Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do \textit{not} intersect.

\textbf{Stage 1:} quick rejection if their bounding boxes \textit{do not} intersect.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect.
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect if and only if \( x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2 \)
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect if and only if \( x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2 \)
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect

if and only if \( x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2 \)

\( L \) right of \( M \)? \( L \) left of \( M \)? \( L \) above \( M \)? \( L \) below \( M \)?
Two Segments Intersect?

One method: 1) solve for the intersection point of the two lines containing the two line segments, and 2) check whether this point lies on both segments.

In practice, the two input segments often do not intersect.

Stage 1: quick rejection if their bounding boxes do not intersect

if and only if  \( x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2 \)

\( L \) right of \( M \)? \( L \) left of \( M \)? \( L \) above \( M \)? \( L \) below \( M \)?

Case 1: bounding boxes do not intersect; neither will the segments.
Case 2: Bounding boxes intersect; the segments may or may not intersect. Needs to be further checked in Stage 2.
Two line segments do not intersect if and only if one segment lies entirely to one side of the line containing the other segment.

Two line segments intersect if and only if the (underlined) condition is false.
Two line segments do \textit{not} intersect if and only if \textbf{one segment} lies entirely to one side of the line containing the other segment.

\[(p_3 - p_2) \times (p_1 - p_2) \text{ and } (p_4 - p_2) \times (p_1 - p_2)\] are both positive!

Two line segments intersect if and only if the (underlined) condition is false.
Negation of the Previous Condition

Two cross products in each pair below have different signs (or at least one cross product in the pair is 0).

\[(p_1 - p_4) \times (p_3 - p_4)\] and \[(p_2 - p_4) \times (p_3 - p_4)\]
\[(p_3 - p_2) \times (p_1 - p_2)\] and \[(p_4 - p_2) \times (p_1 - p_2)\]

Not as convenient as testing the falsity of non-intersecting of the two segments.
Negation of the Previous Condition

Two cross products in each pair below have different signs (or at least one cross product in the pair is 0).

\[(p_1 - p_4) \times (p_3 - p_4) \text{ and } (p_2 - p_4) \times (p_3 - p_4)\]
\[(p_3 - p_2) \times (p_1 - p_2) \text{ and } (p_4 - p_2) \times (p_1 - p_2)\]

Not as convenient as testing the falsity of non-intersecting of the two segments.
Negation of the Previous Condition

Two cross products in each pair below have different signs (or at least one cross product in the pair is 0).

\[(p_1 - p_4) \times (p_3 - p_4) \text{ and } (p_2 - p_4) \times (p_3 - p_4)\]
\[(p_3 - p_2) \times (p_1 - p_2) \text{ and } (p_4 - p_2) \times (p_1 - p_2)\]

// the line through \(p_3, p_4\) // intersects \(p_1 p_2\).

Not as convenient as testing the falsity of non-intersecting of the two segments.
Negation of the Previous Condition

Two cross products in each pair below have different signs (or at least one cross product in the pair is 0).

\[(p_1 - p_4) \times (p_3 - p_4)\] and \[(p_2 - p_4) \times (p_3 - p_4)\]
\[(p_3 - p_2) \times (p_1 - p_2)\] and \[(p_4 - p_2) \times (p_1 - p_2)\]

// the line through \(p_3, p_4\) intersects \(p_1p_2\).

Not as convenient as testing the falsity of non-intersecting of the two segments.
Negation of the Previous Condition

Two cross products in each pair below have different signs (or at least one cross product in the pair is 0).

\[(p_1 - p_4) \times (p_3 - p_4)\] and \[(p_2 - p_4) \times (p_3 - p_4)\]
\[(p_3 - p_2) \times (p_1 - p_2)\] and \[(p_4 - p_2) \times (p_1 - p_2)\]

Not as convenient as testing the falsity of non-intersecting of the two segments.
**Input:** a set of $n$ line segments in the plane.

**Output:** all intersections and for each intersection the involved segments.
A Brute-Force Algorithm

Simply take each pair of segments, and check if they intersect. If so, output the intersection.
A Brute-Force Algorithm

Simply take each pair of segments, and check if they intersect. If so, output the intersection.

Running time $\Theta(n^2)$. 
A Brute-Force Algorithm

Simply take each pair of segments, and check if they intersect. If so, output the intersection.

Running time $\Theta(n^2)$.

Nevertheless, *sparse distribution* in practice:
A Brute-Force Algorithm

Simply take each pair of segments, and check if they intersect. If so, output the intersection.

Running time $\Theta(n^2)$.

Nevertheless, *sparse distribution* in practice:

Most segments do not intersect, or if they do, only with a few other segments.
A Brute-Force Algorithm

Simply take each pair of segments, and check if they intersect. If so, output the intersection.

Running time $\Theta(n^2)$.

Nevertheless, *sparse distribution* in practice:

Most segments do not intersect, or if they do, only with a few other segments.

Need a faster algorithm to deal with such situations!
The Sweeping Algorithm

Avoid testing pairs of segments that are *far apart*. 
The Sweeping Algorithm

Avoid testing pairs of segments that are far apart.
The Sweeping Algorithm

Avoid testing pairs of segments that are *far apart*. 
Avoid testing pairs of segments that are *far apart*.

**Idea**: *imagine* a vertical sweep line passes through the given set of line segments, from left to right.
The Sweeping Algorithm

Avoid testing pairs of segments that are *far apart*.

**Idea:** imagine a vertical sweep line passes through the given set of line segments, from left to right.
The Sweeping Algorithm

Avoid testing pairs of segments that are far apart.

Idea: imagine a vertical sweep line passes through the given set of line segments, from left to right.
The Sweeping Algorithm

Avoid testing pairs of segments that are *far apart*.

**Idea:** *imagine* a vertical sweep line passes through the given set of line segments, from left to right.
Handling Non-degeneracy

If $\geq 1$ vertical segment, imagine all segments are rotated clockwise by a tiny angle and then test for intersection. This means:

For each vertical segment, the sweep line will hit its lower endpoint before upper point.
Sweep Line Status

The set of segments intersecting the sweep line.
Sweep Line Status

The set of segments intersecting the sweep line.
Sweep Line Status

The set of segments intersecting the sweep line.

It changes as the sweep line moves, but not continuously.
Sweep Line Status

*The set of segments intersecting the sweep line.*

It changes as the sweep line moves, but *not continuously.*

Updates of status happen only at *event points.*
**Sweep Line Status**

*The set of segments intersecting the sweep line.*

It changes as the sweep line moves, but *not continuously.*

Updates of status happen only at *event points.*

\[
\text{endpoints, intersections}
\]

\[
A \quad C \quad T
\]
Sweep Line Status

The set of segments intersecting the sweep line.

It changes as the sweep line moves, but *not continuously*.

Updates of status happen only at *event points*.

- endpoints
- intersections

[Diagram showing sweep line and event points]
A *total order* over the segments that intersect the current position of the sweep line:
Ordering Segments

A total order over the segments that intersect the current position of the sweep line:

\[ B > C > D \]

(A and E not in the ordering)
A *total order* over the segments that intersect the current position of the sweep line:

\[ A \succ B \succ C \succ D \succ E \]

\( C \succ D \)

(B drops out of the ordering)
Ordering Segments

A total order over the segments that intersect the current position of the sweep line:

At an event point, the sequence of segments changes:
A total order over the segments that intersect the current position of the sweep line:

At an event point, the sequence of segments changes:

- Update the status.
A total order over the segments that intersect the current position of the sweep line:

At an event point, the sequence of segments changes:

- Update the status.
- Detect the intersections.
Event point is the left endpoint of a segment.
Event point is the left endpoint of a segment.

- A new segment $L$ intersects the sweep line.
Event point is the left endpoint of a segment.

- A new segment $L$ intersects the sweep line.
Event point is the left endpoint of a segment.

- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).
Status Update (1)

Event point is the left endpoint of a segment.

- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).

$K, M, N$ $K, L, M, N$
Status Update (1)

Event point is the left endpoint of a segment.

- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).
Status Update (1)

Event point is the left endpoint of a segment.

- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).
- Intersection(s) are new event points.
Event point is an intersection.
Status Update (2)

Event point is an intersection.

- The two intersecting segments ($L$ and $M$) change order.
Status Update (2)

Event point is an intersection.

- The two intersecting segments $(L$ and $M)$ change order.
Event point is an intersection.

- The two intersecting segments \((L \text{ and } M)\) change order.
- Check intersection with new neighbors \((M \text{ with } O \text{ and } L \text{ with } N)\).
Event point is an intersection.

- The two intersecting segments (L and M) change order.
- Check intersection with new neighbors (M with O and L with N).
- Intersection(s) are new event points.
Event point is a right endpoint of a segment.
Event point is a right endpoint of a segment.

- The two neighbors ($O$ and $L$) become adjacent.
Event point is a right endpoint of a segment.

- The two neighbors ($O$ and $L$) become adjacent.
The two neighbors ($O$ and $L$) become adjacent.

- Check if they ($O$ and $L$) intersect.

Event point is a right endpoint of a segment.
Status Update (3)

Event point is a right endpoint of a segment.

- The two neighbors ($O$ and $L$) become adjacent.
- Check if they ($O$ and $L$) intersect.
- Intersection is new event point.
**Correctness**

*Invariant* at any time during the plane sweep:

All intersection points to the left of the sweep line have been computed correctly.

The correctness of the algorithm thus follows.
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$. 
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$.

- fetching the next event
- inserting an event
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$.

- fetching the next event
- inserting an event

Every event point $p$ is stored with all segments starting at $p$. 
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$.

- fetching the next event
- inserting an event

Every event point $p$ is stored with all segments starting at $p$.

Data structure: balanced binary search tree (e.g., red-black tree).
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$.

- fetching the next event
- inserting an event

Every event point $p$ is stored with all segments starting at $p$.

Data structure: balanced binary search tree (e.g., red-black tree).

$m = \text{#event points in the queue}$
Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$.

- fetching the next event $\mathcal{O}(\log m)$
- inserting an event $\mathcal{O}(\log m)$

Every event point $p$ is stored with all segments starting at $p$.

Data structure: balanced binary search tree (e.g., red-black tree).

$m = \# \text{event points in the queue}$
Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.
Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$. 
Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.

  - Insert($T, s$)
  - Delete($T, s$)
  - Above($T, s$) // segment immediately above $s$
  - Below($T, s$) // segment immediately below $s$
Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.
  
  Insert($T$, $s$)
  Delete($T$, $s$)
  Above($T$, $s$) // segment immediately above $s$
  Below($T$, $s$) // segment immediately below $s$

- e.g. Red-black trees, splay trees (key comparisons replaced by cross-product comparisons).
Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.

  - `Insert(T, s)`
  - `Delete(T, s)`
  - `Above(T, s)`  // segment immediately above $s$
  - `Below(T, s)`  // segment immediately below $s$

- E.g., Red-black trees, splay trees (key comparisons replaced by cross-product comparisons).

- $O(\log n)$ for each operation.
An Example
An Example

The bottom-up order of the segments correspond to the *left-to-right* order of the leaves in the tree $T$. 
An Example

- The bottom-up order of the segments correspond to the left-to-right order of the leaves in the tree $T$.
- Each internal node stores the segment from the rightmost leaf in its left subtree.
Additional Operation

Searching for the segment immediately below some point $p$ on the sweep line.
Searching for the segment immediately below some point $p$ on the sweep line.
Searching for the segment immediately below some point \( p \) on the sweep line.

- Descend binary search tree all the way down to a leaf.
Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($q$) or the leaf immediately to its left ($p$).
Searching for the segment immediately below some point \( p \) on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf \((q)\) or the leaf immediately to its left \((p)\).
Searching for the segment immediately below some point \( p \) on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf (\( q \)) or the leaf immediately to its left (\( p \)).
Additional Operation

Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($q$) or the leaf immediately to its left ($p$).
Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($q$) or the leaf immediately to its left ($p$).

$O(t)$ time
The Algorithm

FindIntersections(S)

Input: a set $S$ of line segments

Output: all intersection points and for each intersection the segment containing it.

1. $Q \leftarrow \emptyset$ // initialize an empty event queue
2. Insert the segment endpoints into $Q$ // store with every left endpoint the corresponding segments
3. $T \leftarrow \emptyset$ // initialize an empty status structure
4. while $Q \neq \emptyset$
5. do extract the next event point $p$
6. $Q \leftarrow Q - \{p\}$
7. HandleEventPoint($p$)
Handling Event Points

Status updates (1) – (3) presented earlier.
Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: several segments are involved in one event point (tricky).
Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: several segments are involved in one event point (tricky).
Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: several segments are involved in one event point (tricky).
Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: several segments are involved in one event point (tricky).

(a) Delete $D, E, A, C$
(all ending at $p$)
Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: several segments are involved in one event point (tricky).

(a) Delete $D, E, A, C$  
(all ending at $p$)

(b) Insert $B, A, C$  
(all starting at $p$)
Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: several segments are involved in one event point (tricky).

(a) Delete $D, E, A, C$ (all ending at $p$)
(b) Insert $B, A, C$ (all starting at $p$)
The Code for Event Handling

HandleEventPoint(p)

1. \( L(p) \leftarrow \{ \text{segments with left endpoint } p \} \) // they are stored with \( p \)
2. \( S(p) \leftarrow \{ \text{segments containing } p \} \) // all adjacent in \( T \) and found by a search
3. \( R(p) = \{ \text{segments with right endpoint } p \} \) // \( L(p) \subseteq S(p) \)
4. \( C(p) = \{ \text{segments with } p \text{ in interior} \} \) // \( C(p) \subseteq S(p) \)
5. if \( |L \cup R \cup C| > 1 \)
6. then report \( p \) as an intersection along with \( L, R, C \)
7. \( T \leftarrow T - (R \cup C) \)
8. \( T \leftarrow T \cup (L \cup C) \) // order as intersected by sweep line just to the right of \( p \). segments in \( C(p) \) have order reversed.
9. if \( L \cup C = \emptyset \) // right endpoint only
10. then let \( s_b \) and \( s_a \) be the neighbors right below and above \( p \) in \( T \)
11. FindNewEvent(\( s_b, s_a, p \))
12. else \( s' \leftarrow \) lowest segment in \( L \cup C \)
13. \( s_b \leftarrow \) segment right below \( s' \)
14. FindNewEvent(\( s_b, s', p \))
15. \( s'' \leftarrow \) highest segment in \( L \cup C \)
16. \( s_a \leftarrow \) segment right above \( s'' \)
17. FindNewEvent(\( s'', s_a, p \))
Finding New Event

\[ \text{FindNewEvent}(s_l, s_r, p) \]

1. if \( s_l \) and \( s_r \) intersect to the right of \( p \) // sweep line position
2. then insert the intersection point as an event in \( Q \)
The tree $T$ stores every segment once. $O(n)$
Time & Storage

The tree $T$ stores every segment once. $O(n)$
The size of the event queue $Q$: $O(n + I)$. 

Time & Storage

The tree $T$ stores every segment once. $O(n)$
The size of the event queue $Q$: $O(n + I)$.

Reduce the storage to $O(n)$. 

The tree $T$ stores every segment once. $O(n)$
The size of the event queue $Q$: $O(n + I)$.

Reduce the storage to $O(n)$.

- Store intersections among adjacent segments in the event queue.
- Delete those of segments that stop being adjacent.
- Before the deleted point is reached, the segments must have become adjacent again, resulting in the addition of the point to the even queue.
Time & Storage

The tree $T$ stores every segment once. $O(n)$
The size of the event queue $Q$: $O(n + I)$.

Reduce the storage to $O(n)$.

- Store intersections among adjacent segments in the event queue.
- Delete those of segments that stop being adjacent.
- Before the deleted point is reached, the segments must have become adjacent again, resulting in the addition of the point to the even queue.

**Theorem** All $I$ intersections of $n$ line segments in the plane can be reported in $O((n + I) \log n)$ time and $O(n)$ space.
Correctness

**Lemma** Algorithm *FindIntersections* computes all intersections and their containing segments correctly.
Correctness

**Lemma** Algorithm FindIntersections computes all intersections and their containing segments correctly.

**Proof** By induction. Let \( p \) be an intersection point and assume all intersections with a higher priority have been computed correctly.
Lemma Algorithm FindIntersections computes all intersections and their containing segments correctly.

Proof By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

$p$ is an endpoint. $\implies$ stored in the event queue $Q$.
$L(p)$ is also stored in $Q$. 
Correctness

**Lemma** Algorithm FindIntersections computes all intersections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

$p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
$L(p)$ is also stored in $Q$.
$R(p)$ and $C(p)$ are stored in $T$ and will be found.
Correctness

**Lemma** Algorithm **FindIntersections** computes all intersections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

$p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
$L(p)$ is also stored in $Q$.
$R(p)$ and $C(p)$ are stored in $T$ and will be found.

All involved are determined correctly.
Correctness

Lemma Algorithm FindIntersections computes all intersections and their containing segments correctly.

Proof By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  - $L(p)$ is also stored in $Q$.
  - $R(p)$ and $C(p)$ are stored in $T$ and will be found.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.

All involved are determined correctly.
Lemma Algorithm FindIntersections computes all intersections and their containing segments correctly.

Proof By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$. $L(p)$ is also stored in $Q$. $R(p)$ and $C(p)$ are stored in $T$ and will be found. All involved are determined correctly.
- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$. All involved segments have $p$ in interior. Order them by polar angle.
Correctness

**Lemma** Algorithm FindIntersections computes all intersections and their containing segments correctly.

**Proof** By induction. Let \( p \) be an intersection point and assume all intersections with a higher priority have been computed correctly.

\[ p \text{ is an endpoint. } \Rightarrow \text{ stored in the event queue } Q. \]
\[ L(p) \text{ is also stored in } Q. \]
\[ R(p) \text{ and } C(p) \text{ are stored in } T \text{ and will be found.} \]

\[ p \text{ is not an endpoint. We show that } p \text{ will be inserted into } Q. \]

All involved segments have \( p \) in interior. Order them by polar angle. Let \( A \) and \( B \) be neighboring segments.
Correctness

**Lemma** Algorithm FindIntersections computes all intersections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  - $L(p)$ is also stored in $Q$.
  - $R(p)$ and $C(p)$ are stored in $T$ and will be found.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.
  - All involved segments have $p$ in interior. Order them by polar angle.
  - Let $A$ and $B$ be neighboring segments.
  - There exists event point $q < p$ after which $A$ and $B$ become adjacent.
Correctness

**Lemma** Algorithm FindIntersections computes all intersections and their containing segments correctly.

**Proof** By induction. Let \( p \) be an intersection point and assume all intersections with a higher priority have been computed correctly.

- \( p \) is an endpoint. \( \Rightarrow \) stored in the event queue \( Q \).
  - \( L(p) \) is also stored in \( Q \).
  - \( R(p) \) and \( C(p) \) are stored in \( T \) and will be found.

- \( p \) is not an endpoint. We show that \( p \) will be inserted into \( Q \).
  - All involved segments have \( p \) in interior. Order them by polar angle.
  - Let \( A \) and \( B \) be neighboring segments.
    - There exists event point \( q < p \) after which \( A \) and \( B \) become adjacent.
    - By induction, \( q \) was handled correctly and \( p \) is detected and stored in \( Q \).
Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.
Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time $O((n + k) \log n)$
Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time \( O((n + k) \log n) \)

output size
(intersections and their containing segments)
Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time \( O((n + k) \log n) \)

output size

(output size (intersections and their containing segments))

But one intersection may consist of \( \Theta(n) \) segments.
Output Sensitivity

An algorithm is output sensitive if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

\[
\text{running time } \mathcal{O}((n + k) \log n)
\]

But one intersection may consist of \(\Theta(n)\) segments.

When this happens, running time becomes \(\mathcal{O}(n^2 \log n)\)
Output Sensitivity

An algorithm is **output sensitive** if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

\[
\text{running time } O((n + k) \log n)
\]

\[
\text{output size (intersections and their containing segments)}
\]

But one intersection may consist of \(\Theta(n)\) segments.

When this happens, running time becomes \(O(n^2 \log n)\)

Not tight! – the total number of intersections may still be \(\Theta(n)\).
A Tighter Bound

The running time is $O(n \log n + I \log n)$. 
A Tighter Bound

The running time is $O(n \log n + I \log n)$.

# intersections
A Tighter Bound

The running time is $O(n \log n + I \log n)$.

Idea in analysis  all time cost is spent on maintaining the two data structures:
1) the event queue $Q$, and 2) the status structure tree $T$. 

# intersections
A Tighter Bound

The running time is $O(n \log n + I \log n)$.

# intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures: 1) the event queue $Q$, and 2) the status structure tree $T$.

**FindIntersections**(S)

1. $Q \leftarrow \emptyset$ // initialize an empty event queue
2. Insert the segment endpoints into $Q$ // balanced binary search tree $O(n \log n)$
3. $T \leftarrow \emptyset$ // initialize an empty status structure
4. while $Q \neq \emptyset$
5. do extract the next event point $p$
6. $Q \leftarrow Q - \{p\}$ // deletion from $Q$ takes time $O(\log n)$.
7. HandleEventPoint($p$) // 1 or 2 calls to FindNewEvent
A Tighter Bound

The running time is $O(n \log n + I \log n)$.

Idea in analysis  all time cost is spent on maintaining the two data structures:
1) the event queue $Q$, and 2) the status structure tree $T$.

FindIntersections($S$)
1. $Q \leftarrow \emptyset$  // initialize an empty event queue
2. Insert the segment endpoints into $Q$  // balanced binary search tree $O(n \log n)$
3. $T \leftarrow \emptyset$  // initialize an empty status structure
4. while $Q \neq \emptyset$
5. do extract the next event point $p$
6. $Q \leftarrow Q - \{p\}$  // deletion from $Q$ takes time $O(\log n)$.
7. HandleEventPoint($p$)  // 1 or 2 calls to FindNewEvent
     // each call may result in an insertion into $Q$: $O(\log n)$.
A Tighter Bound

The running time is \( O(n \log n + I \log n) \).

\[\uparrow\]

# intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures:
1) the event queue \( Q \), and 2) the status structure tree \( T \).

\textbf{FindIntersections}(S)
1. \( Q \leftarrow \emptyset \) // initialize an empty event queue
2. Insert the segment endpoints into \( Q \) // balanced binary search tree \( O(n \log n) \)
3. \( T \leftarrow \emptyset \) // initialize an empty status structure
4. while \( Q \neq \emptyset \)
5. do extract the next event point \( p \)
6. \( Q \leftarrow Q - \{p\} \) // deletion from \( Q \) takes time \( O(\log n) \).
7. \( \text{HandleEventPoint}(p) \) // 1 or 2 calls to \( \text{FindNewEvent} \)
   // each call may result in an insertion into \( Q \): \( O(\log n) \).
   // each operation on \( T \) – insertion, deletion, neighbor
   // finding – takes time \( O(\log n) \).
A Tighter Bound

The running time is $O(n \log n + I \log n)$.

# intersections

Idea in analysis all time cost is spent on maintaining the two data structures:
1) the event queue $Q$, and 2) the status structure tree $T$.

FindIntersections($S$)

1. $Q \leftarrow \emptyset$ // initialize an empty event queue
2. Insert the segment endpoints into $Q$ // balanced binary search tree $O(n \log n)$
3. $T \leftarrow \emptyset$ // initialize an empty status structure
4. while $Q \neq \emptyset$
5. do extract the next event point $p$
6. $Q \leftarrow Q - \{p\}$ // deletion from $Q$ takes time $O(\log n)$.
7. HandleEventPoint($p$) // 1 or 2 calls to FindNewEvent
   // each call may result in an insertion into $Q$: $O(\log n)$.
   // each operation on $T$ - insertion, deletion, neighbor
   // finding - takes time $O(\log n)$ .
   // #operations on $T$ is $\Theta(|L(p) \cup R(p) \cup C(p)|)$
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).
Total Number of Tree Operations

Let  \( m = \sum_{p} |L(p) \cup R(p) \cup C(p)| \)

Then the running time is  \( O((m + n) \log n) \).

Claim  \( m = O(n + I) \).
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

Claim \( m = O(n + I) \).

Proof View the set of segments as a planar graph.
Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

**Claim** \( m = O(n + I) \).

**Proof** View the set of segments as a planar graph.

\[ |L(p) \cup R(p) \cup C(p)| \leq \deg(p) \]
Total Number of Tree Operations

Let \( m = \sum_{p} |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

**Claim** \( m = O(n + I) \).

**Proof** View the set of segments as a planar graph.

\[ |L(p) \cup R(p) \cup C(p)| \leq \deg(p) \]

Each may contribute 2 to the degree.
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

Claim \( m = O(n + I) \).

Proof View the set of segments as a planar graph.

\[
|L(p) \cup R(p) \cup C(p)| \leq \deg(p)
\]

\[
\Rightarrow m \leq \sum_p \deg(p)
\]

Each may contribute 2 to the degree
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

Claim \( m = O(n + I) \).

Proof View the set of segments as a planar graph.

Each may contribute 2 to the degree

\[ |L(p) \cup R(p) \cup C(p)| \leq \text{deg}(p) \]

\[ m \leq \sum_p \text{deg}(p) \]
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

**Claim** \( m = O(n + I) \).

**Proof** View the set of segments as a planar graph.

Each may contribute 2 to the degree

\[ |L(p) \cup R(p) \cup C(p)| \leq \text{deg}(p) \]

\[ m \leq \sum_p \text{deg}(p) \]

\( n_e = \#\text{edges} \quad n_v = \#\text{vertices} \)
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

**Claim** \( m = O(n + I) \).

**Proof** View the set of segments as a planar graph.

\[ |L(p) \cup R(p) \cup C(p)| \leq \deg(p) \]

\[ m \leq \sum_p \deg(p) \]

\( n_e = \# \text{edges} \quad n_v = \# \text{vertices} \)

Each may contribute 2 to the degree

Every edge contributes one to the degree of each of its two vertices.
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

**Claim** \( m = O(n + I) \).

**Proof** View the set of segments as a planar graph.

Each may contribute 2 to the degree

\(| L(p) \cup R(p) \cup C(p)| \leq \deg(p) \)

\[ m \leq \sum_p \deg(p) \]

\( n_e = \text{#edges} \quad n_v = \text{#vertices} \)

Every edge contributes one to the degree of each of its two vertices.

\[ \sum_p \deg(p) = 2n_e \]
Total Number of Tree Operations

Let \( m = \sum_p |L(p) \cup R(p) \cup C(p)| \)

Then the running time is \( O((m + n) \log n) \).

Claim \( m = O(n + I) \).

Proof View the set of segments as a planar graph.

\[ |L(p) \cup R(p) \cup C(p)| \leq \deg(p) \]
\[ \Rightarrow m \leq \sum_p \deg(p) \]

\( n_e = \#\text{edges} \quad n_v = \#\text{vertices} \)

Every edge contributes one to the degree of each of its two vertices.

\[ \sum_p \deg(p) = 2n_e \]
\[ \Rightarrow m \leq 2n_e \]
Meanwhile, a region is bounded by at least three edges.

\[ 3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e \]
Meanwhile, a region is bounded by at least three edges.

\[ 3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3} n_e \]

\[ n_v \leq 2n + I \] (some endpoints may be also intersections)
Meanwhile, a region is bounded by at least three edges.

\[ 3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e \]

\[ n_v \leq 2n + I \] (some endpoints may be also intersections)

Euler’s equation:  

\[ n_v - n_e + n_f = k + 1 \geq 2 \]  

# connected components
Meanwhile, a region is bounded by at least three edges.

\[ 3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e \]

\[ n_v \leq 2n + I \] (some endpoints may be also intersections)

Euler’s equation:

\[ n_v - n_e + n_f = k + 1 \geq 2 \]

\[ n_f \leq \frac{2}{3}n_e \]

\[ n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2 \]
Meanwhile, a region is bounded by at least three edges.

\[ 3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3} n_e \]

\[ n_v \leq 2n + I \quad \text{(some endpoints may be also intersections)} \]

**Euler’s equation:**  
- \# connected components

\[ n_v - n_e + n_f = k + 1 \geq 2 \]

\[ n_f \leq \frac{2}{3} n_e \]

\[ n_v - n_e + \frac{2}{3} n_e \geq n_v - n_e + n_f \geq 2 \]

\[ n_v \leq 2n + I \]

\[ n_e \leq 3n_v - 6 \leq 6n + 3I - 6 \]
Meanwhile, a region is bounded by at least three edges.

\[ 3n_f \leq 2n_e \implies n_f \leq \frac{2}{3}n_e \]

\[ n_v \leq 2n + I \quad \text{(some endpoints may be also intersections)} \]

Euler’s equation:

\[ n_v - n_e + n_f = k + 1 \geq 2 \]

\[ n_f \leq \frac{2}{3}n_e \]

\[ n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2 \]

\[ n_v \leq 2n + I \]

\[ n_e \leq 3n_v - 6 \leq 6n + 3I - 6 \]

\[ m \leq 2n_e \leq 12n + 6I - 12 = O(n + I) \]
Meanwhile, a region is bounded by at least three edges.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e$$

$$n_v \leq 2n + I \text{ (some endpoints may be also intersections)}$$

Euler’s equation:

$$n_v - n_e + n_f = k + 1 \geq 2$$

$$n_f \leq \frac{2}{3}n_e$$

$$n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2$$

$$n_v \leq 2n + I$$

$$n_e \leq 3n_v - 6 \leq 6n + 3I - 6$$

$$m \leq 2n_e \leq 12n + 6I - 12 = O(n + I)$$