Polygon Triangulation

Outline

I. $y$-monotone polygon

II. Partitioning a simple polygon into $y$-monotone pieces

III. Triangulating a $y$-monotone polygon
Brute-Force Triangulation

1. Find a diagonal.
2. Triangulate the two resulting subpolygons recursively.
1. Find a diagonal.

2. Triangulate the two resulting subpolygons recursively.

How to find a diagonal?
Brute-Force Triangulation

1. Find a diagonal.
2. Triangulate the two resulting subpolygons recursively.

How to find a diagonal?
Brute-Force Triangulation

1. Find a diagonal.
2. Triangulate the two resulting subpolygons recursively.

How to find a diagonal?

leftmost vertex

case 1

closest to v

case 2
1. Find a diagonal.

2. Triangulate the two resulting subpolygons recursively.

How to find a diagonal?

*Case 1:* Leftmost vertex

*Case 2:* Closest to $v$
Brute-Force Triangulation

1. Find a diagonal.
2. Triangulate the two resulting subpolygons recursively.

How to find a diagonal?

\( O(n) \) time to find a diagonal at every step.
\( \Theta(n^2) \) time in the worst case.
Triangulating a Convex Polygon
Triangulating a Convex Polygon
Triangulating a Convex Polygon

$\Theta(n)$ time!
Triangulating a Convex Polygon

Idea:
- Decompose a simple polygon into convex pieces.
- Triangulate the pieces.

$\Theta(n)$ time!
Triangulating a Convex Polygon

Idea:
- Decompose a simple polygon into convex pieces.
- Triangulate the pieces.

$\Theta(n)$ time!
Triangulating a Convex Polygon

Idea:
- Decompose a simple polygon into convex pieces.
- Triangulate the pieces.

$\Theta(n)$ time!
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.
I. $y$-monotone Pieces

$y$-monotone if any line perpendicular to the $y$-axis has a connected intersection with the polygon.

Walk always downward or horizontal.

Highest vertex

Lowest vertex
I. \( y \)-monotone Pieces

\( y \)-monotone if any line perpendicular to the \( y \)-axis has a connected intersection with the polygon.

**Strategy:**

Partition the polygon into monotone pieces and then triangulate.
Turn Vertex

**Turn vertex** is where the walk from highest vertex to the lowest vertex switches direction.
Turn vertex is where the walk from highest vertex to the lowest vertex switches direction.
Turn vertex is where the walk from highest vertex to the lowest vertex switches direction.
Turn vertex is where the walk from highest vertex to the lowest vertex switches direction.

At vertex $\nu$

- Both adjacent edges are below.
- The polygon interior lies above.
Turn vertex is where the walk from highest vertex to the lowest vertex switches direction.

At vertex $v$

- Both adjacent edges are below.
- The polygon interior lies above.

Choose a diagonal that goes up.
Five Types of Vertices

Point \( p = (x, y) \) is “below” a different point \( q = (u, v) \) if

\[
y < v \quad \text{or} \quad y = v \text{ and } x > u.
\]
Point $p = (x, y)$ is “below” a different point $q = (u, v)$ if $y < v$ or $y = v$ and $x > u$. 

\[ q \] 
\[ p \]
Five Types of Vertices

Point $p = (x, y)$ is "below" a different point $q = (u, v)$ if

$y < v$ or $y = v$ and $x > u$. 
Five Types of Vertices

Point $p = (x, y)$ is “below” a different point $q = (u, v)$ if

$$y < v \quad \text{or} \quad y = v \text{ and } x > u.$$ 

Otherwise $p$ is “above” $q$. 
Five Types of Vertices

Point $p = (x, y)$ is “below” a different point $q = (u, v)$ if

$$y < v \quad \text{or} \quad y = v \text{ and } x > u.$$ 

Otherwise $p$ is “above” $q$. 
Five Types of Vertices

Point \( p = (x, y) \) is “below” a different point \( q = (u, v) \) if

\[
y < v \quad \text{or} \quad y = v \text{ and } x > u.
\]

Otherwise \( p \) is “above” \( q \).
Five Types of Vertices

Point \( p = (x, y) \) is “below” a different point \( q = (u, v) \) if

\[
y < v \quad \text{or} \quad y = v \quad \text{and} \quad x > u.
\]

Otherwise \( p \) is “above” \( q \).

- **Start vertex**: lies above its two neighbors and has interior angle \(< \pi\).
- **Split vertex**: lies above its two neighbors and has interior angle \(> \pi\).
- **End vertex**: lies below its two neighbors and has interior angle \(< \pi\).
- **Merge vertex**: lies below its two neighbors and has interior angle \(> \pi\).
- **Regular vertex**: the remaining vertices (no turn)
Five Types of Vertices

Point \( p = (x, y) \) is “below” a different point \( q = (u, v) \) if

\[
y < v \quad \text{or} \quad y = v \text{ and } x > u.
\]

Otherwise \( p \) is “above” \( q \).

4 types of turn vertices

- **Start vertex**: lies above its two neighbors and has interior angle \(< \pi\).
- **Split vertex**: lies above its two neighbors and has interior angle \(> \pi\).
- **End vertex**: lies below its two neighbors and has interior angle \(< \pi\).
- **Merge vertex**: lies below its two neighbors and has interior angle \(> \pi\).

- **Regular vertex**: the remaining vertices (no turn).
Point \( p = (x, y) \) is “below” a different point \( q = (u, v) \) if
\[ y < v \quad \text{or} \quad y = v \text{ and } x > u. \]
Otherwise \( p \) is “above” \( q \).

What happens if we rotate the polygon by \( \pi \)?
Five Types of Vertices

Point $p = (x, y)$ is “below” a different point $q = (u, v)$ if

$y < v$  \hspace{1em} or \hspace{1em}  $y = v$ and $x > u$.

Otherwise $p$ is “above” $q$.

What happens if we rotate the polygon by $\pi$?

- **Start vertex**: lies above its two neighbors and has interior angle $< \pi$.
- **Split vertex**: lies above its two neighbors and has interior angle $> \pi$.
- **End vertex**: lies below its two neighbors and has interior angle $< \pi$.
- **Merge vertex**: lies below its two neighbors and has interior angle $> \pi$.
- **Regular vertex**: the remaining vertices (no turn)

start vertices $\Leftrightarrow$ end vertices  \hspace{1em}  split vertices $\Leftrightarrow$ merge vertices
Local Non-Monotonicity

Lemma A polygon is $y$-monotone if it has no split or merge vertices.
**Local Non-Monotonicity**

**Lemma** A polygon is $\gamma$-monotone if it has no split or merge vertices.

**Proof** Suppose the polygon is not $\gamma$-monotone. We prove that it contains a split or merge vertex.
Lemma A polygon is $y$-monotone if it has no split or merge vertices.

Proof Suppose the polygon is not $y$-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line $l$ intersecting the polygon in $> 1$ components, with leftmost segment $\overline{pq}$ ($p$ left and $q$ right).
**Local Non-Monotonicity**

**Lemma** A polygon is \( y \)-monotone if it has no split or merge vertices.

**Proof** Suppose the polygon is not \( y \)-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line \( l \) intersecting the polygon in > 1 components, with leftmost segment \( \overline{pq} \) (\( p \) left and \( q \) right).

Start at \( q \), traverse the boundary counterclockwise until it crosses the line \( l \) again at point \( r \).
**Lemma** A polygon is $y$-monotone if it has no split or merge vertices.

**Proof** Suppose the polygon is not $y$-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line $l$ intersecting the polygon in $> 1$ components, with leftmost segment $\overline{pq}$ ($p$ left and $q$ right).

Start at $q$, traverse the boundary counterclockwise until it crosses the line $l$ again at point $r$.

$r \neq p$
Local Non-Monotonicity

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

**Proof** Suppose the polygon is not $y$-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line $l$ intersecting the polygon in $>1$ components, with leftmost segment $pq$ ($p$ left and $q$ right).

Start at $q$, traverse the boundary counterclockwise until it crosses the line $l$ again at point $r$.

$r = p$ and $r \neq p$.
Lemma A polygon is $y$-monotone if it has no split or merge vertices.

Proof Suppose the polygon is not $y$-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line $l$ intersecting the polygon in $> 1$ components, with leftmost segment $\overline{pq}$ ($p$ left and $q$ right).

Start at $q$, traverse the boundary counterclockwise until it crosses the line $l$ again at point $r$.

$r = p$

The highest vertex during the traversal from $q$ to $r$ must be a split vertex.
**Lemma** A polygon is $y$-monotone if it has no split or merge vertices.

**Proof** Suppose the polygon is not $y$-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line $l$ intersecting the polygon in $>1$ components, with leftmost segment $pq$ ($p$ left and $q$ right).

Start at $q$, traverse the boundary counterclockwise until it crosses the line $l$ again at point $r$.

$r = p$
Traverse in the opposite direction from $q$ and crosses line $l$ again at point $r'$. The lowest point during this traversal must be a merge vertex.

$r \neq p$
The highest vertex during the traversal from $q$ to $r$ must be a split vertex.
**Local Non-Monotonicity**

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

**Proof** Suppose the polygon is not $y$-monotone. We prove that it contains a split or merge vertex.

There exists a horizontal line $l$ intersecting the polygon in $>1$ components, with leftmost segment $pq$ ($p$ left and $q$ right).

Start at $q$, traverse the boundary counterclockwise until it crosses the line $l$ again at point $r$.

$r = p$

Traverse in the opposite direction from $q$ and crosses line $l$ again at point $r'$. The lowest point during this traversal must be a merge vertex.

$r \neq p$

The highest vertex during the traversal from $q$ to $r$ must be a split vertex.
II. Partitioning into Monotone Pieces

The lemma implies that the polygon will have $y$-monotone pieces once its split and merge vertices are removed.
II. Partitioning into Monotone Pieces

The lemma implies that the polygon will have $y$-monotone pieces once its split and merge vertices are removed.

- Add a downward diagonal at every merge vertex.
II. Partitioning into Monotone Pieces

The lemma implies that the polygon will have $y$-monotone pieces once its split and merge vertices are removed.

- Add a downward diagonal at every merge vertex.
- Add an upward diagonal at every split vertex.
II. Partitioning into Monotone Pieces

The lemma implies that the polygon will have $y$-monotone pieces once its split and merge vertices are removed.

- Add a downward diagonal at every merge vertex.
- Add an upward diagonal at every split vertex.

Use a downward plane sweep.
II. Partitioning into Monotone Pieces

The lemma implies that the polygon will have $\gamma$-monotone pieces once its split and merge vertices are removed.

- Add a downward diagonal at every merge vertex.
- Add an upward diagonal at every split vertex.

Use a downward plane sweep.

- No new event point will be created except the vertices.
- The event queue is implemented as priority queue (e.g., heap).

\[ v_1, v_2, \ldots, v_n \]
II. Partitioning into Monotone Pieces

The lemma implies that the polygon will have $y$-monotone pieces once its split and merge vertices are removed.

- Add a downward diagonal at every merge vertex.
- Add an upward diagonal at every split vertex.

Use a downward plane sweep.

- No new event point will be created except the vertices.
- The event queue is implemented as priority queue (e.g., heap).

$$v_1, v_2, \ldots, v_n$$

- The next event is found in time $O(\log n)$. 

Removal of a Split Vertex
Removal of a Split Vertex

\[ e_i - 1 \]

\[ e_i \]

\[ v_i \]: split vertex
Removal of a Split Vertex

\[ e_{i-1} \rightarrow e_i : \text{edge immediately to its left.} \]

\[ v_i : \text{split vertex} \]

\[ e_j : \text{edge immediately to its left.} \]
Removal of a Split Vertex

\[ e_{i-1} \rightarrow e_i \rightarrow e_{i+1} \rightarrow \ldots \rightarrow e_n \rightarrow e_1 \rightarrow \ldots \rightarrow e_{i-2} \rightarrow e_{i-1} \]

\[ e_j \] : edge immediately to its left.

\[ e_k \] : edge immediately to its right.

\[ v_i \] : split vertex
Removal of a Split Vertex

$e_i - 1 e_j e_k e_i v_i h(e_j)$

$v_i$ : split vertex
$e_j$ : edge immediately to its left.
$e_k$ : edge immediately to its right.
$h(e_j)$: *lowest* vertex above $v_i$ and between $e_j$ and $e_k$. 
Removal of a Split Vertex

\[ e_{i-1} \rightarrow e_{i} \rightarrow e_{i+1} \]

- \( v_i \): split vertex
- \( e_j \): edge immediately to its left.
- \( e_k \): edge immediately to its right.
- \( h(e_j) \): lowest vertex above \( v_i \) and between \( e_j \) and \( e_k \).

(edge helper)
Removal of a Split Vertex

\[ e_{i-1} v_i h(e_j) e_i e_k \]

\( v_i \) : split vertex
\( e_j \) : edge immediately to its left.
\( e_k \) : edge immediately to its right.
\( h(e_j) \): *lowest* vertex above \( v_i \) and between \( e_j \) and \( e_k \).

Connect \( v_i \) to \( h(e_j) \).
$h(e_j)$ could also be one of the following:

a) the upper vertex of $e_j$. 

$h(e_j) = v_j$
$h(e_j)$ could also be one of the following:

a) the upper vertex of $e_j$.

b) the upper vertex of $e_k$. 

$h(e_j) = v_j$
$h(e_j)$ could also be one of the following:

a) the upper vertex of $e_j$.

b) the upper vertex of $e_k$.

$h(e_j) = v_j$

$v_{k+1} = h(e_j)$
Helper Vertex

$h(e_j)$ could also be one of the following:

a) the upper vertex of $e_j$.

b) the upper vertex of $e_k$.

Connect $v_i$ to $h(e_j)$. 

$v_{k+1} = h(e_j)$
Removal of a Merge Vertex
Removal of a Merge Vertex

Merge vertices can be handled the same way in an upward sweep as split vertices in a downward sweep.
Removal of a Merge Vertex

- Merge vertices can be handled the same way in an upward sweep as split vertices in a downward sweep.
- But why not all in the same downward sweep?
Removal of a Merge Vertex

Merge vertices can be handled the same way in an upward sweep as split vertices in a downward sweep.

But why not all in the same downward sweep?

\( v_i \) : merge vertex
Removal of a Merge Vertex

Merge vertices can be handled the same way in an upward sweep as split vertices in a downward sweep.

But why not all in the same downward sweep?

- $v_i$: merge vertex
- $e_j$: edge immediately to its left.
Removal of a Merge Vertex

Merge vertices can be handled the same way in an upward sweep as split vertices in a downward sweep.

But why not all in the same downward sweep?

\( v_i \) : merge vertex

\( e_j \) : edge immediately to its left.

\( e_k \) : edge immediately to its right.
Removal of a Merge Vertex

Merge vertices can be handled the same way in an upward sweep as split vertices in a downward sweep.

But why not all in the same downward sweep?

- $v_i$: merge vertex
- $e_j$: edge immediately to its left.
- $e_k$: edge immediately to its right.
- $v_m$: highest vertex below the sweep line and between $e_j$ and $e_k$. (The vertex is unknown when the sweep line reaches $v_i$.)

![Diagram of removal of a merge vertex]
Removal (Cont’d)

When we reach a vertex $v_m$ to replace $v_i$ as the helper of $e_j$. 
Removal (Cont’d)

When we reach a vertex $v_m$ to replace $v_i$ as the helper of $e_j$.

- Check if the old helper is a merge vertex and add the diagonal if so.
Removal (Cont’d)

When we reach a vertex $v_m$ to replace $v_i$ as the helper of $e_j$.

- Check if the old helper is a merge vertex and add the diagonal if so.

Connect $v_i$ to $v_m$. 
Removal (Cont’d)

When we reach a vertex $v_m$ to replace $v_i$ as the helper of $e_j$.

- Check if the old helper is a merge vertex and add the diagonal if so.

Connect $v_i$ to $v_m$.

- The diagonal is always added if $v_m$ is a split vertex.

Get rid of a split vertex and a merge vertex with the same diagonal.
Removal (Cont’d)

When we reach a vertex $v_m$ to replace $v_i$ as the helper of $e_j$.

- Check if the old helper is a merge vertex and add the diagonal if so.

Connect $v_i$ to $v_m$.

- The diagonal is always added if $v_m$ is a split vertex.

Get rid of a split vertex and a merge vertex with the same diagonal.

Also works if $v_m$ is the lower point of $e_j$. 
Removal (Cont’d)

When we reach a vertex $v_m$ to replace $v_i$ as the helper of $e_j$.

- Check if the old helper is a merge vertex and add the diagonal if so.

**Connect $v_i$ to $v_m$.**

- The diagonal is always added if $v_m$ is a split vertex.

Get rid of a split vertex and a merge vertex with the same diagonal.

Also works if $v_m$ is the lower point of $e_j$. 

Sweep-Line Status

Implemented as a binary search tree.

Store edges intersecting the sweep line in the leaves.
Need only store edges that bounds $P$ from the left.

$v_{k+1} = h(e_j)$

♦ $e_k$ is not stored.
♦ $v_{k+1}$ is referenced by $e_j$. 
Sweep-Line Status

Implemented as a binary search tree.

- Store edges intersecting the sweep line in the leaves.
  Need only store edges that bounds $P$ from the left.

- Edges are stored in the left-to-right order.

$\forall e_k$ is not stored.

$\forall v_{k+1}$ is referenced by $e_j$. 

$v_{k+1} = h(e_j)$
Sweep-Line Status

Implemented as a **binary search tree**.

- Store edges intersecting the sweep line in the leaves. Need only store edges that bounds $P$ from the left.

![Diagram](image)

$v_{k+1} = h(e_j)$

- $e_k$ is not stored.
- $v_{k+1}$ is referenced by $e_j$.

- Edges are stored in the left-to-right order.

- With every edge $e$ its helper $h(e)$ is also stored.
DCEL Representation
DCEL Representation

Construct a doubly-connected edge list $D$ to represent the polygon.

\[ n \text{ vertices} + n \text{ edges} + 2 \text{ faces} \quad (\text{initially}) \]
Construct a doubly-connected edge list $D$ to represent the polygon.

$n$ vertices + $n$ edges + 2 faces (initially)

Add in diagonals computed for split and merge vertices.
DCEL Representation

- Construct a doubly-connected edge list $D$ to represent the polygon.

  \[ n \text{ vertices} + n \text{ edges} + 2 \text{ faces} \quad (\text{initially}) \]

- Add in diagonals computed for split and merge vertices.

- Edges in the status BST and corresponding ones in DCEL cross-point each other.

Insertion of a diagonal into DCEL takes $O(1)$ time.
The Algorithm

MakeMonotone(\(P\))

**Input**: A simple polygon \(P\) stored in DCEL \(D\).

**Output**: A partitioning of \(P\) into monotone subpolygons stored in \(D\).

1. \(Q \leftarrow \) priority queue storing vertices of \(P\)  
2. \(T \leftarrow \emptyset\) // initialize the sweep line status as a binary search tree.  
3. \(i \leftarrow 0\)  
4. while \(Q \neq \emptyset\)  
5. do \(v_i \leftarrow \) the highest priority vertex from \(Q\) // removal from \(Q\)  
6. case \(v_i\) of  
7. start vertex: HandleStartVertex(\(v_i\))  
8. end vertex: HandleEndVertex(\(v_i\))  
9. split vertex: HandleSplitVertex(\(v_i\))  
10. merge vertex: HandleMergeVertex(\(v_i\))  
11. regular vertex: HandleRegularVertex(\(v_i\))  
12. \(i \leftarrow i + 1\)
Handling Start & End Vertices

HandleStartVertex\left(v_i\right)
1. \textbf{T} \leftarrow \textbf{T} \cup \{ e_i \}
2. \text{h}(e_i) \leftarrow v_i \text{ // set the helper}
Handling Start & End Vertices

HandleStartVertex($v_i$)
1. $T \leftarrow T \cup \{ e_i \}$
2. $h(e_i) \leftarrow v_i$ // set the helper
Handling Start & End Vertices

HandleStartVertex($v_i$)
1. $T \leftarrow T \cup \{e_i\}$
2. $h(e_i) \leftarrow v_i$ // set the helper
Handling Start & End Vertices

HandleStartVertex \( (v_i) \)
1. \( T \leftarrow T \cup \{ e_i \} \)
2. \( h(e_i) \leftarrow v_i \) // set the helper

HandleEndVertex \( (v_i) \)
1. if \( h(e_{i-1}) \) is a merge vertex
2. then insert the diagonal connecting \( v_i \) to \( h(e_{i-1}) \) in \( D \)
3. \( T \leftarrow T - \{ e_{i-1} \} \)
Handling Start & End Vertices

**HandleStartVertex**(\(v_i\))
1. \(T \leftarrow T \cup \{ e_i \}\)
2. \(h(e_i) \leftarrow v_i \) // set the helper

**HandleEndVertex**(\(v_i\))
1. if \(h(e_{i-1})\) is a merge vertex
2. then insert the diagonal connecting \(v_i\) to \(h(e_{i-1})\) in \(D\)
3. \(T \leftarrow T - \{ e_{i-1} \}\)
Handling Start & End Vertices

HandleStartVertex($v_i$)
1. $T \leftarrow T \cup \{e_i\}$
2. $h(e_i) \leftarrow v_i$ // set the helper

HandleEndVertex($v_i$)
1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$

insert into $T$

$i = 5$

$i = 15$
Handling Start & End Vertices

HandleStartVertex($v_i$)
1. $T \leftarrow T \cup \{ e_i \}$
2. $h(e_i) \leftarrow v_i$ \small{\text{// set the helper}}

HandleEndVertex($v_i$)
1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{ e_{i-1} \}$
HandleSplitVertex($v_i$)
1. Search in $T$ to find $e_j$ directly left of $v_i$
2. Insert the diagonal connect $v_i$ to $h(e_j)$ into $D$
3. $h(e_j) \leftarrow v_i$
4. $T \leftarrow T + \{e_i\}$
5. $h(e_i) \leftarrow v_i$
Handling Split Vertex

HandleSplitVertex($v_i$)
1. Search in $T$ to find $e_j$ directly left of $v_i$
2. Insert the diagonal connect $v_i$ to $h(e_j)$ into $D$
3. $h(e_j) \leftarrow v_i$
4. $T \leftarrow T + \{e_i\}$
5. $h(e_i) \leftarrow v_i$
Handling Split Vertex

HandleSplitVertex \( (v_i) \)
1. Search in \( T \) to find \( e_j \) directly left of \( v_i \)
2. insert the diagonal connect \( v_i \) to \( h(e_j) \) into \( D \)
3. \( h(e_j) \leftarrow v_i \)
4. \( T \leftarrow T + \{ e_i \} \)
5. \( h(e_i) \leftarrow v_i \)

\[ i = 14 \]
**Handling Split Vertex**

**HandleSplitVertex**(\(v_i\))

1. Search in \(T\) to find \(e_j\) directly left of \(v_i\)
2. insert the diagonal connect \(v_i\) to \(h(e_j)\) into \(D\)
3. \(h(e_j) \leftarrow v_i\)
4. \(T \leftarrow T + \{e_i\}\)
5. \(h(e_i) \leftarrow v_i\)

\[i = 14 \quad j = 9\]
**Handling Split Vertex**

HandleSplitVertex($v_i$)
1. Search in $T$ to find $e_j$ directly left of $v_i$
2. insert the diagonal connect $v_i$ to $h(e_j)$ into $D$
3. $h(e_j) \leftarrow v_i$
4. $T \leftarrow T + \{e_i\}$
5. $h(e_i) \leftarrow v_i$

$$i = 14 \quad j = 9 \quad v_8 = h(e_9)$$
Handling Split Vertex

\textbf{HandleSplitVertex}(v_i)

1. Search in $T$ to find $e_j$ directly left of $v_i$
2. Insert the diagonal connect $v_i$ to $h(e_j)$ into $D$
3. $h(e_j) \leftarrow v_i$
4. $T \leftarrow T + \{e_i\}$
5. $h(e_i) \leftarrow v_i$

$i = 14 \quad j = 9 \quad v_8 = h(e_9)$
Handling Merge Vertex (1)

HandleMergeVertex($v_i$)

1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$

...
Handling Merge Vertex (1)

HandleMergeVertex(\(v_i\))

1. if \(h(e_{i-1})\) is a merge vertex
2. then insert the diagonal connecting \(v_i\) to \(h(e_{i-1})\) in \(D\)
3. \(T \leftarrow T - \{e_{i-1}\}\)

...
Handling Merge Vertex (1)

HandleMergeVertex($v_i$)

1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$

...
Handling Merge Vertex (1)

HandleMergeVertex($v_i$)
1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$

...
Handling Merge Vertex (1)

HandleMergeVertex($v_i$)
1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$

...
Handling Merge Vertex (2)

HandleMergeVertex($v_i$)
1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$
4. Search in $T$ to find the edge $e_j$ directly left of $v_i$.
5. if $h(e_j)$ is a merge vertex
   then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$
6. $h(e_j) \leftarrow v_i$
Handling Merge Vertex (2)

**HandleMergeVertex**($v_i$)

1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$
4. Search in $T$ to find the edge $e_j$ directly left of $v_i$.
5. if $h(e_j)$ is a merge vertex
   then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$
6. $h(e_j) \leftarrow v_i$
Handling Merge Vertex (2)

HandleMergeVertex($v_i$)

1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$

3. $T \leftarrow T - \{e_{i-1}\}$

4. Search in $T$ to find the edge $e_j$ directly left of $v_i$.

5. if $h(e_j)$ is a merge vertex
    then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$

6. $h(e_j) \leftarrow v_i$
Handling Merge Vertex (2)

HandleMergeVertex($v_i$)
1. if $h(e_{i-1})$ is a merge vertex
2. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
3. $T \leftarrow T - \{e_{i-1}\}$
4. Search in $T$ to find the edge $e_j$ directly left of $v_i$.
5. if $h(e_j)$ is a merge vertex
   then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$
6. $h(e_j) \leftarrow v_i$
Handling Regular Vertices (1)

**HandleRegularVertex**($v_i$)

1. if the polygon interior lies to the right of $v_i$

2. then if $h(e_{i-1})$ is a merge vertex

3. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$

4. $T \leftarrow T - \{e_{i-1}\}$

5. $T \leftarrow T + \{e_i\}$

6. $h(e_i) \leftarrow v_i$

7. else search in $T$ to find the edge $e_j$ directly left of $v_i$.

8. if $h(e_j)$ is a merge vertex then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$

9. $h(e_j) \leftarrow v_i$
Handling Regular Vertices (1)

\textbf{HandleRegularVertex}(v_i)

1. if the polygon interior lies to the right of \(v_i\)

2. then if \(h(e_{i-1})\) is a merge vertex

3. then insert the diagonal connecting \(v_i\) to \(h(e_{i-1})\)

4. in \(D\)

5. \(T \leftarrow T \setminus \{e_{i-1}\}\)

6. \(T \leftarrow T + \{e_i\}\)

7. \(h(e_i) \leftarrow v_i\)

8. else search in \(T\) to find the edge \(e_j\) directly left of \(v_i\).

9. if \(h(e_j)\) is a merge vertex

then insert the diagonal connecting \(v_i\) to \(h(e_j)\)

in \(D\)

9. \(h(e_j) \leftarrow v_i\)
Handling Regular Vertices (1)

HandleRegularVertex($v_i$)
1. if the polygon interior lies to the right of $v_i$
2. then if $h(e_{i-1})$ is a merge vertex
3. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
4. $T \leftarrow T - \{e_{i-1}\}$
5. $T \leftarrow T + \{e_i\}$
6. $h(e_i) \leftarrow v_i$
7. else search in $T$ to find the edge $e_j$ directly left of $v_i$.
8. if $h(e_j)$ is a merge vertex
    then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$
9. $h(e_j) \leftarrow v_i$
Handling Regular Vertices (1)

**HandleRegularVertex**(\(v_i\))

1. if the polygon interior lies to the right of \(v_i\)

2. then if \(h(e_{i-1})\) is a merge vertex

3. then insert the diagonal connecting \(v_i\) to \(h(e_{i-1})\) in \(D\)

4. \(T \leftarrow T - \{e_{i-1}\}\)

5. \(T \leftarrow T + \{e_i\}\)

6. \(h(e_i) \leftarrow v_i\)

7. else search in \(T\) to find the edge \(e_j\) directly left of \(v_i\).

8. if \(h(e_j)\) is a merge vertex

   then insert the diagonal connecting \(v_i\) to \(h(e_j)\) in \(D\)

9. \(h(e_j) \leftarrow v_i\)
Handle Regular Vertices (2)

HandleRegularVertex($v_i$)
1. if the polygon interior lies to the right of $v_i$
2. then if $h(e_{i-1})$ is a merge vertex
3. then insert the diagonal connecting $v_i$ to $h(e_{i-1})$ in $D$
4. $T \leftarrow T - \{e_{i-1}\}$
5. $T \leftarrow T + \{e_i\}$
6. $h(e_i) \leftarrow v_i$
7. else search in $T$ to find the edge $e_j$ directly left of $v_i$.
8. if $h(e_j)$ is a merge vertex
   then insert the diagonal connecting $v_i$ to $h(e_j)$ in $D$
9. $h(e_j) \leftarrow v_i$
Correctness

**Theorem** The algorithm adds a set of non-intersecting diagonals that partitions the polygon into monotone pieces.
Correctness

**Theorem**  The algorithm adds a set of non-intersecting diagonals that partitions the polygon into monotone pieces.

**Proof**  The pieces that result from the partitioning contain no split or merge vertices. Hence they are monotone by an earlier lemma.
Correctness

Theorem  The algorithm adds a set of non-intersecting diagonals that partitions the polygon into monotone pieces.

Proof  The pieces that result from the partitioning contain no split or merge vertices. Hence they are monotone by an earlier lemma.

We need only prove that *the added segments are diagonals that intersect neither the polygon edges nor each other.*
Correctness

**Theorem** The algorithm adds a set of non-intersecting diagonals that partitions the polygon into monotone pieces.

**Proof** The pieces that result from the partitioning contain no split or merge vertices. Hence they are monotone by an earlier lemma.

We need only prove that the added segments are diagonals that intersect neither the polygon edges nor each other.

Establish the above claim for the handling of each of the five type of vertices during the sweep. (Read the textbook on how to do this for the case of a split vertex.)
Running Time on Partitioning

**MakeMonotone**($P$)

**Input:** A simple polygon $P$ stored in DCEL $D$.

**Output:** A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5. do $v_i \leftarrow$ the highest priority vertex from $Q$
6.   case $v_i$ of
7.     start vertex: HandleStartVertex($v_i$)
8.     end vertex: HandleEndVertex($v_i$)
9.     split vertex: HandleSplitVertex($v_i$)
10.    merge vertex: HandleMergeVertex($v_i$)
11.    regular vertex: HanleRegularVertex($v_i$)
12.   $i \leftarrow i + 1$
Running Time on Partitioning

MakeMonotone($P$)

Input: A simple polygon $P$ stored in DCEL $D$.
Output: A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$ // $O(n)$ heap construction
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5.   do $v_i \leftarrow$ the highest priority vertex from $Q$
6.   case $v_i$ of
7.     start vertex: HandleStartVertex($v_i$)
8.     end vertex: HandleEndVertex($v_i$)
9.     split vertex: HandleSplitVertex($v_i$)
10.    merge vertex: HandleMergeVertex($v_i$)
11.    regular vertex: HandleRegularVertex($v_i$)
12.   $i \leftarrow i + 1$
Running Time on Partitioning

MakeMonotone($P$)

Input: A simple polygon $P$ stored in DCEL $D$.

Output: A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$ // $O(n)$ heap construction
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5. do $v_i \leftarrow$ the highest priority vertex from $Q$ // $O(\log n)$
6. case $v_i$ of
7. start vertex: HandleStartVertex($v_i$)
8. end vertex: HandleEndVertex($v_i$)
9. split vertex: HandleSplitVertex($v_i$)
10. merge vertex: HandleMergeVertex($v_i$)
11. regular vertex: HandleRegularVertex($v_i$)
12. $i \leftarrow i + 1$
Running Time on Partitioning

MakeMonotone($P$)

Input: A simple polygon $P$ stored in DCEL $D$.
Output: A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$ // $O(n)$ heap construction
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5. do $v_i \leftarrow$ the highest priority vertex from $Q$ // $O(\log n)$ each case
6. case $v_i$ of
7. start vertex: HandleStartVertex($v_i$) // $O(\log n)$ each case
8. end vertex: HandleEndVertex($v_i$)
9. split vertex: HandleSplitVertex($v_i$)
10. merge vertex: HandleMergeVertex($v_i$)
11. regular vertex: HandleRegularVertex($v_i$)
12. $i \leftarrow i + 1$
MakeMonotone($P$)

**Input:** A simple polygon $P$ stored in DCEL $D$.

**Output:** A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$ // $O(n)$ heap construction
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5. do $v_i \leftarrow$ the highest priority vertex from $Q$ // $O(\log n)$
6. case $v_i$ of
7. start vertex: HandleStartVertex($v_i$) // $O(\log n)$ each case
8. end vertex: HandleEndVertex($v_i$) // queries and updates
9. split vertex: HandleSplitVertx($v_i$) // in $O(\log n)$ time and
10. merge vertex: HandleMergeVertex($v_i$) // insertion of a
11. regular vertex: HanleRegularVertex($v_i$) // diagonal in $O(1)$
12. $i \leftarrow i + 1$
Running Time on Partitioning

MakeMonotone($P$)

**Input:** A simple polygon $P$ stored in DCEL $D$.

**Output:** A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$ /// $O(n)$ heap construction
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5. do $v_i \leftarrow$ the highest priority vertex from $Q$ /// $O(\log n)$
6. case $v_i$ of
7. start vertex: HandleStartVertex($v_i$) /// $O(\log n)$ each case
8. end vertex: HandleEndVertex($v_i$) /// queries and updates in $O(\log n)$ time and
9. split vertex: HandleSplitVertex($v_i$) /// insertion of a
10. merge vertex: HandleMergeVertex($v_i$) /// diagonal in $O(1)$
11. regular vertex: HandleRegularVertex($v_i$) /// time.
12. $i \leftarrow i + 1$

**Total time:** $O(n \log n)$
Running Time on Partitioning

MakeMonotone($P$)

**Input:** A simple polygon $P$ stored in DCEL $D$.
**Output:** A partitioning of $P$ into monotone subpolygons stored in $D$.

1. $Q \leftarrow$ priority queue storing vertices of $P$  // $O(n)$ heap construction
2. $T \leftarrow \emptyset$
3. $i \leftarrow 0$
4. while $Q \neq \emptyset$
5. do $v_i \leftarrow$ the highest priority vertex from $Q$  // $O(\log n)$
6. case $v_i$ of
7. start vertex: HandleStartVertex($v_i$)  // $O(\log n)$ each case
8. end vertex: HandleEndVertex($v_i$)  // queries and updates
9. split vertex: HandleSplitVertex($v_i$)  // in $O(\log n)$ time and
10. merge vertex: HandleMergeVertex($v_i$)  // insertion of a
11. regular vertex: HandleRegularVertex($v_i$)  // diagonal in $O(1)$
12. $i \leftarrow i + 1$

Total time: $O(n \log n)$

Total storage: $O(n)$
Assumption  The polygon is strictly \( y \)-monotone (no horizontal edges).
It is for clarity of presentation and can be easily removed.
III. Triangulating a $\gamma$-Monotone Polygon

**Assumption**  The polygon is strictly $\gamma$-monotone (no horizontal edges).
It is for clarity of presentation and can be easily removed.
Order of processing: in decreasing $\gamma$-coordinate.
III. Triangulating a \( \gamma \)-Monotone Polygon

**Assumption** The polygon is strictly \( \gamma \)-monotone (no horizontal edges).

It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing \( \gamma \)-coordinate.

A stack \( S \): vertices that have been encountered and may still need diagonals.
III. Triangulating a $\gamma$-Monotone Polygon

**Assumption**  The polygon is strictly $\gamma$-monotone (no horizontal edges).

It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing $\gamma$-coordinate.

A stack $S$: vertices that have been encountered and may still need diagonals.

lowest vertex on top.
III. Triangulating a $y$-Monotone Polygon

**Assumption** The polygon is strictly $y$-monotone (no horizontal edges). It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing $y$-coordinate.

A stack $S$: vertices that have been encountered and may still need diagonals. Lowest vertex on top.
Assumption The polygon is strictly \( y \)-monotone (no horizontal edges). It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing \( y \)-coordinate.

A stack \( S \): vertices that have been encountered and may still need diagonals. The lowest vertex on top.

Idea: Add as many diagonals from the current vertex handled to those on the stack as possible.
III. Triangulating a $\gamma$-Monotone Polygon

**Assumption** The polygon is strictly $\gamma$-monotone (no horizontal edges). It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing $\gamma$-coordinate.

A stack $S$: vertices that have been encountered and may still need diagonals. Lowest vertex on top.

**Idea**: Add as many diagonals from the current vertex handled to those on the stack as possible.

**Invariants of iteration**: 
III. Triangulating a $\gamma$-Monotone Polygon

**Assumption**  The polygon is strictly $\gamma$-monotone (no horizontal edges). It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing $\gamma$-coordinate.

A stack $S$: vertices that have been encountered and may still need diagonals. Lowest vertex on top.

**Idea:** Add as many diagonals from the current vertex handled to those on the stack as possible.

**Invariants of iteration:**
- One boundary of the funnel is a polygon edge.
III. Triangulating a $\gamma$-Monotone Polygon

Assumption The polygon is strictly $\gamma$-monotone (no horizontal edges). It is for clarity of presentation and can be easily removed.

Order of processing: in decreasing $\gamma$-coordinate.

A stack $S$: vertices that have been encountered and may still need diagonals. lowest vertex on top.

Idea: Add as many diagonals from the current vertex handled to those on the stack as possible.

Invariants of iteration:

- One boundary of the funnel is a polygon edge.
- The other boundary is a chain of reflex vertices (with interior angles $> \pi$) plus one convex vertex (the highest) at bottom of the stack.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

Pop these vertices from the stack.

Add diagonals from the current vertex to them (except the bottom one) as they are popped.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

Pop these vertices from the stack.

Add diagonals from the current vertex to them (except the bottom one) as they are popped.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

Pop these vertices from the stack.

Add diagonals from the current vertex to them (except the bottom one) as they are popped.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

Pop these vertices from the stack.

Add diagonals from the current vertex to them (except the bottom one) as they are popped.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

Pop these vertices from the stack.

Add diagonals from the current vertex to them (except the bottom one) as they are popped.
This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

Pop these vertices from the stack.

Add diagonals from the current vertex to them (except the bottom one) as they are popped.

Push the previous top of the stack and the current vertex back onto the stack.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

- Pop these vertices from the stack.
- Add diagonals from the current vertex to them (except the bottom one) as they are popped.
- Push the previous top of the stack and the current vertex back onto the stack.
Case 1: Next Vertex on Opposite Chain

This vertex must be the lower endpoint of the single edge $e$ bounding the chain.

- Pop these vertices from the stack.
- Add diagonals from the current vertex to them (except the bottom one) as they are popped.
- Push the previous top of the stack and the current vertex back onto the stack.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

Pop one vertex from the stack.

It shares an edge with the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

Pop one vertex from the stack.

It shares an edge with the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

- Pop one vertex from the stack.
- It shares an edge with the current vertex.
- Pop other vertices from the stack as long as they are visible from the current vertex.
- Draw a diagonal between each of them and the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

- Pop one vertex from the stack.
- It shares an edge with the current vertex.
- Pop other vertices from the stack as long as they are visible from the current vertex.
- Draw a diagonal between each of them and the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

- Pop one vertex from the stack.
- It shares an edge with the current vertex.
- Pop other vertices from the stack as long as they are visible from the current vertex.

Draw a diagonal between each of them and the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

- Pop one vertex from the stack.
- It shares an edge with the current vertex.
- Pop other vertices from the stack as long as they are visible from the current vertex.
- Draw a diagonal between each of them and the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

- Pop one vertex from the stack. It shares an edge with the current vertex.
- Pop other vertices from the stack as long as they are visible from the current vertex.
- Draw a diagonal between each of them and the current vertex.
- Push the last popped vertex back onto the stack followed by the current vertex.
Case 2: Next Vertex on the Same Chain

The vertices that can connect to the current vertex are all on the stack.

- Pop one vertex from the stack.

It shares an edge with the current vertex.

- Pop other vertices from the stack as long as they are visible from the current vertex.

Draw a diagonal between each of them and the current vertex.

- Push the last popped vertex back onto the stack followed by the current vertex.
The Triangulation Algorithm

**TriangulateMonotonePolygon**(\(P\))

**Input**: A *strictly y-monotone* polygon \(P\) stored in DCEL \(D\).

**Output**: A triangulation of \(P\) stored in \(D\).

1. Merge the vertices on the left and right chains into one sequence sorted in decreasing \(y\)-coordinate. (In case there is a tie, the one with smaller \(x\)-coordinate comes first.)
2. Push\((u_1, S)\)
3. Push\((u_2, S)\)
4. for \(j \leftarrow 3 \) to \(n-1\)
5. do  
6.  if \(u_j\) and Top\((S)\) are on different chains  
7.  then while Next(Top\((S)\)) \(\neq \) NULL  
8.  \(v \leftarrow \) Top\((S)\)  
9.  Pop\((S)\)  
10.  insert a diagonal from \(u_j\) to \(v\)
11.  Pop\((S)\)
12.  Push\((u_{j-1}, S)\)
13.  Push\((u_j, S)\)
14. else Pop\((S)\)
15.  while Top\((S)\) is visible from \(u_j\) inside the polygon  
16.  \(v \leftarrow \) Top\((S)\)  
17.  insert a diagonal between \(u_j\) and \(v\)
18.  Pop\((S)\)
19.  Push the last popped vertex back onto \(S\)
20.  Push\((u_j, S)\)
An Example
An Example

Start:

\text{Start:}

\begin{align*}
\text{\text{u_2}} \\
\text{\text{u_1}}
\end{align*}
An Example

\[ \begin{align*}
\text{Start:} & \\
& \begin{array}{c}
\begin{array}{c}
\text{u}_2 \\
\text{u}_1
\end{array}
\end{array}
\end{align*} \]

\[ j = 3: \]
\[ \begin{array}{c}
\begin{array}{c}
\text{u}_2 \\
\text{u}_1
\end{array}
\end{array} \]
An Example

\[ u_i \]

Start:

\[ \begin{array}{c}
\text{Start:} \\
  u_2 \\
  u_1 
\end{array} \]

\[ j = 3: \]

\[ u_1 \]
An Example

Start:

\[
\begin{array}{c}
    u_2 \\
    u_1 \\
\end{array}
\]

\(j = 3:\)

\[
\begin{array}{c}
    u_2 \\
    u_1 \\
\end{array}
\]
An Example

Start:

\[
\begin{align*}
\text{j = 3:} & & \begin{array}{c}
\text{u}_3 \\
\text{u}_2 \\
\text{u}_1
\end{array} \\
\text{j = 4:} & & \begin{array}{c}
\text{u}_3 \\
\text{u}_2 \\
\text{u}_1
\end{array}
\end{align*}
\]
An Example

Start:

\[ j = 3: \]
\[
\begin{array}{c}
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]

\[ j = 4: \]
\[
\begin{array}{c}
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]
An Example

Start:

\[
\begin{array}{c}
\hline
u_2 \\
u_1 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\hline
u_3 \\
u_2 \\
u_1 \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\hline
u_1 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\hline
u_1 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\hline
\hline
u_1 \\
\hline
\end{array}
\]
An Example

Start:

\[
\begin{array}{c}
  u_2 \\
  u_1 \\
\end{array}
\]

\[ j = 3: \quad \begin{array}{c}
  u_3 \\
  u_2 \\
  u_1 \\
\end{array} \]

\[ j = 4: \quad \begin{array}{c}
  u_1 \\
\end{array} \]
An Example

Start:

\[
\begin{array}{c}
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]

\[j = 3:\]

\[
\begin{array}{c}
\text{u}_3 \\
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]

\[j = 4:\]

\[
\begin{array}{c}
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]
An Example

Start:

\[
\begin{array}{c}
\text{j = 3:} \\
\begin{array}{c}
\underline{u_3} \\
\underline{u_2} \\
\underline{u_1}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{j = 4:} \\
\begin{array}{c}
\underline{u_4} \\
\underline{u_2} \\
\underline{u_1}
\end{array}
\end{array}
\]
An Example

\[ j = 3: \begin{array}{c} u_3 \\ u_2 \\ u_1 \end{array} \quad j = 4: \begin{array}{c} u_4 \\ u_2 \\ u_1 \end{array} \quad j = 5: \begin{array}{c} u_4 \\ u_2 \\ u_1 \end{array} \]
An Example

\[\begin{align*}
\text{Start:} & \quad \begin{array}{c}
\text{u}_2 \\
\text{u}_1
\end{array} \\
\text{j = 3:} & \quad \begin{array}{c}
\text{u}_3 \\
\text{u}_2 \\
\text{u}_1
\end{array} \\
\text{j = 4:} & \quad \begin{array}{c}
\text{u}_4 \\
\text{u}_2 \\
\text{u}_1
\end{array} \\
\text{j = 5:} & \quad \begin{array}{c}
\text{u}_2 \\
\text{u}_1
\end{array}
\end{align*}\]
An Example

Start:

\[
\begin{array}{c}
\text{\( j = 3: \)} \\
\begin{array}{c}
\text{\( u_3 \)} \\
\text{\( u_2 \)} \\
\text{\( u_1 \)}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{\( j = 4: \)} \\
\begin{array}{c}
\text{\( u_4 \)} \\
\text{\( u_2 \)} \\
\text{\( u_1 \)}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{\( j = 5: \)} \\
\begin{array}{c}
\text{\( u_1 \)}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{\( u_2 \)} \\
\text{\( u_1 \)}
\end{array}
\end{array}
\]

An Example

Start:

\[ j = 3: \]
\[
\begin{array}{c}
\mathbf{u_1} \\
\mathbf{u_2} \\
\end{array}
\]

\[ j = 4: \]
\[
\begin{array}{c}
\mathbf{u_1} \\
\mathbf{u_2} \\
\mathbf{u_4} \\
\end{array}
\]

\[ j = 5: \]
\[
\mathbf{u_1}
\]
An Example
An Example

Start:

\[ j = 3: \]
\[
\begin{array}{c}
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]

\[ j = 4: \]
\[
\begin{array}{c}
\text{u}_4 \\
\text{u}_2 \\
\text{u}_1 \\
\end{array}
\]

\[ j = 5: \]
An Example

Start:

\[
\begin{align*}
&u_2 \\
&u_1
\end{align*}
\]

\[
\begin{align*}
&j = 3: \\
&u_3 \\
&u_2 \\
&u_1
\end{align*}
\]

\[
\begin{align*}
&j = 4: \\
&u_4 \\
&u_2 \\
&u_1
\end{align*}
\]

\[
\begin{align*}
&j = 5: \\
&u_1
\end{align*}
\]
An Example

Start:

\[ j = 3: \]
\[ \begin{array}{c}
  u_3 \\
  u_2 \\
  u_1 \\
\end{array} \]

\[ j = 4: \]
\[ \begin{array}{c}
  u_4 \\
  u_2 \\
  u_1 \\
\end{array} \]

\[ j = 5: \]
\[ \begin{array}{c}
  u_5 \\
  u_1 \\
\end{array} \]
An Example

Start:

\begin{align*}
  j = 3: & \quad u_3 & \quad u_2 & \quad u_1 \\
  j = 4: & \quad u_4 & \quad u_2 & \quad u_1 \\
  j = 5: & \quad u_5 & \quad u_1 \\
  j = 6: & \quad u_1
\end{align*}
An Example

Start:

\[ j = 3: \]
\[ u_3 \]
\[ u_2 \]
\[ u_1 \]

\[ j = 4: \]
\[ u_4 \]
\[ u_2 \]
\[ u_1 \]

\[ j = 5: \]
\[ u_5 \]
\[ u_1 \]

\[ j = 6: \]
\[ u_1 \]
An Example

\[ j = 3: \]
\[
\begin{array}{c}
\text{Start:} \\
\hline
u_1 \\
\hline
u_2 \\
\hline
u_3 \\
\hline
u_4 \\
\hline
j = 4: \\
\begin{array}{c}
\text{Start:} \\
\hline
u_1 \\
\hline
u_2 \\
\hline
u_3 \\
\hline
u_4 \\
\hline
j = 5: \\
\begin{array}{c}
\text{Start:} \\
\hline
u_1 \\
\hline
u_2 \\
\hline
u_3 \\
\hline
u_4 \\
\hline
j = 6: \\
\begin{array}{c}
\text{Start:} \\
\hline
u_1 \\
\hline
u_2 \\
\hline
u_3 \\
\hline
u_4 \\
\hline
\end{array}
\end{array}
\end{array}
\]
An Example

Start:

\[ \begin{array}{c}
\text{u}_2 \\
\text{u}_1 \\
\end{array} \]

\[ \begin{array}{c}
\text{u}_3 \\
\text{u}_2 \\
\text{u}_1 \\
\end{array} \]

\[ \begin{array}{c}
\text{u}_5 \\
\text{u}_4 \\
\text{u}_3 \\
\end{array} \]

\[ \begin{array}{c}
\text{u}_5 \\
\text{u}_4 \\
\text{u}_3 \\
\end{array} \]

\[ \begin{array}{c}
\text{u}_5 \\
\text{u}_4 \\
\text{u}_3 \\
\end{array} \]

\[ \begin{array}{c}
\text{u}_5 \\
\text{u}_4 \\
\text{u}_3 \\
\end{array} \]
An Example

Start:

\[ j = 3: \begin{align*} u_3 & \ \ u_2 \\ u_2 & \ \ u_1 \end{align*} \]

\[ j = 4: \begin{align*} u_4 & \ \ u_2 \\ u_2 & \ \ u_1 \end{align*} \]

\[ j = 5: \begin{align*} u_5 & \ \ u_1 \\ u_1 & \ \ u_1 \end{align*} \]

\[ j = 6: \begin{align*} u_6 & \ \ u_1 \\ u_1 & \ \ u_5 \end{align*} \]
Example (cont’d)

\[ u_6 u_5 j = 7: \]

\[
\begin{align*}
u_1 & \quad u_2 \\
u_3 & \quad u_4 \\
u_5 & \\
u_6 & \\
u_7 & \quad u_8 \\
u_9 & \\
u_{10} & \quad u_{11} \\
u_{12} & \quad u_{13} \\
u_{14} & \\
u_{15} & \\
u_{16} & \\
u_{17} & \\
u_{18} & \\
u_{19} & \\
u_{20} & \\
\end{align*}
\]
Example (cont’d)

\[ j = 7: \]

\( u_5 \)
Example (cont’d)

\[ j = 7: \]

\[ u_5 \]
Example (cont’d)

\[ j = 7: \]
Example (cont’d)

$j = 7$:

$u_6$
Example (cont’d)

\[ j = 7: \]

\[
\begin{array}{c}
u_7 \\
u_6
\end{array}
\]
Example (cont’d)

\[ j = 7: \]
\[ u_7 \]
\[ u_6 \]

\[ j = 8: \]
\[ u_7 \]
\[ u_6 \]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
\quad u_7 \\
\quad u_6
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
\quad u_6
\end{array}
\]
Example (cont’d)

\[ j = 7: \]

\[
\begin{align*}
&u_7 \\
&u_6
\end{align*}
\]

\[ j = 8: \]

\[
\begin{align*}
&u_6
\end{align*}
\]
Example (cont’d)

\[ j = 7: \]

\[
\begin{align*}
\{u_7, u_6\}
\end{align*}
\]

\[ j = 8: \]

\[
\begin{align*}
\{u_7, u_6\}
\end{align*}
\]
Example (cont’d)

\[ j = 7: \]

\[ j = 8: \]
Example (cont’d)

\( j = 7: \)

\[
\begin{align*}
&u_7 \\
&u_6
\end{align*}
\]

\( j = 8: \)

\[
\begin{align*}
&u_8 \\
&u_7
\end{align*}
\]
Example (cont’d)

\[ j = 7: \begin{array}{c} u_7 \\ u_6 \end{array} \]

\[ j = 8: \begin{array}{c} u_8 \\ u_7 \end{array} \]

\[ j = 9: \begin{array}{c} u_9 \\ u_8 \end{array} \]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
\hline
u_7 \\
\hline
u_6 \\
\hline
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
\hline
u_8 \\
\hline
u_7 \\
\hline
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{c}
\hline
u_9 \\
\hline
u_8 \\
\hline
\end{array}
\]

\[ j = 10: \]
\[
\begin{array}{c}
\hline
u_9 \\
\hline
u_8 \\
\hline
\end{array}
\]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
  u_7 \\
  u_6
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
  u_8 \\
  u_7 \\
  u_8
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{c}
  u_9 \\
  u_8
\end{array}
\]

\[ j = 10: \]
\[
\begin{array}{c}
  u_8
\end{array}
\]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
\text{u}_7 \\
\text{u}_6 \\
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
\text{u}_8 \\
\text{u}_7 \\
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{c}
\text{u}_9 \\
\text{u}_8 \\
\end{array}
\]

\[ j = 10: \]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
\text{u7} \\
\text{u6}
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
\text{u8} \\
\text{u7}
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{c}
\text{u9} \\
\text{u8}
\end{array}
\]

\[ j = 10: \]
Example (cont’d)

\[ j = 7: \]
\[ \begin{align*}
  & u_7 \\
  & u_6 
\end{align*} \]

\[ j = 8: \]
\[ \begin{align*}
  & u_8 \\
  & u_7 
\end{align*} \]

\[ j = 9: \]
\[ \begin{align*}
  & u_9 \\
  & u_8 
\end{align*} \]

\[ j = 10: \]
\[ u_8 \]
Example (cont’d)

\[ j = 7: \]

\[
\begin{array}{c}
\text{u}_7 \\
\text{u}_6 \\
\end{array}
\]

\[ j = 8: \]

\[
\begin{array}{c}
\text{u}_8 \\
\text{u}_7 \\
\end{array}
\]

\[ j = 9: \]

\[
\begin{array}{c}
\text{u}_9 \\
\text{u}_8 \\
\end{array}
\]

\[ j = 10: \]

\[
\begin{array}{c}
\text{u}_{10} \\
\text{u}_8 \\
\end{array}
\]
Example (cont’d)
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
\hline
u_7 \\
\hline
u_6 \\
\hline
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
\hline
u_8 \\
\hline
u_7 \\
\hline
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{c}
\hline
u_9 \\
\hline
u_8 \\
\hline
\end{array}
\]

\[ j = 10: \]
\[
\begin{array}{c}
\hline
u_{10} \\
\hline
u_8 \\
\hline
\end{array}
\]

\[ j = 11: \]
\[
\begin{array}{c}
\hline
u_8 \\
\hline
\end{array}
\]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{c}
\text{u}_7 \\
\text{u}_6
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{c}
\text{u}_8 \\
\text{u}_7
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{c}
\text{u}_9 \\
\text{u}_8
\end{array}
\]

\[ j = 10: \]
\[
\begin{array}{c}
\text{u}_{10} \\
\text{u}_8
\end{array}
\]

\[ j = 11: \]
\[
\begin{array}{c}
\text{u}_8
\end{array}
\]
Example (cont’d)

\[ j = 7: \]
\[
\begin{array}{l}
  u_7 \\
  u_6 \\
\end{array}
\]

\[ j = 8: \]
\[
\begin{array}{l}
  u_8 \\
  u_7 \\
\end{array}
\]

\[ j = 9: \]
\[
\begin{array}{l}
  u_9 \\
  u_8 \\
\end{array}
\]

\[ j = 10: \]
\[
\begin{array}{l}
  u_{10} \\
  u_8 \\
\end{array}
\]

\[ j = 11: \]
\[
\begin{array}{l}
  u_8 \\
  u_8 \\
\end{array}
\]
Example (cont’d)

\[ j = 7: \]
\[ u_7 \]
\[ u_6 \]

\[ j = 8: \]
\[ u_8 \]
\[ u_7 \]

\[ j = 9: \]
\[ u_9 \]
\[ u_8 \]

\[ j = 10: \]
\[ u_{10} \]
\[ u_8 \]

\[ j = 11: \]
\[ u_{10} \]
Example (cont’d)

\[ j = 7: \]
\[ u_7 \]
\[ u_6 \]

\[ j = 8: \]
\[ u_8 \]
\[ u_7 \]

\[ j = 9: \]
\[ u_9 \]
\[ u_8 \]

\[ j = 10: \]
\[ u_{10} \]
\[ u_8 \]

\[ j = 11: \]
\[ u_{11} \]
\[ u_{10} \]
Example (Cont’d)
Example (Cont’d)
Example (Cont’d)

\[ j = 15: \]

\[
\begin{array}{c}
\text{u}_{15} \\
\text{u}_{10}
\end{array}
\]
Example (Cont’d)

\[ j = 15: \]

\[
\begin{align*}
\begin{array}{c}
\textbf{u}_{15} \\
\textbf{u}_{10}
\end{array}
\end{align*}
\]

\[ j = 16: \]

\[
\begin{align*}
\begin{array}{c}
\textbf{u}_{15} \\
\textbf{u}_{10}
\end{array}
\end{align*}
\]
Example (Cont’d)

\[ j = 15: \]

\[ u_{15} \]

\[ u_{10} \]

\[ j = 16: \]

\[ u_{16} \]

\[ u_{15} \]

\[ u_{10} \]
Example (Cont’d)

\[ j = 15: \]
\[
\begin{array}{c}
\text{\( u_{15} \)} \\
\text{\( u_{10} \)}
\end{array}
\]

\[ j = 16: \]
\[
\begin{array}{c}
\text{\( u_{16} \)} \\
\text{\( u_{15} \)} \\
\text{\( u_{10} \)}
\end{array}
\]

\[ j = 17: \]
\[
\begin{array}{c}
\text{\( u_{16} \)} \\
\text{\( u_{15} \)} \\
\text{\( u_{10} \)}
\end{array}
\]
Example (Cont’d)

\[ j = 15: \]
\[
\begin{array}{c}
\begin{array}{c}
\text{u}_{15} \\
\text{u}_{10}
\end{array}
\end{array}
\]

\[ j = 16: \]
\[
\begin{array}{c}
\begin{array}{c}
\text{u}_{16} \\
\text{u}_{15} \\
\text{u}_{10}
\end{array}
\end{array}
\]

\[ j = 17: \]
\[
\begin{array}{c}
\begin{array}{c}
\text{u}_{15} \\
\text{u}_{10}
\end{array}
\end{array}
\]
Example (Cont’d)

\[ j = 15: \]
\[ u_{15} \]
\[ u_{10} \]

\[ j = 16: \]
\[ u_{16} \]
\[ u_{15} \]
\[ u_{10} \]

\[ j = 17: \]
\[ u_{10} \]
Example (Cont’d)

\[ j = 15: \]
\[ u_{10} \]
\[ u_{15} \]

\[ j = 16: \]
\[ u_{16} \]
\[ u_{15} \]
\[ u_{10} \]

\[ j = 17: \]
\[ u_{10} \]
Example (Cont’d)

\[ j = 15: \]

\[
\begin{align*}
&u_{15} \\
&u_{10} 
\end{align*}
\]

\[ j = 16: \]

\[
\begin{align*}
&u_{16} \\
&u_{15} \\
&u_{10} 
\end{align*}
\]

\[ j = 17: \]

\[
\begin{align*}
&u_{16} \\
&u_{15} \\
&u_{10} 
\end{align*}
\]
Example (Cont’d)

\[ j = 15: \]

\begin{align*}
&u_{15} \\
&u_{10}
\end{align*}

\[ j = 16: \]

\begin{align*}
&u_{16} \\
&u_{15} \\
&u_{10}
\end{align*}

\[ j = 17: \]

\begin{align*}
&u_{15} \\
&u_{10}
\end{align*}
Example (Cont’d)

\[ j = 15: \]

\[ u_{15} \]

\[ u_{10} \]

\[ j = 16: \]

\[ u_{16} \]

\[ u_{15} \]

\[ u_{10} \]

\[ j = 17: \]

\[ u_{10} \]
Example (Cont’d)

\[ j = 15: \]
\[
\begin{array}{c}
\mathbf{u}_{15} \\
\mathbf{u}_{10}
\end{array}
\]

\[ j = 16: \]
\[
\begin{array}{c}
\mathbf{u}_{16} \\
\mathbf{u}_{15} \\
\mathbf{u}_{10}
\end{array}
\]

\[ j = 17: \]
\[
\begin{array}{c}
\mathbf{u}_{17} \\
\mathbf{u}_{10}
\end{array}
\]
Example (Cont’d)

\[ j = 15: \]
\[
\begin{array}{ccc}
  u_{15} \\
  u_{10} \\
\end{array}
\]

\[ j = 16: \]
\[
\begin{array}{ccc}
  u_{16} \\
  u_{15} \\
  u_{10} \\
\end{array}
\]

\[ j = 17: \]
\[
\begin{array}{ccc}
  u_{17} \\
  u_{10} \\
\end{array}
\]

\[ j = 18: \]
\[
\begin{array}{ccc}
  u_{17} \\
  u_{10} \\
\end{array}
\]
Example (Cont’d)

\[ j = 15: \quad \begin{array}{c} u_{15} \\ u_{10} \end{array} \]

\[ j = 16: \quad \begin{array}{c} u_{16} \\ u_{15} \\ u_{10} \end{array} \]

\[ j = 17: \quad \begin{array}{c} u_{17} \\ u_{15} \\ u_{10} \end{array} \]

\[ j = 18: \quad u_{10} \]
Example (Cont’d)

\[ j = 15: \]
\[ u_{15}, u_{10} \]

\[ j = 16: \]
\[ u_{16}, u_{15}, u_{10} \]

\[ j = 17: \]
\[ u_{17}, u_{10}, u_{10} \]

\[ j = 18: \]
\[ u_{10} \]
Example (Cont’d)

\[ j = 15: \]

\[ j = 16: \]

\[ j = 17: \]

\[ j = 18: \]
Example (Cont’d)

\[ j = 15: \]
\[
\begin{align*}
_u{15} \\
_u{10}
\end{align*}
\]

\[ j = 16: \]
\[
\begin{align*}
_u{16} \\
_u{15} \\
_u{10}
\end{align*}
\]

\[ j = 17: \]
\[
\begin{align*}
_u{17} \\
_u{10}
\end{align*}
\]

\[ j = 18: \]
\[
\begin{align*}
_u{18} \\
_u{17}
\end{align*}
\]
Example (Cont’d)

\( j = 15: \)  
\[
\begin{array}{c}
\text{u}_{15} \\
\text{u}_{10}
\end{array}
\]

\( j = 16: \)  
\[
\begin{array}{c}
\text{u}_{16} \\
\text{u}_{15} \\
\text{u}_{10}
\end{array}
\]

\( j = 17: \)  
\[
\begin{array}{c}
\text{u}_{17} \\
\text{u}_{10}
\end{array}
\]

\( j = 18: \)  
\[
\begin{array}{c}
\text{u}_{18} \\
\text{u}_{17}
\end{array}
\]

\( j = 19: \)  
\[
\begin{array}{c}
\text{u}_{18} \\
\text{u}_{17}
\end{array}
\]
Example (Cont’d)

\[ j = 15: \quad u_{15} \]

\[ j = 16: \quad u_{16} \quad u_{15} \quad u_{10} \]

\[ j = 17: \quad u_{17} \quad u_{10} \]

\[ j = 18: \quad u_{18} \quad u_{17} \]

\[ j = 19: \quad u_{17} \]
Example (Cont’d)

\[ j = 15: \]
\[
\begin{align*}
u_{15} \\
u_{10}
\end{align*}
\]

\[ j = 16: \]
\[
\begin{align*}
u_{16} \\
u_{15} \\
u_{10}
\end{align*}
\]

\[ j = 17: \]
\[
\begin{align*}
u_{17} \\
u_{10}
\end{align*}
\]

\[ j = 18: \]
\[
\begin{align*}
u_{18} \\
u_{17}
\end{align*}
\]

\[ j = 19: \]
Example (Cont’d)

\[ j = 15: \]

\[
\begin{array}{c}
\text{u15} \\
\text{u10}
\end{array}
\]

\[ j = 16: \]

\[
\begin{array}{c}
\text{u16} \\
\text{u15} \\
\text{u10}
\end{array}
\]

\[ j = 17: \]

\[
\begin{array}{c}
\text{u17} \\
\text{u10}
\end{array}
\]

\[ j = 18: \]

\[
\begin{array}{c}
\text{u18} \\
\text{u17}
\end{array}
\]

\[ j = 19: \]

\[
\begin{array}{c}
\text{u19} \\
\text{u17}
\end{array}
\]
Removal of Strict $y$-monotonicity

The running time of $\text{TriangulateMonotonePolygon}$ is $\Theta(n)$. 

- $\#\text{pushes} \leq 2n - 4$ (at most 2 vertices pushed in each of $n - 3$ iteration steps plus 2 at the beginning)
- $\#\text{pops} \leq \#\text{pushes}$
Removal of Strict $y$-monotonicity

The running time of $\text{TriangulateMonotonePolygon}$ is $\Theta(n)$.

- #pushes $\leq 2n - 4$ ($\leq 2$ vertices pushed in each of $n - 3$ iteration steps plus 2 at the beginning)
- #pops $\leq$ #pushes

What to do if some vertices have the same $y$-coordinates?
Removal of Strict y-monotonicity

The running time of TriangulateMonotonePolygon is $\Theta(n)$.

- #pushes $\leq 2n - 4$ (at most 2 vertices pushed in each of $n - 3$ iteration steps plus 2 at the beginning)
- #pops $\leq$ #pushes

What to do if some vertices have the same $y$-coordinates?

- Treat them from left to right.
Removal of Strict y-monotonicity

The running time of TriangulateMonotonePolygon is $\Theta(n)$.

- $\#\text{pushes} \leq 2n - 4$ ($\leq 2$ vertices pushed in each of $n - 3$ iteration steps plus 2 at the beginning)
- $\#\text{pops} \leq \#\text{pushes}$

What to do if some vertices have the same $y$-coordinates?

- Treat them from left to right.
- The effect of this is equivalent to that of rotating the plane slightly clockwise and then every vertex will have different $y$ coordinate.
Time Complexity of Triangulation
Time Complexity of Triangulation

1. Partition a simple polygon into monotone pieces.

\( O(n \log n) \)
Time Complexity of Triangulation

1. Partition a simple polygon into monotone pieces.
   \[ O(n \log n) \]

2. Triangulate each monotone piece.
   \[ \Theta(n) \] for all monotone pieces together
Time Complexity of Triangulation

1. Partition a simple polygon into monotone pieces.

\[ O(n \log n) \]

2. Triangulate each monotone piece.

\[ \Theta(n) \] for all monotone pieces together

**Theorem** A simple polygon can be triangulated in \( O(n \log n) \) time and \( O(n) \) storage.
Triangulation of a Planar Subdivision

The algorithm for splitting a polygon into monotone pieces does *not* use the fact that the polygon was simple.
Triangulation of a Planar Subdivision

The algorithm for splitting a polygon into monotone pieces does \textit{not} use the fact that the polygon was simple.

- The plane sweep for decomposition of a polygon into monotone pieces takes as input only edges that lie to the left of the interior.
Triangulation of a Planar Subdivision

The algorithm for splitting a polygon into monotone pieces does \textit{not} use the fact that the polygon was simple.

- The plane sweep for decomposition of a polygon into monotone pieces takes as input only edges that lie to the left of the interior.
- This easily generalizes to a planar subdivision in a bounding box.
Triangulation of a Planar Subdivision

The algorithm for splitting a polygon into monotone pieces does not use the fact that the polygon was simple.

- The plane sweep for decomposition of a polygon into monotone pieces takes as input only edges that lie to the left of the interior.
- This easily generalizes to a planar subdivision in a bounding box.

Thus, a planar subdivision with $n$ vertices can also be triangulated in $O(n \log n)$ time using $O(n)$ storage.