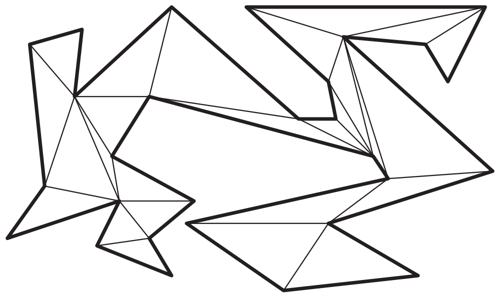
G Polygon Triangulation

G.1 Introduction

Recall from last time that a *polygon* is a region of the plane bounded by a cycle of straight edges joined end to end. Given a polygon, we want to decompose it into triangles by adding *diagonals*: new line segments between the vertices that don't cross the boundary of the polygon. Because we want to keep the number of triangles small, we don't allow the diagonals to cross. We call this decomposition a *triangulation* of the polygon. Most polygons can have more than one triangulation; we don't care which one we compute.

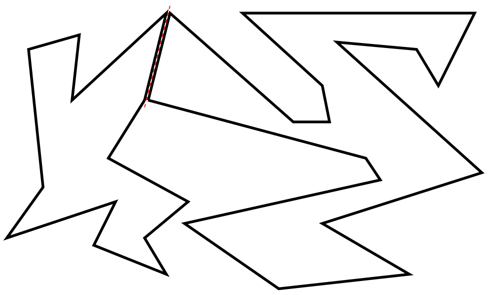


Two triangulations of the same polygon.

Before we go any further, I encourage you to play around with some examples. Draw a few polygons (making sure that the edges are straight and don't cross) and try to break them up into triangles.

G.2 Existence and Complexity

If you play around with a few examples, you quickly discover that every triangulation of an n-sided has $n-2$ triangles. You might even try to prove this observation by induction. The base case $n = 3$ is trivial: there is only one triangulation of a triangle, and it obviously has only one triangle! To prove the general case, let P be a polygon with n edges. Draw a diagonal between two vertices. This splits P into two smaller polygons. One of these polygons has k edges of P plus the diagonal, for some integer k between 2 and $n-2$, for a total of $k+1$ edges. So by the induction hypothesis, this polygon can be broken into $k-1$ triangles. The other polygon has $n-k+1$ edges, and so by the induction hypothesis, it can be broken into $n-k-1$ tirangles. Putting the two pieces back together, we have a total of $(k-1)+(n-k-1) = n-2$ triangles.



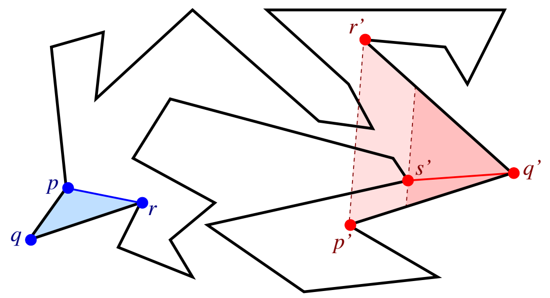
Breaking a polygon into two smaller polygons with a diagonal.

This is a fine induction proof, which any of you could have discovered on your own (right?), except for one small problem. How do we know that every polygon *has* a diagonal? This seems patently obvious, but it's surprisingly hard to prove, and in fact many incorrect proofs were actually published as late as 1975. The following proof is due to Meisters in 1975.

Lemma 1. *Every polygon with more than three vertices has a diagonal.*

Proof: Let P be a polygon with more than three vertices. Every vertex of a P is either *convex* or *concave*, depending on whether it points into or out of P, respectively. Let q be a convex vertex, and let p and r be the vertices on either side of q. For example, let q be the leftmost vertex. (If there is more than one leftmost vertex, let q be the the lowest one.) If $\overline{pr}$ is a diagonal, we're done; in this case, we say that the triangle $\triangle pqr$ is an *ear*.

If pr is not a diagonal, then $\triangle pqr$ must contain another vertex of the polygon. Out of all the vertices inside $\triangle pqr$, let s be the vertex furthest away from the line $\overleftrightarrow{pr}$. In other words, if we take a line parallel to $\overleftrightarrow{pr}$ through q, and translate it towards $\overleftrightarrow{pr}$, then then s is the first vertex that the line hits. Then the line segment $\overline{qs}$ is a diagonal. $\square$



The leftmost vertex q is the tip of an ear, so pr is a diagonal.
The rightmost vertex q' is not, since $\triangle p'q'r'$ contains three other vertices. In this case, $q's'$ is a diagonal.

G.3 Existence and Complexity

Meister's existence proof immediately gives us an algorithm to compute a diagonal in linear time. The input to our algorithm is just an array of vertices in counterclockwise order around the polygon. First, we can find the (lowest) leftmost vertex q in $O(n)$ time by comparing the x-coordinates of the vertices (using y-coordinates to break ties). Next, we can determine in $O(n)$ time whether the triangle $\triangle pqr$ contains any of the other $n-3$ vertices. Specifically, we can check whether one point lies inside a triangle by performing three counterclockwise tests. Finally, if the triangle is not empty, we can find the vertex s in $O(n)$ time by comparing the areas of every triangle $\triangle pqs$; we can compute this area using the counterclockwise determinant.

Here's the algorithm in excruciating detail. We need three support subroutines to compute the area of a polygon, to determine if three poitns are in counterclockwise order, and to determine if a point is inside a triangle.

⟨⟨*Return twice the signed area of* $\triangle P[i]P[j]P[k]$⟩⟩
<u>AREA(i, j, k):</u>
  return $(P[k].y - P[i].y)(P[j].x - P[i].x) - (P[k].x - P[i].x)(P[j].y - P[i].y)$

⟨⟨*Are* $P[i], P[j], P[k]$ *in counterclockwise order?*⟩⟩
<u>CCW(i, j, k):</u>
  return AREA$(i, j, k) > 0$

⟨⟨*Is* $P[i]$ *inside* $\triangle P[p]P[q]P[r]$*?*⟩⟩
<u>INSIDE(i, p, q, r):</u>
  return CCW(i, p, q) and CCW(i, q, r) and CCW(i, r, p)

```
FindDiagonal(P[1..n]):
  q ← 1
  for i ← 2 to n
      if P[i].x < P[q].x
          q ← i
  p ← q − 1 mod n
  r ← q + 1 mod n
  ear ← True
  s ← p
  for i ← 1 to n
      if i ≤ p and i ≠ q and i ≠ r and Inside(i, p, q, r)
          ear ← False
          if Area(i, r, p) > Area(s, r, p)
              s ← i
  if ear = True
      return (p, r)
  else
      return (q, s)
```

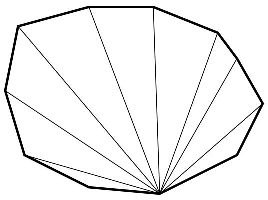
Once we have a diagonal, we can recursively triangulate the two pieces. The worst-case running time of this algorithm satisfies almost the same recurrence as quicksort:

$$T(n) \le \max_{2\le k\le n-2} T(k+1) + T(n-k+1) + O(n).$$

Just like quicksort, the solution is $T(n) = O(n^2)$. So we can now triangulate any polygon in quadratic time.

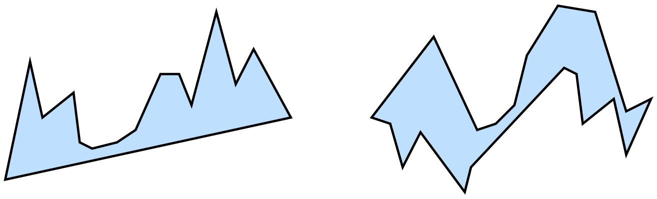
G.4 Faster Special Cases

For certain special cases of polygons, we can do much better than $O(n^2)$ time. For example, we can easily triangulate any convex polygon by connecting any vertex to every other vertex. Since we're given the counterclockwise order of the vertices as input, this takes only $O(n)$ time.



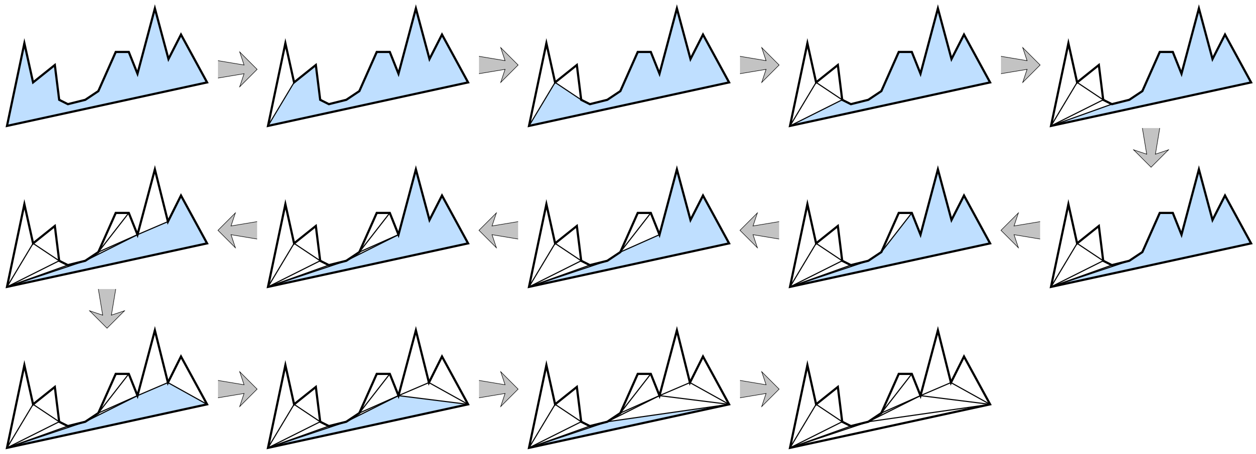
Triangulating a convex polygon is easy.

Another easy special case is *monotone mountains*. A polygon is *monotone* if any vertical line intersects the boundary in at most two points. A monotone polygon is a *mountain* if it contains an edge from the rightmost vertex to the leftmost vertex. Every monotone polygon consists of two chains of edges going left to right between the two extreme vertices; for mountains, one of these chains is a single edge.



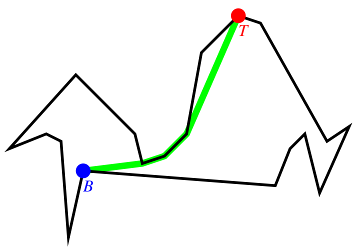
A monotone mountain and a monotone non-mountain.

Triangulating a monotone mounting is extremely easy, since every convex vertex is the tip of an ear, except possibly for the vertices on the far left and far right. Thus, all we have to do is scan through the intermediate vertices, and when we find a convex vertex, cut off the ear. The simplest method for doing this is probably the three-penny algorithm used in the "Graham's scan" convex hull algorithm—instead of filling in the outside of a polygon with triangles, we're filling in the inside, both otherwise it's the same process. This takes $O(n)$ time.



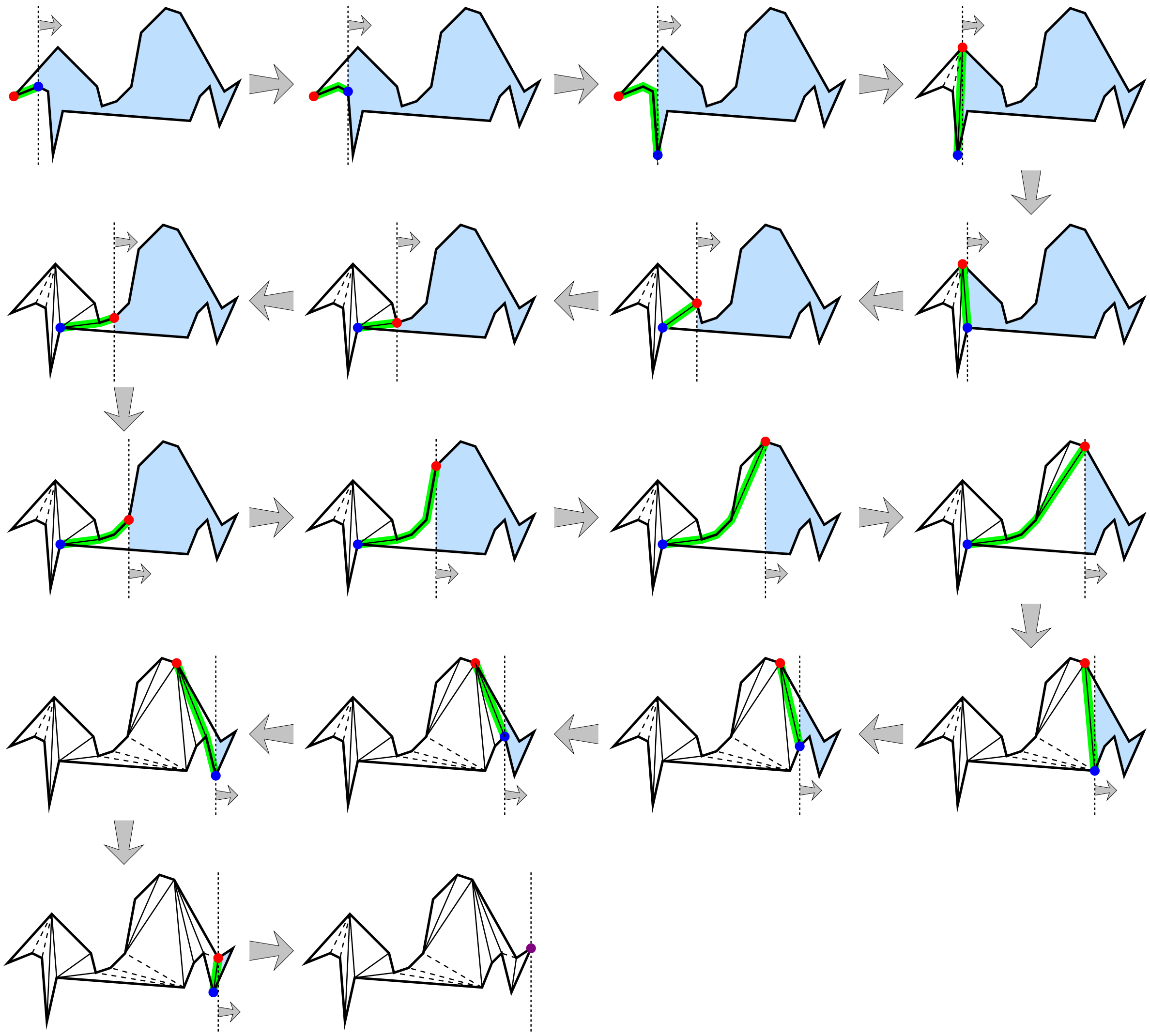
Triangulating a monotone mountain. (Some of the triangles are very thin.)

We can also triangulate general monotone polygons in linear time, but the process is more complicated. A good way to visualize the algorithm is to think of the polygon as a complicated room. Two people named Tom and Bob are walking along the top and bottom walls, both starting at the left end and going to the right. At all times, they have a rubber band stretched between them that can never leave the room.



A rubber band stretched between a vertex on the top and a vertex on the bottom of a monotone polygon.

Now we loop through *all* the vertices of the polygon in order from left to right. Whenever we see a new bottom vertex, Bob moves onto it, and whenever we see a new bottom vertex Tom moves onto it. After either person moves, we cut the polygon along the rubber band. (In fact, this will only cut the polygon along a single diagonal at any step.) When we're done, the polygon is decomposed into triangles and *boomerangs*—nonconvex polygons consisting of two straight edges and a concave chain. A boomerang can only be triangulated in one way, by joining the *apex* to every vertex in the concave chain.



Triangulating a monotone polygon by walking a rubber band from left to right.

I don't want to go into too many implementation details, but a few observations shoudl convince you that this algorithm can be implemented to run in $O(n)$ time. Notice that at all times, the rubber band forms a concave chain. The leftmost edge in the rubber band joins a top vertex to a bottom vertex. If the rubber band has any other vertices, either they are all on top or all on bottom. If all the other vertices are on top, there are just three ways the rubber band can change:

1. The bottom vertex changes, the rubber band straighens out, and we get a new boomerang.

2. The top vertex changes and the rubber band gets a new concave vertex.

3. The top vertex changes, the rubber band loses some vertices, and we get a new boomerang.

Deciding between the first case and the other two requires a simple comparison between x-coordinates. Deciding between the last two requires a counterclockwise test.