Polygon Triangulation

Partha Bhowmick
https://cse.iitkgp.ac.in/~pb

1 Introduction

Triangulation is the process of subdividing a polygon P into triangles by adding nonintersecting diagonals. A **diagonal** is a line segment that connects two of its vertices, such that all its interior points lie strictly in the interior of P. The term **triangulation** also refers to the set of triangles resulting from triangulating a polygon, with the meaning being clear from the context.

We assume that P is a simple polygon, meaning its edges intersect only at their endpoints. Additionally, P has no holes and is given as an ordered sequence of vertices, either clockwise or counterclockwise, starting from an arbitrary vertex. For clarity, we will often assume that all vertices have distinct x-coordinates and/or distinct y-coordinates. However, the theoretical analysis and algorithm can be adapted to handle cases where this assumption does not hold.

Triangulation is not necessarily unique, and additional criteria may be applied for optimization. Triangulating polygons is crucial in GPU computing, shape analysis, and various algorithms that operate on polygonal structures. Figure 8 illustrates three different triangulations of a polygon, with the first being GPU-friendly, as the triangles have similar shape and size—ensuring comparable processing time, for example, in rasterization within a graphics pipeline.

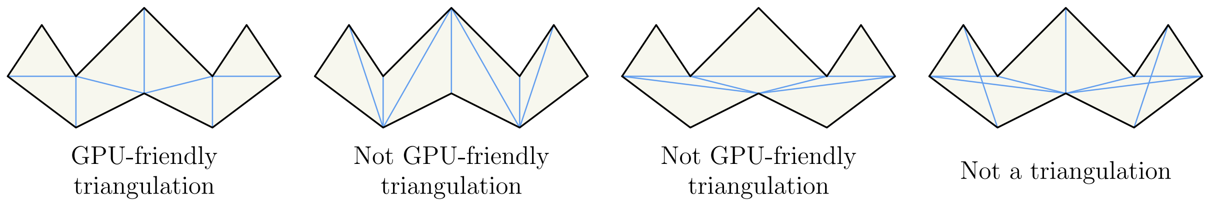

GPU-friendly triangulation
Not GPU-friendly triangulation
Not GPU-friendly triangulation
Not a triangulation

Figure 1: Different triangulations of a polygon, excepting the last. In the last example, diagonals intersect at non-vertex points, and hence do not result in a valid triangulation.

2 Art Gallery Problem

The **Art Gallery Problem** goes as follows: Given an art gallery represented as a simple polygon P with n vertices, how many cameras are needed to fully guard it? Interestingly, its answer is connected to triangulation of P, as we see shortly.

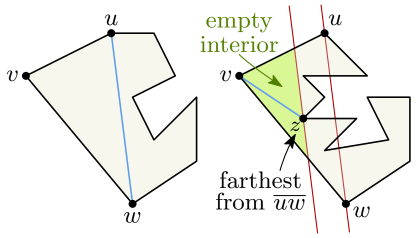
Theorem 1. *Every polygon admits a triangulation, and any such triangulation has exactly $n-2$ triangles.*

Proof. We prove this by induction on n.

Basis: For $n = 3$, the polygon itself is a triangle, and the claim is trivially true.

Hypothesis: Assume the theorem holds for all polygons with fewer than n vertices.

Induction step: We first establish the existence of a diagonal. Let P be an n-vertex polygon. Let v be a/the leftmost vertex of P, with u and w as its adjacent vertices. If the segment $\overline{uw}$, excluding its endpoints, lies inside P, then $\overline{uw}$ is a diagonal. Otherwise, some vertices lie inside $\triangle uvw$ or on $\overline{uw}$. Among these, let z be a/the farthest from $\overleftrightarrow{uw}$. Thus, $\overline{vz}$ is a diagonal, as it does not intersect any edge of P due to our choice of z.



Since a diagonal always exists, it divides P into two smaller polygons P_1 and P_2 with n_1 and n_2 vertices, respectively. Both n_1 and n_2 are less than n, so by the induction hypothesis, each of these polygons can be triangulated. Thus, P itself can be triangulated.

To show that any triangulation of P consists of exactly $n-2$ triangles, consider an arbitrary triangulation $T(P)$. Any diagonal in $T(P)$ splits P into two sub-polygons with n_1 and n_2 vertices, respectively. Since each vertex belongs uniquely to one of the two sub-polygons, except for the endpoints of the diagonal, we have

$$n_1 + n_2 = n + 2.$$

Applying the induction hypothesis, any triangulation of P_i consists of $n_i - 2$ triangles. Therefore, the total number of triangles in $T(P)$ is

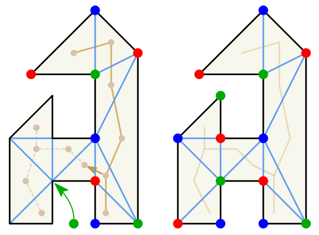
$$(n_1 - 2) + (n_2 - 2) = n - 2.$$ □

By Theorem 1, any polygon with n vertices can be triangulated. A naive approach places a camera inside every triangle, requiring $n-2$ cameras—an overkill. A camera positioned on a diagonal can cover two triangles, potentially reducing the count to about $n/2$. Placing cameras at vertices is even more efficient, as a single vertex may be incident to multiple triangles. Theorem 2, stated shortly, provides the tightest upper bound. To prove it, we need the following lemma.

Lemma 1. *For any triangulation $T(P)$, the vertices of P can be colored with 3 colors such that every two vertices connected by an edge or diagonal in $T(P)$ have different colors.*

Proof. Consider the dual graph $G(T(P))$ of $T(P)$. Each node i in $G(T(P))$ corresponds to a triangle t_i in $T(P)$, and an edge exists between two nodes i and j if t_i and t_j share a diagonal in $T(P)$.
Since each diagonal divides P into two, removing an edge from $G(T(P))$ disconnects the graph, meaning $G(T(P))$ is a tree. A 3-coloring can be done via depth-first search (DFS) on $G(T(P))$:

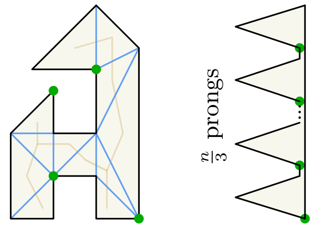
1. Start at node 1 (or any node) of $G(T(P))$, assigning 3 colors to the three vertices of t_1.
2. Upon visiting a new node j from a previously colored node i, observe that t_i shares a diagonal with t_j. Since two vertices of t_j are already colored, its third vertex is assigned the available color.

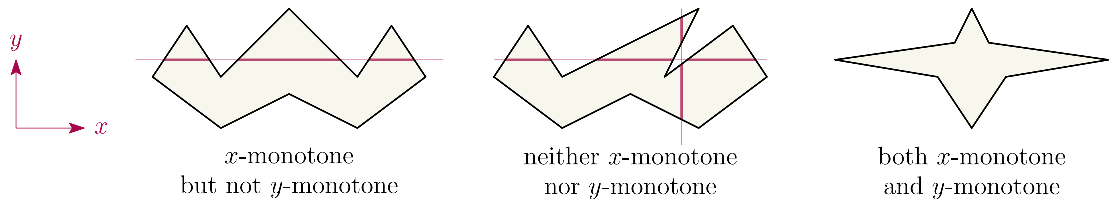


Since $G(T(P))$ is a tree, its every node is accessed exactly once, ensuring no conflict and a valid 3-coloring thereof. □

Theorem 2 (Art Gallery Theorem)**.** *$\lfloor n/3 \rfloor$ cameras are occasionally necessary and always sufficient to ensure that every point in the polygon is visible from at least one camera.*

Proof. Consider any triangulation $T(P)$ of P, and do a proper 3-coloring of $T(P)$. Since each triangle contains exactly one vertex of each color, placing cameras at all vertices of the *least-used color* guarantees full coverage, and its count will be at most $\lfloor n/3 \rfloor$.
To show that $\lfloor n/3 \rfloor$ cameras are sometimes necessary, consider a comb-shaped polygon with $n/3$ prongs, such that no single camera can cover two prongs simultaneously. Consequently, $\lfloor n/3 \rfloor$ cameras are required. □





y
x
x-monotone but not y-monotone
neither x-monotone nor y-monotone
both x-monotone and y-monotone

Figure 2: Monotone versus non-monotone polygons.

3 Basic Properties and Naive Algorithm

Suppose that P has n vertices. Then, by induction on n, it's easy to show that the number of diagonals in any triangulation of P is $n-3$, and the number of triangles is $n-2$. As discussed in the class, any polygon has always an **ear** (three consecutive vertices u, v, w such that $\overline{uw}$ is a diagonal), and by **ear decomposition**, it is easy to triangulate but the algorithm is not efficient due to the complexity of the visibility test. An optimal $O(n)$ algorithm was found by Bernard Chazelle in 1991, but it is quite complex. We'll discuss here an $O(n \log n)$ algorithm, which is preferable in practice. It is based on:

1. Decomposing the simple polygon into **monotone polygons** in $O(n \log n)$ time.
2. Triangulating each monotone polygon separately in $O(n)$ time.

The triangulation results in a planar subdivision, which can be stored in a **doubly connected edge list** (DCEL)—a linked structure representing the vertices, edges, and triangular faces, and their interrelation in a precise form. Notably, the dual graph of a triangulation forms a tree.

4 Monotone Decomposition of Polygons

Let P be a polygon with p_L and p_R as its respective leftmost and rightmost vertices. If, when traversing the upper chain of P from p_L to p_R, the vertices appear in left-to-right order, and similarly for the lower chain, then P is called ***x*-monotone**. The notion of ***y*-monotone** with respect to the y-axis—or, in general, of **ℓ-monotone** with respect to any line ℓ—is defined analogously. It is easy to prove that if P is ℓ-monotone for any line ℓ, then any line orthogonal to ℓ intersects P in at most one connected component, as shown in Figure 2.

In our discussion, we focus on x-monotone polygons. We first establish certain facts that aid in designing the algorithm. We denote by θ_v the internal angle at v, and by u and w the adjacent vertices of v. The angle θ_v is said to be **reflex** if its value is greater than 180°. We use a 3-tuple as the label for each vertex v, denoted by $\mathbb{Z}_v$, in which the first two elements are labels of u and w, while the third one indicates whether θ_v is reflex (`X`) or non-reflex (`N`). The positions of u and w relative to v are denoted by `L` (left) or `R` (right). We have 6 possible labels resulting in 8 possible cases, as listed in Figure 3.

The labels serve as identifiers for naming the vertices, as shown in Figure 3. For example, vertices v with $\mathbb{Z}_v =$ `LLX` are called **merge vertices**, and those with $\mathbb{Z}_v =$ `RRX` are referred to as **split vertices**. An example of vertex classification for a polygon is given in Figure 4.

The subdivision uses a plane-sweep approach in which the sweep-line, denoted by λ, is vertical and sweeps from left to right. It halts only at event points and performs the necessary processing. The **event points** consist solely of vertices. So, the endpoints of the edges of the polygon are sorted by increasing order of x-coordinates and the events may be stored in a simple sorted list (i.e., no priority queue is needed). The algorithm follows a greedy strategy to add a minimal (but not necessarily minimum) set of nonintersecting diagonals to merge and split vertices, as discussed next.

Observation 1. *P is **x-monotone** if and only if no vertex is a merge or split vertex.*

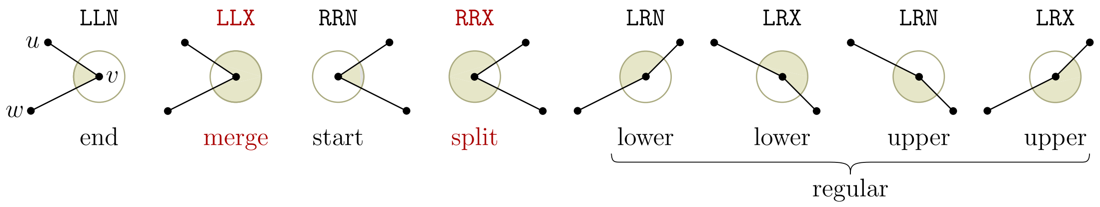


Figure 3: Vertex classification via a 3-tuple representation. For an ε-disc centered at vertex v, assuming ε is close to 0, the shaded region marks the polygon's interior, revealing whether the internal angle at v is reflex.

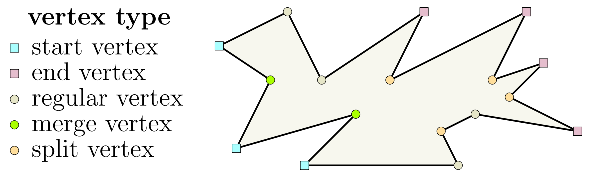


Figure 4: Different types of vertices in a non-monotone polygon.

Observation 2. *Here are the necessary and sufficient conditions for adding diagonals:*

1. *At every merge vertex v, a diagonal should be added with its other endpoint lying to the right of v.*
2. *At every split vertex, a diagonal should be added with its other endpoint lying to the left of v.*
3. *For every diagonal, at least one endpoint should be a merge/split vertex.*

We maintain a dynamic data structure T_λ for λ. It stores the IDs of the edges intersected by λ. In order that these edges are stored in top-to-bottom order, T_λ is taken as a height-balanced BST (e.g., AVL tree). As λ moves from one vertex to the next, T_λ gets updated. The notion of visibility is useful here as we see in the following observation. Note that two points in P are visible from each other if the line segment joining them lies in P.

Observation 3. *Upon reaching v, suppose e and e' are two consecutive edges intersected by λ such that e lies above e', and the points $e \cap \lambda$ and $e' \cap \lambda$ are visible from v. At this juncture, consider all vertices to the left of λ that lie between e and e' and are vertically visible from e. The rightmost vertex among them, referred to as the **helper of** e and denoted by $h(e)$, is always visible from v.*

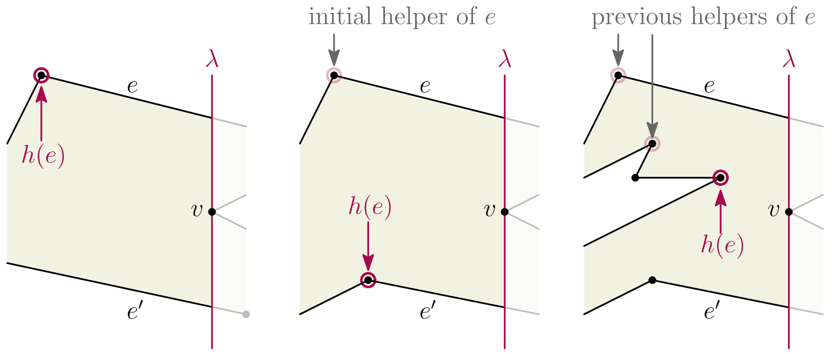
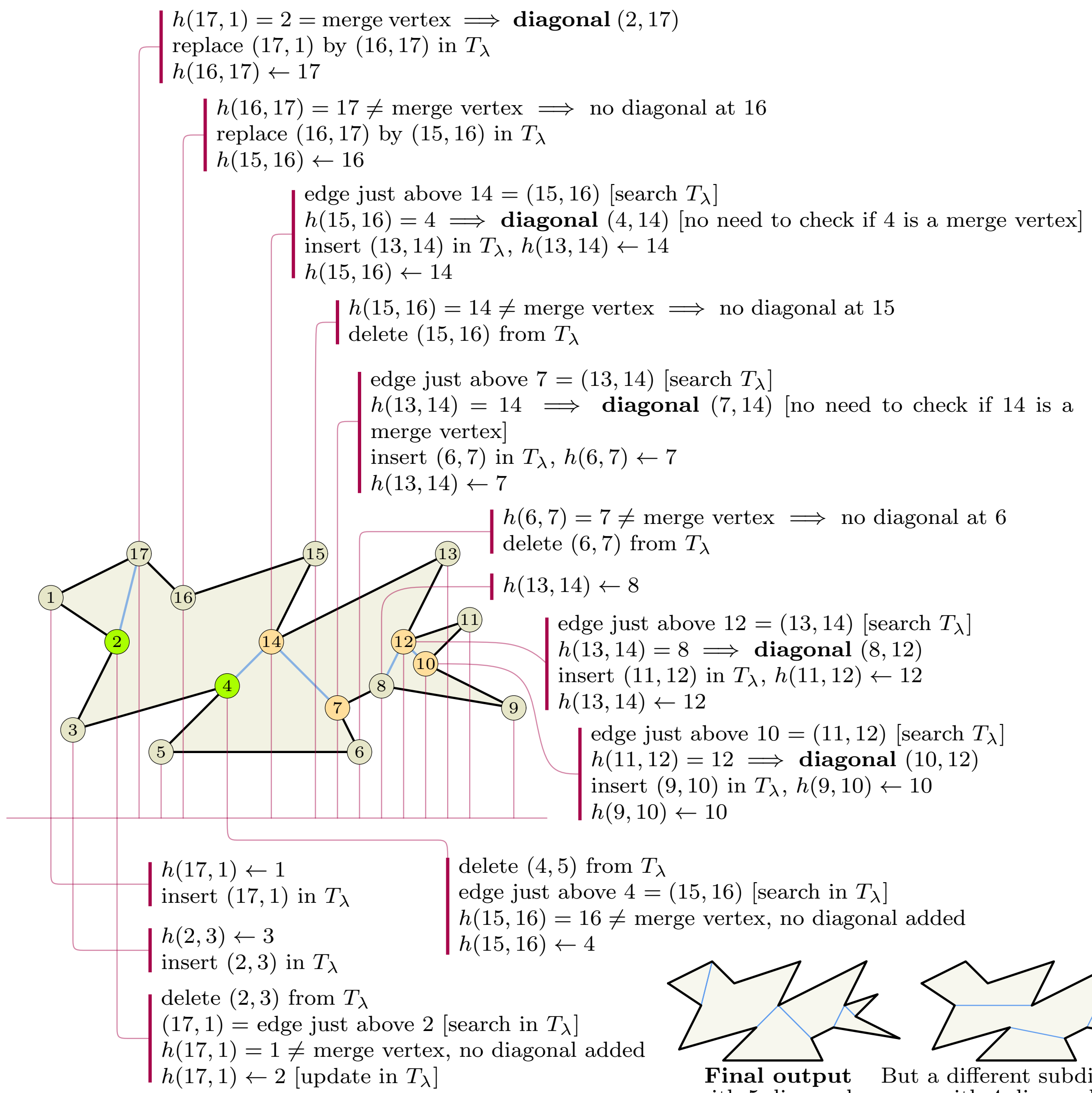


Figure 5: Three possible cases showing the helper of e when λ reaches a split vertex v. In all these cases, $h(e)$ is visible from v and a diagonal is added from v to $h(e)$. In the 3rd case, coincidentally $h(e)$ is a merge vertex, so a single diagonal fixes the diagonal requirement for both $h(e)$ and v.

The basic operations on T_λ include inserting a new edge, deleting an existing edge, searching an edge lying just above the newly encountered vertex, and updating the auxiliary information (helper) of an edge. A diagonal is a added to a split vertex v as soon as we reach it, because the other endpoint of the diagonal, i.e., $h(e)$, lies to the left of v, as shown in Figure 5. For every other combination, the decision of adding diagonals is listed in Figure 6. A diagonal is a added to a merge vertex when λ encounters the next visible vertex v to its right, since at that point the helper of e is that merge vertex. A detailed demonstration of the algorithm is given in Figure 6.



$h(17,1) = 2 =$ merge vertex $\Longrightarrow$ **diagonal** $(2,17)$
replace $(17,1)$ by $(16,17)$ in T_λ
$h(16,17) \leftarrow 17$

$h(16,17) = 17 \neq$ merge vertex $\Longrightarrow$ no diagonal at 16
replace $(16,17)$ by $(15,16)$ in T_λ
$h(15,16) \leftarrow 16$

edge just above $14 = (15,16)$ [search T_λ]
$h(15,16) = 4 \Longrightarrow$ **diagonal** $(4,14)$ [no need to check if 4 is a merge vertex]
insert $(13,14)$ in T_λ, $h(13,14) \leftarrow 14$
$h(15,16) \leftarrow 14$

$h(15,16) = 14 \neq$ merge vertex $\Longrightarrow$ no diagonal at 15
delete $(15,16)$ from T_λ

edge just above $7 = (13,14)$ [search T_λ]
$h(13,14) = 14 \Longrightarrow$ **diagonal** $(7,14)$ [no need to check if 14 is a merge vertex]
insert $(6,7)$ in T_λ, $h(6,7) \leftarrow 7$
$h(13,14) \leftarrow 7$

$h(6,7) = 7 \neq$ merge vertex $\Longrightarrow$ no diagonal at 6
delete $(6,7)$ from T_λ

$h(13,14) \leftarrow 8$

edge just above $12 = (13,14)$ [search T_λ]
$h(13,14) = 8 \Longrightarrow$ **diagonal** $(8,12)$
insert $(11,12)$ in T_λ, $h(11,12) \leftarrow 12$
$h(13,14) \leftarrow 12$

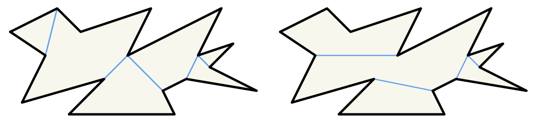
edge just above $10 = (11,12)$ [search T_λ]
$h(11,12) = 12 \Longrightarrow$ **diagonal** $(10,12)$
insert $(9,10)$ in T_λ, $h(9,10) \leftarrow 10$
$h(9,10) \leftarrow 10$

$h(17,1) \leftarrow 1$
insert $(17,1)$ in T_λ

$h(2,3) \leftarrow 3$
insert $(2,3)$ in T_λ

delete $(2,3)$ from T_λ
$(17,1) =$ edge just above 2 [search in T_λ]
$h(17,1) = 1 \neq$ merge vertex, no diagonal added
$h(17,1) \leftarrow 2$ [update in T_λ]

delete $(4,5)$ from T_λ
edge just above $4 = (15,16)$ [search in T_λ]
$h(15,16) = 16 \neq$ merge vertex, no diagonal added
$h(15,16) \leftarrow 4$



Final output
with 5 diagonals

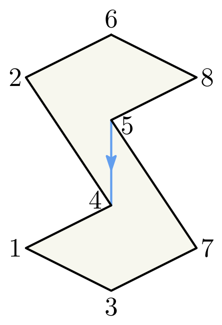
But a different subdivision
with 4 diagonals
is possible!

Note:
Verices are given as input in counterclockwise order. Green = merge vertex, Yellow = split vertex.
h means *helper*, i means vertex, (i,j) means edge. h is only for the upper edge at a start vertex.
T_λ = height-balanced BST containing the edges intersected by λ.
An edge with P above it needn't be inserted in T_λ [see 4-author book].

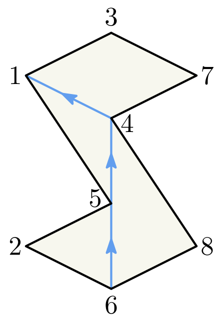
For vertices $13, 11, 9$, the corresponding upper edges—i.e., $(13,14), (11,12), (9,10)$—all have split vertices $(12, 10, 10$, respectively) as helpers, so no diagonals are added to them.

type of v	**type of** $h(e)$	**diagonal added?**
split	any	Yes
any	merge	Yes
start	$\times$	$\times$
end/regular/merge	$\neq$ merge	No

Figure 6: Subdividing a polygon into x-monotone polygons.



v	Operations on T_λ and related actions
1	ins$(1,4)$, $h(1,4) \leftarrow 1$
2	ins$(2,6)$, $h(2,6) \leftarrow 2$
3	$h(1,4) \leftarrow 3$
4	del$(1,4)$, $h(2,6) \leftarrow 4$
5	$(e_{\text{up}}(5) = (2,6)) \wedge (h(2,6) = 4) \implies$ **diagonal** $= (5,4)$ ins$(5,7)$, $h(5,7) \leftarrow 5$, $h(2,6) \leftarrow 5$
6	del$(2,6)$, ins$(6,8)$, $h(6,8) \leftarrow 6$ [$h(2,6) = 5$ (split vertex) $\implies$ no diagonal]
7	del$(5,7)$
8	del$(6,8)$



v	Some of the actions
$\vdots$	
4	$(e_{\text{up}}(4) = (1,3)) \wedge (h(1,3) = 1) \implies$ **diagonal** $= (4,1)$
5	$(e_{\text{up}}(5) = (4,8)) \wedge (h(4,8) = 4) \implies$ **diagonal** $= (5,4)$; $h(4,8) \leftarrow 5$
6	$(e_{\text{up}}(6) = (4,8)) \wedge (h(4,8) = 5) \implies$ **diagonal** $= (6,5)$
$\vdots$	

Figure 7: Subdivision of a polygon into x-monotone polygons: special case. The integer labels on the vertices indicate the order of processing, not their enumeration order in the input set. In this example, the vertex sets $\{1,2\}$, $\{3,4,5,6\}$, and $\{7,8\}$ lie on the same vertical lines. $e_{\text{up}}(v)$ denotes the edge lying immediately above a vertex v. **Top:** Vertices with same x-coordinate are processed from bottom to top. **Bottom:** Vertices with same x-coordinate are processed from top to bottom, producing unwanted collinear diagonals.

Time and space complexities At every event point, there are a fixed number of operations involving T_λ, each operation taking a time logarithmic in the number of edges intersected by λ. As the event points are simply the vertices, the total runtime is $O(n \log n)$. The total space is $O(n)$ both for the event queue and T_λ.

Handling Special Cases If two or more vertices possess the same x-coordinate, they should be processed from bottom to top, as we consider the immediate upper edge e for the helper at each such vertex v. Processing them in the reverse order may introduce unwanted diagonals, as shown in Figure 7.

Alternatively, if we consider the immediate lower edge for the helper, the vertices should be processed in the reverse order.

5 Triangulation of monotone polygons

Given an x-monotone polygon P, we'll triangulate it using the a greedy plane-sweep approach. As P is x-monotone, its input sequence of vertices can be processed in linear time to traverse from left to right using the vertical sweep-line λ. The event points will be the vertices only, with the additional information about their belongingness in upper or lower chains.

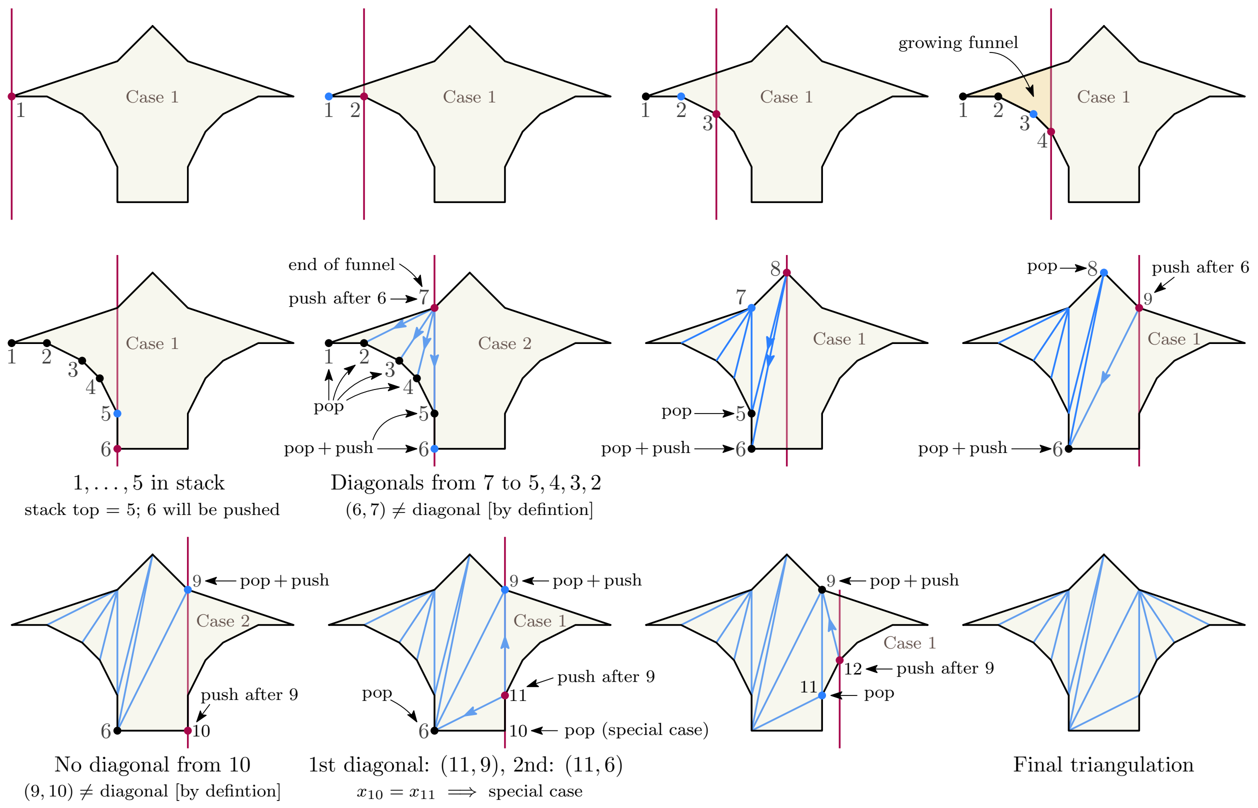


Figure 8: Triangulation of a monotone polygon. Case 1: The current vertex (red) and the stack-top vertex (blue) belong to the same chain (upper or lower). Case 2: The current vertex and the stack-top vertex belong to different chains. Some stack operations are shown; the rest are straightforward.

By the greedy technique, we triangulate whatever possible to the left of λ by adding diagonals, and then remove the triangulated region from further consideration. The algorithm has the optimal linear-time runtime because when we arrive at a vertex, the untriangulated region to the left of λ invariably has a simple *funnel-shaped structure*, defined by a sequence of reflex vertices in one chain and a partial edge on the other end (Figure 8: top row). This structure allows us to determine in constant time whether it is possible to add another diagonal. And, if needed, we can add each diagonal in constant time.

Maintain a stack in which all vertices to the left of λ awaiting diagonals are stored in the order they were encountered by λ.

Denote the current vertex by i. If all vertices have distinct x-coordinates, then the stack-top will be $i-1$. Otherwise, special care is needed (Figure 8: $i = 5, 6$ and $i = 10, 11$). Since three vertices cannot be collinear, no three can share the same x-coordinate.

There are two cases:

Case 1 ($\text{chain}(i) = \text{chain}(i-1)$): If $i-1$ is a reflex vertex, push i onto the stack (Figure 8: $i = 3, \dots, 6$). Otherwise, traverse the reflex chain using the stack, adding diagonals from i to each vertex in this chain as long as possible, and pop them out (Figure 8: $i = 11$). Push $i - 1$ back onto the stack, followed by i. (This step resembles the inner loop of Graham's scan.)

Case 2 ($\text{chain}(i) \neq \text{chain}(i-1)$): Add diagonals connecting i to all vertices in the stack, except the bottommost one (which already has an edge with i), and pop them out. Push $i - 1$ back onto the stack, followed by i.

In either case, the funnel invariant is preserved.