Monotone Polygons: A polygonal chain $C$ is said to be strictly monotone with respect to a given line $L$, if any line that is orthogonal to $L$ intersects $C$ in at most one point. A chain $C$ is monotone with respect to $L$ if each line that is orthogonal to $L$ intersects $C$ in a single connected component. Thus it may intersect, not at all, at a single point, or along a single line segment. A polygon $P$ is said to be monotone with respect to a line $L$ if its boundary, $\partial P$, can be split into two chains, each of which is monotone with respect to $L$.

Henceforth, let us consider monotonicity with respect to the $x$-axis. We will call these polygons horizontally monotone. It is easy to test whether a polygon is horizontally monotone. How?
(a) Find the leftmost and rightmost vertices (min and max x-coordinate) in $O(n)$ time.

(b) These vertices split the polygon’s boundary into two chains, an upper chain and a lower chain. Walk from left to right along each chain, verifying that the x-coordinates are nondecreasing. This takes $O(n)$ time.

As a challenge, consider the problem of determining whether a polygon is monotone in any (unspecified) direction. This can be done in $O(n)$ time, but is quite a bit harder.

**Triangulation of Monotone Polygons:** We can triangulate a monotone polygon by a simple variation of the plane-sweep method. We begin with the assumption that the vertices of the polygon have been sorted in increasing order of their x-coordinates. (For simplicity we assume no duplicate x-coordinates. Otherwise, break ties between the upper and lower chains arbitrarily, and within a chain break ties so that the chain order is preserved.) Observe that this does not require sorting. We can simply extract the upper and lower chain, and merge them (as done in mergesort) in $O(n)$ time.

The idea behind the triangulation algorithm is quite simple: Try to triangulate everything you can to the left of the current vertex by adding diagonals, and then remove the triangulated region from further consideration.

In the example, there is obviously nothing to do until we have at least 3 vertices. With vertex 3, it is possible to add the diagonal to vertex 2, and so we do this. In adding vertex 4, we can add the diagonal to vertex 2.
However, vertices 5 and 6 are not visible to any other nonadjacent vertices so no new diagonals can be added. When we get to vertex 7, it can be connected to 4, 5, and 6. The process continues until reaching the final vertex.

The important thing that makes the algorithm efficient is the fact that when we arrive at a vertex the untriangulated region that lies to the left of this vertex always has a very simple structure. This structure allows us to determine in constant time whether it is possible to add another diagonal. And in general we can add each additional diagonal in constant time. Since any triangulation consists of \( n - 3 \) diagonals, the process runs in \( O(n) \) total time. This structure is described in the lemma below.

**Lemma: (Main Invariant)** For \( i \geq 2 \), let \( v_i \) be the vertex just processed by the triangulation algorithm. The untriangulated region lying to the left of \( v_i \) consists of two \( x \)-monotone chains, a lower chain and an upper chain each containing at least one edge. If the chain from \( v_i \) to \( u \) has two or more edges, then these edges form a reflex chain (that is, a sequence of vertices with interior angles all at least 180 degrees). The other chain consists of a single edge whose left endpoint is \( u \) and whose right endpoint lies to the right of \( v_i \).

We will prove the invariant by induction. As the basis case, consider the case of \( v_2 \). Here \( u = v_1 \), and one chain consists of the single edge \( v_2 v_1 \) and the other chain consists of the other edge adjacent to \( v_1 \).

To prove the main invariant, we will give a case analysis of how to handle the next event, involving \( v_i \), assuming that the invariant holds at \( v_{i-1} \) and see that the invariant is satisfied after each event has been processed. There are the following cases that the algorithm needs to deal with.

**Case 1:** \( v_i \) lies on the opposite chain from \( v_{i-1} \):

In this case we add diagonals joining \( v_i \) to all the vertices on the reflex chain, from \( v_{i-1} \) back to (but not including) \( u \). Note that all of these vertices are visible from \( v_i \). Certainly \( u \) is visible to \( v_i \). Because the chain is reflex, \( x \)-monotone, and lies to the left of \( v_i \) it follows that the chain itself cannot block the visibility from \( v_i \) to some other vertex on the chain. Finally, the fact that the polygon is \( x \)-monotone implies that the unprocessed portion of the polygon (lying to the right of \( v_i \)) cannot “sneak back” and block visibility to the chain.

After doing this, we set \( u = v_{i-1} \). The invariant holds, and the reflex chain is trivial, consisting of the single edge \( v_i v_{i-1} \).

![Initial invariant](image)

![Case 1](image)

![Case 2a](image)

![Case 2b](image)

**Figure 18: Triangulation cases.**

**Case 2:** \( v \) is on the same chain as \( v_{i-1} \):

We walk back along the reflex chain adding diagonals joining \( v_i \) to prior vertices until we find the first that is not visible to \( v_i \). As can be seen in the figure, this may involve connecting \( v_i \) to one or more vertices (2a) or it may involve connecting \( v_i \) to no additional vertices (2b), depending on whether the first angle is less or greater than 180 degrees. In either case the vertices that were cut off by diagonals are no longer
in the chain, and \( v_i \) becomes the new endpoint to the chain. Again, by \( x \)-monotonicity it follows that the unprocessed portion of the polygon cannot block visibility of \( v_i \) to the chain.

Note that when we are done the remaining chain from \( v_i \) to \( u \) is a reflex chain. (Note the similarity between this step and the main iteration in Graham’s scan.)

How is this implemented? The vertices on the reflex chain can be stored in a stack. We keep a flag indicating whether the stack is on the upper chain or lower chain, and assume that with each new vertex we know which chain of the polygon it is on. Note that decisions about visibility can be based simply on orientation tests involving \( v_i \) and the top two entries on the stack. When we connect \( v_i \) by a diagonal, we just pop the stack.

**Analysis:** We claim that this algorithm runs in \( O(n) \) time. As we mentioned earlier, the sorted list of vertices can be constructed in \( O(n) \) time through merging. The reflex chain is stored on a stack. In \( O(1) \) time per diagonal, we can perform an orientation test to determine whether to add the diagonal and (assuming a DCEL) the diagonal can be added in constant time. Since the number of diagonals is \( n - 3 \), the total time is \( O(n) \).

**Monotone Subdivision:** In order to run the above triangulation algorithm, we first need to subdivide an arbitrary simple polygon \( P \) into monotone polygons. This is also done by a plane-sweep approach. We will add a set of nonintersecting diagonals that partition the polygon into monotone pieces.

Observe that the absence of \( x \)-monotonicity occurs only at vertices in which the interior angle is greater than 180 degrees and both edges lie either to the left of the vertex or both to the right. Following our book’s notation, we call the first type a *merge vertex* (since as the sweep passes over this vertex the edges seem to be merging) and the latter type a *split vertex*.

Let’s discuss the case of a split vertex first (both edges lie to the right of the vertex). When a split vertex \( v \) is encountered in the sweep, there will be an edge \( e_a \) of the polygon lying above and an edge \( e_b \) lying below. We might consider attaching the split vertex to left endpoint of one of these two edges, but it might be that neither endpoint is visible to the split vertex. We need to maintain a vertex that is visible to any split vertex that may arise between \( e_a \) and \( e_b \). To do this, imagine a fluorescent light shining down from every point on \( e_a \) (these are the white vertices in the figure below, left). Note that \( e_a \) left endpoint is considered in this set as well. We say that these vertices are *vertically visible below* \( e_a \). Among all the vertices that are illuminated by these vertical light rays, let \( u \) be the rightmost. We claim that \( u \) is visible to every point along the sweep line between \( e_a \) and \( e_b \). This suggests the following concept, which is defined for each edge \( e_a \) that intersects the sweep line, such that the polygon’s interior lies locally below \( e_a \).

![Figure 19: Split vertices, merge vertices, and helpers.](image)

**helper**\((e_a)\) : Let \( e_b \) be the edge of the polygon lying just below \( e_a \) on the sweep line. The helper is the rightmost vertically visible vertex below \( e_a \) on the polygonal chain between \( e_a \) and \( e_b \).

We join each split vertex to **helper**\((e_a)\), where \( e_a \) is the edge of \( P \) immediately above the split vertex. Note that it is possible that the helper is the left endpoint of \( e_a \). Also note that **helper**\((e_a)\) is defined with respect to the
current location of the sweep line. As the sweep line moves, its value changes. Also, it is only defined for those edges intersected by the sweep line.

**Events:** The endpoints of the edges of the polygon. These are sorted by increasing order of $x$-coordinates. Since no new events are generated, the events may be stored in a simple sorted list (i.e., no priority queue is needed).

**Sweep status:** The sweep line status consists of the list of edges that intersect the sweep line, sorted from top to bottom. Our book notes that we actually only need to store edges such that the polygon lies just below this edge (since these are the only edges that we evaluate helper() from). These edges are stored in a dictionary (e.g., a balanced binary tree or a skip list), so that the operations of insert, delete, find, predecessor and successor can be evaluated in $O(\log n)$ time each.

**Event processing:** There are six event types based on a case analysis of the local structure of edges around each vertex. Let $v$ be the current vertex encountered by the sweep.

- **Split vertex:** Search the sweep line status to find the edge $e$ lying immediately above $v$. Add a diagonal connecting $v$ to helper$(e)$. Add the two edges incident to $v$ in the sweep line status, and make $v$ the helper of the lower of these two edges and make $v$ the new helper of $e$.

- **Merge vertex:** Find the two edges incident to this vertex in the sweep line status (they must be adjacent). Delete them both. Let $e$ be the edge lying immediately above them. Make $v$ the new helper of $e$.

- **Start vertex:** (Both edges lie to the right of $v$, but the interior angle is less than 180 degrees.) Insert this vertex’s edges into the sweep line status. Set the helper of the upper edge to $v$.

- **End vertex:** (Both edges lie to the left of $v$, but the interior angle is less than 180 degrees.) Delete both edges from the sweep line status.

- **Upper-chain vertex:** (One edge is to the left, and one to the right, and the polygon interior is below.) Replace the left edge with the right edge in the sweep line status. Make $v$ the helper of the new edge.

- **Lower-chain vertex:** (One edge is to the left, and one to the right, and the polygon interior is above.) Replace the left edge with the right edge in the sweep line status. Let $e$ be the edge lying above here. Make $v$ the helper of $e$.

![Figure 20: Plane sweep cases.](image)

This only inserts diagonals to fix the split vertices. What about the merge vertices? This could be handled by applying essentially the same algorithm using a reverse (right to left) sweep. It can be shown that this will never introduce crossing diagonals, but it might attempt to insert the same diagonal twice. However, the book suggests a simpler approach. Whenever we change a helper vertex, check whether the original helper vertex is a merge vertex. If so, the new helper vertex is then connected to the merge vertex by a new diagonal. It is not hard to show that this essentially has the same effect as a reverse sweep, and it is easier to detect the possibility of a duplicate insertion (in case the new vertex happens to be a split vertex).

There are many special cases (what a pain!), but each one is fairly easy to deal with, so the algorithm is quite efficient. As with previous plane sweep algorithms, it is not hard to show that the running time is $O(\log n)$ times the number of events. In this case there is one event per vertex, so the total time is $O(n \log n)$. This gives us an $O(n \log n)$ algorithm for polygon triangulation.