices

\begin{verbatim}
makeMonotone(polygon P) 
D ← DCEL(V(P), E(P))
Q ← priority queue on V(P)
T ← empty bin. search tree
while Q ≠ ∅ do 
    v ← Q.extractMax()
    type ← type of vertex v
    handleTypeVertex(v)
return DCEL D

handleMergeVertex(vertex v)
    e ← edge following v cw
    if helper(e) merge vtx then 
        D.insert(diag(v, helper(e)))
        T.delete(e)
    e′ ← T.edgeLeftOf(v)
    if helper(e′) merge vtx then
        D.insert(diag(v, helper(e′)))
    helper(e′) ← v
\end{verbatim}
Analysis

Lemma. makeMonotone() adds a set of non-intersecting diagonals to $P$ such that $P$ is partitioned into $y$-monotone subpolygons.
Analysis

**Lemma.** makeMonotone() adds a set of non-intersecting diagonals to $P$ such that $P$ is partitioned into $y$-monotone subpolygons.

**Lemma.** A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a \( y \)-Monotone Polygon \( P \)

**Approach:** greedy, going from top to bottom
Triangulating a \( y \)-Monotone Polygon \( P \)

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

Approach: greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

Approach: greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
**Triangulating a $y$-Monotone Polygon $P$**

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

Invariant?
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

Invariant?
Triangulating a $y$-Monotone Polygon $P$

Approach: greedy, going from top to bottom

Invariant?
Approach: greedy, going from top to bottom

Invariant?
Triangulating a \( y \)-Monotone Polygon \( P \)

**Approach:** greedy, going from top to bottom

Invariant?
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

Invariant?
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of $P$ that we have seen but not yet triangulated is a *funnel*. 
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of $P$ that we have seen but not yet triangulated is a **funnel**.

chains of reflex vtc
Triangulating a \( y \)-Monotone Polygon \( P \)

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of \( P \) that we have seen but not yet triangulated is a *funnel*.

- angle in \( P \) > 180°
- reflex vtc

chains of reflex vtc
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of $P$ that we have seen but not yet triangulated is a *funnel*.

- angle in $P > 180^\circ$
- reflex vtc
- convex vtc.
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of $P$ that we have seen but not yet triangulated is a *funnel*.

Our funnels are special:

- chains of reflex vtc
- convex vtc.

angle in $P > 180^\circ$ reflex vtc convex vtc.
Triangulating a \( y \)-Monotone Polygon \( P \)

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of \( P \) that we have seen but not yet triangulated is a *funnel*.

Our funnels are special:

- Just 1 chain!

---

angle in \( P \) > 180°

reflex vtc

convex vtc.

chains of reflex vtc

just 1 chain!
Triangulating a $y$-Monotone Polygon $P$

**Approach:** greedy, going from top to bottom

**Invariant?**

The part of $P$ that we have seen but not yet triangulated is a *funnel*.

Our funnels are special:
- Chains of reflex vtc
- Just 1 chain!

Easy!
**Algorithm**

**TriangulateMonotonePolygon** *(Polygon $P$ as circular vertex list)*

merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $i \leftarrow 3$ to $n - 1$ do
Algorithm

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)

merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $j \leftarrow 3$ to $n - 1$ do

- if $u_j$ and $S$.top() lie on different chains then
  - while not $S$.empty() do
  - draw diagonal ($u_j$, $v$)
  - $S$.push($u_{j-1}$);
  - $S$.push($u_j$)

else

- 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($u_j$)

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon \( P \) as circular vertex list)
merge left and right chain \( \rightarrow \) sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S \); \( S \).push(\( u_1 \)); \( S \).push(\( u_2 \))
for \( j \leftarrow 3 \) to \( n - 1 \) do
  if \( u_j \) and \( S \).top() lie on different chains then
    while not \( S \).empty() do
      \( v \leftarrow S \).pop()
      if not \( S \).empty() then draw diag. \((u_j, v)\)
    \end{while}
  else
    \( v \leftarrow S \).pop()
    while not \( S \).empty() and \( u_j \) sees \( S \).top() do
      \( v \leftarrow S \).pop()
      draw diagonal \((u_j, v)\)
    \end{while}
  \end{if}
draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one
Algorithm

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)
merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. ($u_j, v$)

else

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
**Algorithm**

**TriangulateMonotonePolygon**(Polygon \( P \) as circular vertex list)

merge left and right chain \( \rightarrow \) sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)

Stack \( S; S.push(u_1); S.push(u_2) \)

for \( j \leftarrow 3 \) to \( n - 1 \) do

  if \( u_j \) and \( S.top() \) lie on different chains then

    while not \( S.empty() \) do

      \( v \leftarrow S.pop() \)

      if not \( S.empty() \) then draw diag. \((u_j, v)\)

  else

    \[ \text{draw diagonals from } u_n \text{ to all vtc on } S \text{ except first and last one} \]

\[
\text{draw diagonals from } u_n \text{ to all vtc on } S \text{ except first and last one}
\]
Algorithm

TriangulateMonotonePolygon(Polygon P as circular vertex list)
merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n-1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. ($u_j, v$)
  else

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)

merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $j \leftarrow 3$ to $n - 1$ do

  if $u_j$ and $S$.top() lie on different chains then

    while not $S$.empty() do

      $v \leftarrow S$.pop()

      if not $S$.empty() then draw diag. ($u_j, v$)

  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($u_j$)

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon $P$ as circular vertex list)
merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. $(u_j, v)$
  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($u_j$)

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
**Algorithm**

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)
merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $j \leftarrow 3$ to $n - 1$ do

  if $u_j$ and $S$.top() lie on different chains then

    while not $S$.empty() do

      $v \leftarrow S$.pop()

      if not $S$.empty() then draw diag. $(u_j, v)$

  else

```
```
Algorithm

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)

merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $j \leftarrow 3$ to $n - 1$ do

  if $u_j$ and $S$.top() lie on different chains then

    while not $S$.empty() do

      $v \leftarrow S$.pop()

      if not $S$.empty() then draw diag. $(u_j, v)$

  else

  draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)

merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $j \leftarrow 3$ to $n - 1$ do

  if $u_j$ and $S$.top() lie on different chains then

    while not $S$.empty() do

      $v \leftarrow S$.pop()

      if not $S$.empty() then draw diag. $(u_j, v)$

    $S$.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($u_j$)

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon P as circular vertex list)
merge left and right chain → sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S; S.push(u_1); S.push(u_2) \)
for \( j \leftarrow 3 \) to \( n-1 \) do
    if \( u_j \) and \( S.top() \) lie on different chains then
        while not \( S.empty() \) do
            \( \nu \leftarrow S.pop() \)
            if not \( S.empty() \) then draw diag. \( (u_j, \nu) \)
            \( S.push(u_{j-1}); S.push(u_j) \)
    else

draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one
Algorithm

**TriangulateMonotonePolygon** (Polygon \( P \) as circular vertex list)

merge left and right chain → sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)

Stack \( S \); \( S.push(u_1) \); \( S.push(u_2) \)

for \( j \leftarrow 3 \) to \( n - 1 \) do

  if \( u_j \) and \( S.top() \) lie on different chains then

    while not \( S.empty() \) do

      \( v \leftarrow S.pop() \)

      if not \( S.empty() \) then draw diag. \( (u_j, v) \)

    \( S.push(u_{j-1}) \); \( S.push(u_j) \)

else


draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon $P$ as circular vertex list)
merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. ($u_j$, $v$)
      $S$.push($u_{j-1}$); $S$.push($u_j$)
  else
    $v \leftarrow S$.pop()
Algorithm

TriangulateMonotonePolygon(Polygon $P$ as circular vertex list)
merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S.push(u_1)$; $S.push(u_2)$
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S.top()$ lie on different chains then
    while not $S.empty()$ do
      $v \leftarrow S.pop()$
      if not $S.empty()$ then draw diag. $(u_j, v)$
    $S.push(u_{j-1})$; $S.push(u_j)$
  else
    $v \leftarrow S.pop()$
    while not $S.empty()$ and $u_j$ sees $S.top()$ do
      $v \leftarrow S.pop()$
      draw diagonal $(u_j, v)$

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

\textbf{TriangulateMonotonePolygon}(Polygon \ P \ as \ circular \ vertex \ list)

merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack \(S; \ S.\text{push}(u_1); \ S.\text{push}(u_2)\)

\textbf{for} \(j \leftarrow 3 \ \textbf{to} \ n - 1 \ \textbf{do}\)

\quad \textbf{if} \ \(u_j \ \text{and} \ S.\text{top}() \ \text{lie on different chains} \ \textbf{then}\)

\quad \quad \textbf{while} \ \textbf{not} \ \ S.\text{empty}() \ \textbf{do}

\quad \quad \quad \nu \leftarrow S.\text{pop}()

\quad \quad \quad \textbf{if} \ \textbf{not} \ \ S.\text{empty}() \ \textbf{then} \ \text{draw diagonal} \ (u_j, \nu)

\quad \quad \textbf{S.}\text{push}(u_{j-1}); \ S.\text{push}(u_j)

\quad \textbf{else}

\quad \quad \nu \leftarrow S.\text{pop}()

\quad \quad \textbf{while} \ \textbf{not} \ \ S.\text{empty}() \ \textbf{and} \ u_j \ \text{sees} \ S.\text{top}() \ \textbf{do}

\quad \quad \quad \nu \leftarrow S.\text{pop}()

\quad \quad \quad \text{draw diagonal} \ (u_j, \nu)

\quad \textbf{S.}\text{push}(\nu)

\textbf{draw diagonals from} \ \(u_n\ \text{to all vtc on} \ S\ \text{except first and last one} \)
Algorithm

TriangulateMonotonePolygon(Polygon $P$ as circular vertex list)
merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diagonal $(u_j, v)$
    $S$.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 $(u_j, v)$
  draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon $P$ as circular vertex list)
merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. $(u_j, v)$
      $S$.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 $(u_j, v)$

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon P as circular vertex list)
merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
    if $u_j$ and $S$.top() lie on different chains then
        while not $S$.empty() do
            $v \leftarrow S$.pop()
            if not $S$.empty() then draw diagonal $(u_j, v)$
        $S$.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 $(u_j, v)$

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon P as circular vertex list)
merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. ($u_j$, $v$)
    $S$.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 ($u_j$, $v$)
  draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

\textbf{TriangulateMonotonePolygon}(Polygon P as circular vertex list)
merge left and right chain \rightarrow sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S \); \( S.push(u_1); S.push(u_2) \)
for \( j \leftarrow 3 \) to \( n - 1 \) do
  if \( u_j \) and \( S.top() \) lie on different chains then
    while not \( S.empty() \) do
      \( v \leftarrow S.pop() \)
      if not \( S.empty() \) then draw diagonal \((u_j, v)\)
      \( S.push(u_{j-1}); S.push(u_j) \)
  else
    \( v \leftarrow S.pop() \)
    while not \( S.empty() \) and \( u_j \) sees \( S.top() \) do
      \( v \leftarrow S.pop() \)
      draw diagonal \((u_j, v)\)

draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon P as circular vertex list)
merge left and right chain → sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S \); \( S.push(u_1); S.push(u_2) \)
for \( j \leftarrow 3 \) to \( n - 1 \) do
  if \( u_j \) and \( S.top() \) lie on different chains then
    while not \( S.empty() \) do
      \( v \leftarrow S.pop() \)
      if not \( S.empty() \) then draw diag. \((u_j, v)\)

    \( S.push(u_{j-1}); S.push(u_j) \)
  else
    \( v \leftarrow S.pop() \)
    while not \( S.empty() \) and \( u_j \) sees \( S.top() \) do
      \( v \leftarrow S.pop() \)
      draw diagonal \((u_j, v)\)
**Algorithm**

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)
merge left and right chain → sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$
Stack $S$; $S$.push($u_1$); $S$.push($u_2$)
for $j \leftarrow 3$ to $n - 1$ do
  if $u_j$ and $S$.top() lie on different chains then
    while not $S$.empty() do
      $v \leftarrow S$.pop()
      if not $S$.empty() then draw diag. $(u_j, v)$
      $S$.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($u_j$)

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
**Algorithm**

**TriangulateMonotonePolygon** (Polygon \( P \) as circular vertex list)

merge left and right chain → sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S \); \( S.push(u_1); S.push(u_2) \)

for \( j \leftarrow 3 \) to \( n - 1 \) do
  if \( u_j \) and \( S.top() \) lie on different chains then
    while not \( S.empty() \) do
      \( v \leftarrow S.pop() \)
      if not \( S.empty() \) then draw diag. \((u_j, v)\)
    \( S.push(u_j-1); S.push(u_j) \)
  else
    \( v \leftarrow S.pop() \)
    while not \( S.empty() \) and \( u_j \) sees \( S.top() \) do
      \( v \leftarrow S.pop() \)
      draw diagonal \((u_j, v)\)
    \( S.push(v); S.push(u_j) \)

draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one
Algorithm

**TriangulateMonotonePolygon** (Polygon $P$ as circular vertex list)

merge left and right chain $\rightarrow$ sequence $u_1, \ldots, u_n$ with $y_1 \geq \cdots \geq y_n$

Stack $S$; $S$.push($u_1$); $S$.push($u_2$)

for $j \leftarrow 3$ to $n - 1$ do

\[
\text{if } u_j \text{ and } S\.\text{top}() \text{ lie on different chains then}
\]

\[
\text{while not } S\.\text{empty}() \text{ do}
\]

\[
\quad v \leftarrow S\.\text{pop}()
\]

\[
\quad \text{if } \neg S\.\text{empty}() \text{ then draw diag. } (u_j, v)
\]

\[
S\.push(u_{j-1}); S\.push(u_j)
\]

else

\[
\quad v \leftarrow S\.\text{pop}()
\]

\[
\quad \text{while not } S\.\text{empty}() \text{ and } u_j \text{ sees } S\.\text{top}() \text{ do}
\]

\[
\quad \quad v \leftarrow S\.\text{pop}()
\]

\[
\quad \text{draw diagonal } (u_j, v)
\]

\[
S\.push(v); S\.push(u_j)
\]

draw diagonals from $u_n$ to all vtc on $S$ except first and last one
Algorithm

TriangulateMonotonePolygon(Polygon P as circular vertex list)
merge left and right chain → sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S \); \( S.\text{push}(u_1); \ S.\text{push}(u_2) \)
for \( j \leftarrow 3 \) to \( n - 1 \) do
  if \( u_j \) and \( S.\text{top()} \) lie on different chains then
    while not \( S.\text{empty()} \) do
      \( v \leftarrow S.\text{pop()} \)
      if not \( S.\text{empty()} \) then draw diagonal \( (u_j, v) \)
    \( S.\text{push}(u_{j - 1}); \ S.\text{push}(u_j) \)
  else
    \( v \leftarrow S.\text{pop()} \)
    while not \( S.\text{empty()} \) and \( u_j \) sees \( S.\text{top()} \) do
      \( v \leftarrow S.\text{pop()} \)
      draw diagonal \( (u_j, v) \)
    \( S.\text{push}(v); \ S.\text{push}(u_j) \)
draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one

Running time?
Algorithm

TriangulateMonotonePolygon(Polygon \( P \) as circular vertex list)
merge left and right chain \( \rightarrow \) sequence \( u_1, \ldots, u_n \) with \( y_1 \geq \cdots \geq y_n \)
Stack \( S; S.push(u_1); S.push(u_2) \)
for \( j \leftarrow 3 \) to \( n - 1 \) do
  if \( u_j \) and \( S.top() \) lie on different chains then
    while not \( S.empty() \) do
      \( v \leftarrow S.pop() \)
      if not \( S.empty() \) then draw diagonal \( (u_j, v) \)
    \( S.push(u_{j-1}); S.push(u_j) \)
  else
    \( v \leftarrow S.pop() \)
    while not \( S.empty() \) and \( u_j \) sees \( S.top() \) do
      \( v \leftarrow S.pop() \)
      draw diagonal \( (u_j, v) \)
    \( S.push(v); S.push(u_j) \)
draw diagonals from \( u_n \) to all vtc on \( S \) except first and last one

Running time? \( \Theta(n) \)
Summary

\[
\begin{align*}
n\text{-vtx polygon} & \rightarrow \text{“nice” pieces, } n' \text{ vtc} & \rightarrow n'' \text{ triangles} \\
& \mathcal{O}(n \log n) & \mathcal{O}(n')
\end{align*}
\]
Summary

Lemma. A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.
Summary

A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

**Lemma.** A simple polygon with \( n \) vertices can be subdivided into \( y \)-monotone polygons in \( O(n \log n) \) time.

**Lemma.** A \( y \)-monotone polygon with \( n \) vertices can be triangulated in \( O(n) \) time.

**Lemma.** Subdividing a simple polygon with \( n \) vertices by drawing \( d \) (pairwise non-crossing) diagonals yields \( d + 1 \) simple polygons of total complexity \( O(n) \).
Summary

**Lemma.** A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

**Lemma.** Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

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

**Lemma.** A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

**Lemma.** Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

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

Is this it?
Summary

Lemma. A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

Lemma. Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

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

Is this it? Tarjan & van Wyk [1988]:
Summary

**Lemma.** A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

**Lemma.** Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

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

**Is this it?** Tarjan & van Wyk [1988]: $O(n \log \log n)$
Summary

Lemma. A simple polygon with \( n \) vertices can be subdivided into \( y \)-monotone polygons in \( O(n \log n) \) time.

Lemma. A \( y \)-monotone polygon with \( n \) vertices can be triangulated in \( O(n) \) time.

Lemma. Subdividing a simple polygon with \( n \) vertices by drawing \( d \) (pairwise non-crossing) diagonals yields \( d + 1 \) simple polygons of total complexity \( O(n) \).

Theorem. A simple polygon with \( n \) vertices can be triangulated in \( O(n \log n) \) time.

Is this it? Tarjan & van Wyk [1988]: \( O(n \log \log n) \)

Clarkson, Tarjan, van Wyk [1989]:
A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

A simple polygon with $n$ vertices can be triangulated in $O(n \log n)$ time.

- Tarjan & van Wyk [1988]: $O(n \log \log n)$
- Clarkson, Tarjan, van Wyk [1989]: $O(n \log^* n)$
Summary

**Lemma.** A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

**Lemma.** Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

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

**Is this it?**
- Tarjan & van Wyk [1988]: $O(n \log \log n)$
- Clarkson, Tarjan, van Wyk [1989]: $O(n \log^* n)$
- Chazelle [1991]: $O(n \log^* n)$
A simple polygon with $n$ vertices can be subdivided into $y$-monotone polygons in $O(n \log n)$ time.

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

Subdividing a simple polygon with $n$ vertices by drawing $d$ (pairwise non-crossing) diagonals yields $d + 1$ simple polygons of total complexity $O(n)$.

A simple polygon with $n$ vertices can be triangulated in $O(n \log n)$ time.

Is this it?

Tarjan & van Wyk [1988]: $O(n \log \log n)$
Clarkson, Tarjan, van Wyk [1989]: $O(n \log^* n)$
Chazelle [1991]: $O(n)$
Summary

**Lemma.** A simple polygon with \( n \) vertices can be subdivided into \( y \)-monotone polygons in \( O(n \log n) \) time.

**Lemma.** A \( y \)-monotone polygon with \( n \) vertices can be triangulated in \( O(n) \) time.

**Lemma.** Subdividing a simple polygon with \( n \) vertices by drawing \( d \) (pairwise non-crossing) diagonals yields \( d + 1 \) simple polygons of total complexity \( O(n) \).

**Theorem.** A simple polygon with \( n \) vertices can be triangulated in \( O(n \log n) \) time.

**Is this it?**
- Tarjan & van Wyk [1988]: \( O(n \log \log n) \)
- Clarkson, Tarjan, van Wyk [1989]: \( O(n \log^* n) \)
- Chazelle [1991]: \( O(n) \)
- Kirkpatrick, Klawe, Tarjan [1992]: \( O(n \log n) \)
Summary

Lemma. A simple polygon with \( n \) vertices can be subdivided into \( y \)-monotone polygons in \( O(n \log n) \) time.

- *old*

Lemma. A \( y \)-monotone polygon with \( n \) vertices can be triangulated in \( O(n) \) time.

- *new*

Lemma. Subdividing a simple polygon with \( n \) vertices by drawing \( d \) (pairwise non-crossing) diagonals yields \( d + 1 \) simple polygons of total complexity \( O(n) \).

- *homework*

Theorem. A simple polygon with \( n \) vertices can be triangulated in \( O(n \log n) \) time.

**Is this it?**

- Tarjan & van Wyk [1988]: \( O(n \log \log n) \)
- Clarkson, Tarjan, van Wyk [1989]: \( O(n \log^* n) \)
- Chazelle [1991]: \( O(n) \)
- Kirkpatrick, Klawe, Tarjan [1992]: \( O(n \log n) \)
- Seidel [1991]: randomized