1. Let P be an arbitrary simple polygon with $n \geq 4$ vertices, and let T be an arbitrary frugal triangulation of P. Two triangles in T are *adjacent* if they share an edge. The *degree* of a triangle $\triangle$ is the number of other triangles in T that adjacent to $\triangle$. Finally, for any integer d, let t_d denote the number of triangles with degree d.

   (a) Prove that $t_1 + t_2 + t_3 = n - 2$.

Solution (induction): Let t be the total number of triangles in triangulation T. I will prove by induction that $t = n - 2$. The claim is trivial when $n = 3$, so assume otherwise.

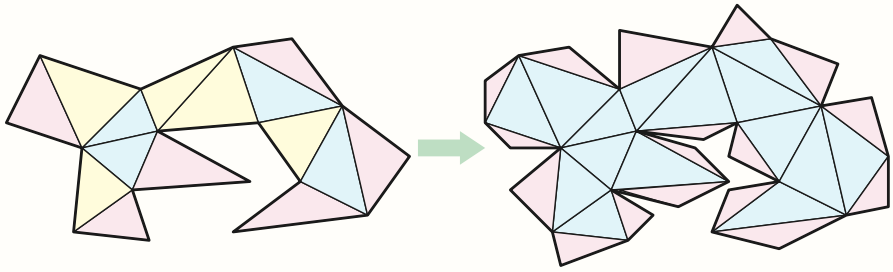
Choose an arbitrary diagonal uv in T. Cutting along uv splits P into two smaller polygons $P^\flat$ and $P^\sharp$ and splits T into two smaller triangulations $T^\flat$ and $T^\sharp$. Suppose $P^\flat$ has $n^\flat$ vertices, $P^\sharp$ has $n^\sharp$ vertices, $T^\flat$ has $t^\flat$ triangles, and $T^\sharp$ has $t^\sharp$ triangles. These variables are related as follows:

- $n = n^\flat + n^\sharp - 2$ — Every vertex of P is a vertex of either $P^\flat$ or $P^\sharp$, except u and v, which are vertices of both.
- $t^\flat = n^\flat - 2$ by the induction hypothesis
- $t^\sharp = n^\sharp - 2$ by the induction hypothesis
- $t = t^\flat + t^\sharp$ — Every triangle in T is in exactly one of the smaller triangulations.

We conclude that $t = t^\flat + t^\sharp = n^\flat + n^\sharp - 4 = n - 2$. ■

Solution (Euler): Let V, E, and F denote the number of vertices, edges, and faces in the triangulation T. Every vertex of T is a vertex of P, so $V = n$. Counting all incidences between edges and faces gives us $2E = 3(F-1) + n$. Euler's formula implies that $V - E + F = n - \frac{1}{2}(3(F-1) + n) + F = 2$, which simplifies to $F = n - 1$. Since only one face is not a triangle, we conclude that T has $n - 2$ triangular faces. ■

Solution (from part (b)): Let T^+ be a polygon triangulation obtained from T by gluing new triangles to every edge of P, as shown below.



Every triangle in T^+ has degree 1 or 3; specifically, T^+ has n triangles with degree 1 and $t_1 + t_2 + t_3$ triangles with degree 3. Applying part (b) to T^+ immediately gives us $t_1 + t_2 + t_3 = n - 2$. ■

Solution (angles): The sum of the signed *exterior* angles of any simple polygon is exactly 1.[a] The internal and external angles at each vertex sum to $\frac{1}{2}$. Thus, the sum of all the *interior* angles of P is $\frac{n}{2} - 1 = \frac{1}{2}(n-2)$.

Every interior angle in P decomposes into the sum of interior angles of triangles in T. Thus, the sum of the interior angles in all triangles of T is also $\frac{1}{2}(n-2)$. The sum of the interior angles in each triangle is $\frac{1}{2}$, so there must be $n-2$ triangles. ■

[a] I am measuring angles in *circles*, as God intended. For example, a right angle is $\frac{1}{4}$ and a flat angle is $\frac{1}{2}$. For unfamiliar heretics: one circle equals either 2π or τ radians, or 360 degrees.

Rubric: 5 points. These are not the only correct solutions.

(b) Prove that $t_1 - t_3 = 2$.

Solution (double-counting): I'll count the edges of the dual tree T^* in two different ways.

- Because T^* is a tree, the number of edges in T^* is one less than its number of vertices: $E = t_1 + t_2 + t_3 - 1$.
- On the other hand, the sum of vertex degrees in any graph is twice the number of edges: $2E = t_1 + 2t_2 + 3t_3$.

Together these expressions imply $2t_1 + 2t_2 + 2t_3 - 2 = t_1 + 2t_2 + 3t_3$; cancelling equal terms simplifies this equation to $t_1 - 2 = t_3$. ■

Solution (Euler): We can count the vertices, edges, and faces of the triangulation T as follows:

- Every vertex of T is also a vertex of the polygon P, so $V = n$.
- Every edge of T is either an edge of P or a diagonal. P has exactly n edges, and the number of diagonals is $\frac{1}{2}(t_1 + 2t_2 + 3t_3)$. Thus, $E = n + \frac{1}{2}(t_1 + 2t_2 + 3t_3)$.
- Finally, every face of T is either a triangle or the unbounded outer face, so $F = t_1 + t_2 + t_3 + 1$.

Now Euler's formula $V - E + F = 2$ implies $n - (n + \frac{1}{2}(t_1 + 2t_2 + 3t_3)) + t_1 + t_2 + t_3 + 1 = 2$, which simplifies to $t_1 - t_3 = 2$. ■

Solution (induction): Let e be any diagonal in T. Cutting along e splits the triangulation T into smaller triangulations $T^\flat$ and $T^\sharp$. Let $\triangle^\flat$ and $\triangle^\sharp$ be the triangles incident to r in $T^\flat$ and $T^\sharp$, respectively. Let $t_1^\flat$ denote the number of degree-1 triangles in $T^\flat$, and define $t_3^\flat$, $t_1^\sharp$, and $t_3^\sharp$ similarly.

If we assume without loss of generality that $\deg(\triangle^\flat) \geq \deg(\triangle^\sharp)$, there are six cases to consider:

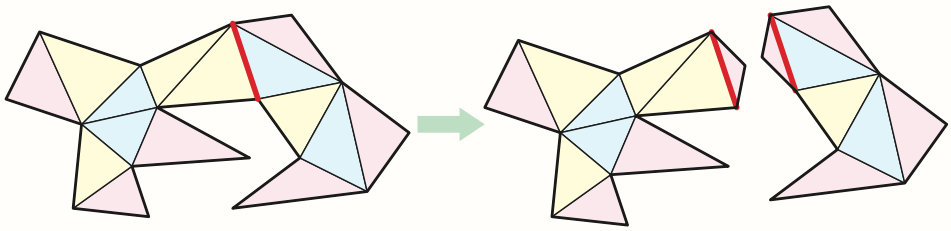
- If $\deg(\triangle^\flat) = \deg(\triangle^\sharp) = 3$, then $t_3 = t_3^\flat + t_3^\sharp + 2$ and $t_1 = t_1^\flat + t_1^\sharp$.
- If $\deg(\triangle^\flat) = 3$ and $\deg(\triangle^\sharp) = 2$, then $t_3 = t_3^\flat + t_3^\sharp + 1$ and $t_1 = t_1^\flat + t_1^\sharp - 1$.
- If $\deg(\triangle^\flat) = 3$ and $\deg(\triangle^\sharp) = 1$, then $t_3 = t_3^\flat + 1$ and $t_1 = t_1^\flat + 1$.
- If $\deg(\triangle^\flat) = \deg(\triangle^\sharp) = 2$, then $t_3 = t_3^\flat + t_3^\sharp$ and $t_1 = t_1^\flat + t_1^\sharp - 2$.
- If $\deg(\triangle^\flat) = 2$ and $\deg(\triangle^\sharp) = 1$, then $t_3 = t_3^\flat$ and $t_1 = t_1^\flat$.

- Finally, if $\deg(\triangle^\flat) = \deg(\triangle^\sharp) = 1$, then $t_3 = 0$ and $t_1 = 2$.

In the first five cases, the induction hypothesis implies $t_1 = t_3 + 2$; in the last case, the equation $t_1 = t_3 + 2$ is immediate. ■

Solution (induction′): We can verify the claim for all triangulations with at most 4 triangles by brute force enumeration. We prove the claim for larger values of n by induction as follows:

Let e be any diagonal in T, such that cutting along e splits T into two smaller triangulations, *each with at least two triangles*; in other words, e is *not* the base of an ear in T. Problem 1 in Homework 2 implies that such a diagonal must exist. Expand these smaller triangulations by gluing new triangles $\triangle^\flat$ and $\triangle^\sharp$ to the two copies of e, as shown below.



Call the two resulting triangulations $T^\flat$ and $T^\sharp$. Let $n^\flat$ be the number of vertices of $T^\flat$, and define $n^\sharp$, $t_1^\flat$, $t_3^\flat$, $t_1^\sharp$, and $t_3^\sharp$ similarly.

Because we duplicated the endpoints of e and added two new vertices, we have $n^\flat + n^\sharp = n + 4$. Because e does not cut off an ear of T, we have $\min\{n^\flat, n^\sharp\} \geq 5$ and therefore $\max\{n^\flat, n^\sharp\} \leq n - 1$. So the inductive hypothesis implies that $t_1^\flat = t_3^\flat + 2$ and $t_1^\sharp = t_3^\sharp + 2$. We conclude that

$$t_1 \;=\; t_1^\flat + t_1^\sharp - 2 \;=\; (t_3^\flat + 2) + (t_3^\sharp + 2) - 2 \;=\; t_3 + 2.$$

■

Rubric: 5 points. These are not the only correct solutions.

2. Suppose you are given an array $H[1..n]$ of points, each with coordinates $H[i].x$ and $H[i].y$. Describe and analyze an algorithm to determine whether H is the counterclockwise sequence of vertices of a convex polygon. For full credit, your algorithm should run in linear time.

Solution: The following algorithm closely follows Graham's scan:

```
VerifyConvex(H[0..n-1]):
  Find the leftmost point of H
  Rotate indices so that H[0] is leftmost point of H
  ⟨⟨Verify that other points are sorted ccw around H[0]⟩⟩
  for i ← 1 to n-2
      if Orient(H[1], H[i], H[i+1]) < 0
          return False
  ⟨⟨Verify that other corners of H are convex⟩⟩
  for i ← 2 to n-2
      if Orient(H[i-1], H[i], H[i+1]) < 0
          return False
  return True
```

■

Solution: We need to verify that every consecutive triple of points is oriented counterclockwise and that the array of x-coordinates is bitonic.

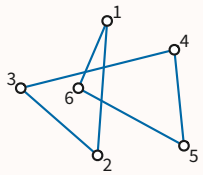
```
VerifyConvex(H[0..n-1]):
  minxfound ← False
  for i ← 0 to n-1
      p ← H[(i-1) mod n]
      q ← H[i]
      r ← H[(i+1) mod n]
      if Orient(p, q, r) < 0
          return False
      if q.x = min{p.x, q.x, r.x}
          if minxfound
              return False
          else
              minxfound ← True
  return True
```

■

Rubric: 10 points. These are not the only correct answers.

- 8.999 points for angle-based solution (see below)
- Max 8 points for $O(n \log n)$ time algorithm (for example, computing the convex hull of H from scratch and comparing the result to H)
- Max 5 points for only checking orientation of consecutive triples (see below)

Non-solution: Just checking that every consecutive triple of points is oriented counterclockwise is not enough; the polygon might self-intersect. Consider the second example in the exam handout.



The three-penny algorithm from Graham's scan works only because it maintains the following invariant: *All surviving vertices are sorted in counterclockwise around the leftmost vertex.* Among other things, this invariant implies that the polygon is simple.

If we run the three-penny algorithm on a polygon P that violates this invariant, the final polygon may not be simple, and therefore may not be convex, *even if the initial polygon P is simple*. ♣

Non-solution (can of worms): We compute the exterior angles at every vertex of the polygon whose vertex sequence is H. The polygon is convex if and only if every exterior angle is positive and the sum of the exterior angles is one full circle.

```
VerifyConvex(H[0..n-1]):
  rot ← 0                          ⟨⟨rotation number⟩⟩
  for i ← 0 to n-1
      u ← H[i] - H[(i-1) mod n]
      v ← H[(i+1) mod n] - H[i]
      cross ← u.x · v.y - u.y · v.x
      dot   ← u.x · v.x + u.y · v.y
      angle ← atan2(dot, cross)
      if angle < 0
          return False             ⟨⟨reflex angle⟩⟩
      rot ← rot + angle/2π
  if rot = 1
      return True
  else
      return False                 ⟨⟨not simple⟩⟩
```

The variable *cross* is the orientation determinant for the triple $H[i-1]$, $H[i]$, $H[i+1]$. The standard function $\texttt{atan2}(y,x)$ computes the argument of the complex number $x+iy$, as an angle (in radians, feh) between $-\pi$ and π.

This algorithm appears to run in $O(n)$ time, but it suffers from a rather severe problem: ***We cannot actually compute angles exactly or quickly***. In practice, `atan2` is implemented using a Taylor series approximation, which is slow, and suffers from numerical precision issues when the exact angle is close to an integer multiple of π.

Somewhat more subtly, working with angles is also problematic even in theory. Geometric algorithms are normally designed and analyzed in the *real RAM* model of

computation, which supports exact real arithmetic operations ($+, -, \times, \div, >, \geq, <, \leq$) in constant time *by definition*. Crucially, this model does *not* include constant-time rounding ($\lfloor \cdot \rfloor$), modular arithmetic, or direct access to the binary representation of real numbers. A real RAM model with any of those extensions can, among other impossible feats, solve any problem in PSPACE in a polynomial number of steps [Hartmanis and Simon 1974].

An extended real RAM model that supports trigonometric (or equivalently, complex exponential and logarithmic) functions in constant time can implement the floor function in constant time as $\lfloor x \rfloor = x - \arcsin(\sin(2\pi x))/2\pi$, where $\pi = 2\arcsin(1)$. I don't know whether supporting `atan2` alone would be enough to release this particular complexity-theoretic kraken, but I'd rather not tempt fate.

tl;dr: Angles bad. Fire burn.

Fortunately, we don't really need the precise values of the angles; we just need to know that they're all positive, and that the edge vectors make only one rotation around the origin. That's what my second solution does. ♣

3. Let $R[1..n]$ be an array of rectangles ("buildings") whose bottom edges are on the x-axis. Each rectangle is represented by its left x-coordinate $R[i].l$, its right x-coordinate $R[i].r$, and its height $R[i].h$. The *skyline* of R is the boundary of the set of points that are above the x-axis and outside every rectangle in R.

   Describe and analyze an efficient algorithm to compute the skyline of R.

Solution (sweep): We use a standard sweep algorithm. The vertical sweep line ℓ maintains a *priority queue* of the heights of the buildings that it intersects, plus the sentinel value 0, subject to the operations INSERT, DELETE, and FINDMAX.

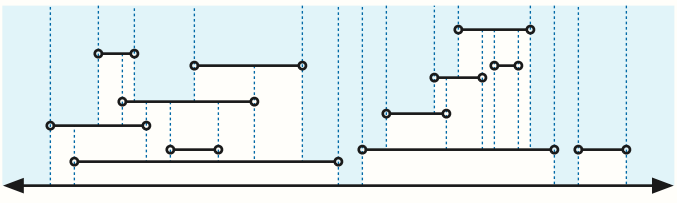
- Whenever ℓ reaches the left side of a rectangle, we INSERT its height.
- Whenever ℓ reaches the right side of a building, we DELETE its height.
- Whenever the output of FINDMAX changes, we add two vertices to the skyline.

Sorting the $2n$ sweep events (x-coordinates) takes $O(n \log n)$ time. Each event requires two priority queue operations (one INSERT or DELETE, and one FINDMAX), so if we implement the priority queue as either a binary heap or a balanced binary search tree, each event is handled in $O(\log n)$ time. The total running time is $O(n \log n)$.

```
MANHATTANSKYLINE(R[1..n]):
  X ← sorted array containing R[i].l and R[i].r for all i
  INSERT(0)
  height ← 0      ⟨⟨current height of skyline⟩⟩
  Skyline ← []    ⟨⟨sequence of skyline vertices⟩⟩
  for i ← 1 to 2n
      if X[i] is a left x-coordinate
          INSERT height of corresponding rectangle
      else if X[i] is a right x-coordinate
          DELETE height of corresponding rectangle
      if FINDMAX( ) ≠ height
          append (X[i], height) to Skyline
          height ← FINDMAX( )
          append (X[i], height) to Skyline
  return Skyline
```

■

Solution (delegate the hard part): Build a trapezoidal decomposition of the top segments of R and the x-axis in $O(n \log n)$ time, using either of the algorithms described in class.



Then for each trapezoid $\triangle$ that is unbounded upward, in order from left to right, output the bottom edge of $\triangle$.

Walking through the $O(n)$ unbounded trapezoids and recording their bottom edges takes $O(n)$ time, so the overall running time is $O(n \log n)$. ■

Rubric: 10 points. These are not the only correct solutions.

Solution (extra credit!): With some extra work, we can reduce the running time from $O(n \log n)$ to $O(n \log h)$, where h is the complexity of the skyline, essentially by following Chan's algorithm.

First, suppose we somehow know an integer $\hbar$ such that $h \le \hbar \le h^2$. We arbitrarily partition R into $k = O(n/\hbar)$ subsets, each containing $O(\hbar)$ rectangles, and then compute the skyline of each subset, in $O(n/\hbar) \cdot O(\hbar \log \hbar) = O(n \log h)$ total time. Call these smaller skylines $S_1, S_2, \ldots, S_k$. We need to merge these k skylines together.

Following Chan's algorithm, after preprocessing the skylines S_i (as described below), we build the global skyline vertex by vertex. The first vertex of the skyline is the point $(\min_i R[i].l, 0)$; we can find this point and the corresponding rectangle $R[i]$ in $O(n)$ time by brute force. The next edge of the skyline goes upward from that vertex along the left edge of $R[i]$.

For every later step of the trace, there are three cases to consider:

- Suppose the trace is moving upward along the left edge of some rectangle $R[i]$. Then the next skyline vertex is the upper left corner $(R[i].l, R[i].h)$, and the next edge moves to the right along the top edge of $R[i]$.
- Suppose the trace is moving to the right along the top edge of some rectangle $R[i]$. (If we add a sentinel rectangle $R[0]$ with x-coordinates $-\infty$ and $+\infty$, and height 0, we can include tracing along the ground in this case.) Then the next skyline vertex is either the top right endpoint $(R[i].r, R[i].h)$ (after which the trace will move downward) or a point on the left side of some rectangle $R[j]$ (after which the trace will move upward).

  In this case, we need to find the first vertical edge in any S_i hit by a ray shooting right from the point $(R[i].l, R[i].h)$. Before tracing, we preprocess each skyline S_i as described below to support rightward ray-shooting queries in $O(\log \hbar)$ time. Performing a ray-shooting query in each S_i takes $O((n/\hbar) \log h)$ time.

  To preprocess S_i, we build a trapezoidal decomposition of the vertical edges of S_i, with horizontal walls instead of vertical walls. If we use a randomized incremental construction, we can build both the decomposition and its history dag in $O(\hbar \log \hbar)$ expected time. Then any rightward ray shooting query can be answered by tracing through the history dag in $O(\log \hbar)$ expected time. The total expected time to preprocess every S_i is $O(n/\hbar) \cdot O(\hbar \log \hbar) = O(n \log h)$.
- Finally, suppose the trace is moving downward along the right edge of some rectangle $R[i]$. Then the next skyline vertex is on the top edge of some rectangle $R[j]$ (possibly the sentinel/ground rectangle).

In this case, we need to find the first horizontal edge in any S_i hit by a ray ρ shooting downward from $(R[i].r, R[i].h)$. Assuming each skyline S_i is represented as a sorted array of vertices, we can find the first horizontal edge of S_i hit by ρ using a simple binary search in $O(\log \hbar) = O(\log h)$ time. Repeating this search for each S_i takes $O((n/\hbar)\log h)$ time.

In summary, we spend at most $O((n/\hbar)\log h)$ time finding each vertex of the final skyline. Since the skyline has h vertices by definition, the overall running time of the tracing algorithm is $O((nh/\hbar)\log h) = O(n \log h)$.

Finally, to obtain the appropriate value of $\hbar$, we run the algorithm described above for the sequence of values $\hbar = 4, 4^2, 4^4, \ldots, 4^{2^i}, \ldots$. Within each iteration, if the trace visits more than $\hbar$ vertices, we abort and proceed to the next iteration. Just as in Chan's algorithm, the expected running times of each iteration form an increasing geometric series, which is dominated by its largest term $O(n \log h)$. ■

Rubric: 15 points. This is not the only correct algorithm with this running time. In particular, $O(n \log h)$ worst-case time can be achieved using a deterministic point-location data structure.

4. Suppose you are given two convex polygons P and Q, represented as usual by arrays of vertices in counterclockwise order. Describe an algorithm to determine whether P lies entirely inside Q. For full credit, your algorithm should run in linear time.

Solution (sweep): As a sanity check, we start by checking point $P[1]$ is inside Q, using the $O(n)$-time point-in-polygon test. If $P[1]$ is outside Q, we immediately return FALSE. If $P[1]$ is inside Q, but the entire polygon P is not inside Q, then some edge of P and some edge of Q must intersect.

We use the Bentley-Ottmann sweep algorithm to find intersections between the edges of P and Q, with two changes to reduce the running time from $O(n \log n)$ to $O(n)$

First, we compute the sorted sequence of events (that is, the vertices sorted by x-coordinate from left to right) in $O(n)$ time as follows. We start by computing the leftmost and rightmost vertices of P and Q by brute force. These extreme vertices partition P and Q into the lower hulls P^- and Q^-, which are sorted from left to right, and the upper hulls P^+ and Q^+, which are sorted from right to left. Merging P^+, Q^+, the reversal of P^-, and the reversal of Q^- gives us the complete sorted sequence of events in $O(n)$ time.

Second, the main sweep algorithm now can process each event in $O(1)$ time. At all times, the sweepline intersects at most four edges. Thus, at each event, we can update the sequence of intersected edges and check for possible intersections *by brute force* in $O(1)$ time. Since there are n events (one for each vertex of P and Q), the entire algorithm runs in $O(n)$ time. ■

Solution (trace): We start by sorting the vertices of Q from left to right. We can do this in $O(n)$ time by identifying the leftmost and rightmost vertices, and then merging the lower hull Q^- with the reversal of the upper hull Q^+. Intuitively, we are decomposing the plane into trapezoids using the edges of Q and vertical lines through the vertices of Q. For each slab between two adjacent vertical lines, we record the two edges of Q that the slab intersects.

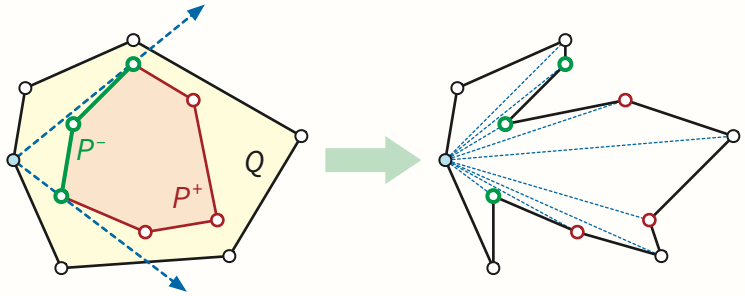
Then we locate one vertex $P[1]$ in the resulting trapezoidal decomposition in $O(\log n)$ time. First we find the slab containing $P[1]$ using a binary search on x-coordinates; then we check whether $P[1]$ is above the unique edge of Q^- and below the unique edge of Q^+ that intersect that slab.

Finally, in the main algorithm, we trace the entire boundary of P, starting at $P[1]$, similarly to the randomized incremental algorithm for building trapezoidal decompositions. In the middle of the trace, the moving point is inside some slab s and on some edge $P[i]P[i+1]$, moving toward $P[i+1]$. In constant time, we can determine in $O(1)$ time whether the moving point first reaches $P[i+1]$, one of the vertical walls of s, or one of the edges of Q crossing s. If the moving point ever crosses an edge of Q, we immediately return FALSE. Otherwise, the current edge of P changes exactly n times, and the current slab changes at most $2n$ times (because P crosses any vertical line at most twice), so the entire trace takes $O(n)$ time. ■

Solution (convex hull): P is inside Q if and only if Q is equal to the convex hull of $P \cup Q$. We can compute the convex hull of $P \cup Q$ in $O(n)$ time, using the following modification of Graham's scan.

First, we identify the leftmost point in $P \cup Q$ by brute force in $O(n)$ time. If this leftmost point is a vertex of P, we immediately return FALSE. Otherwise, if necessary, we rotate the array Q so that the leftmost point is $Q[1]$.

Next, we find the upper and lower tangent lines to P through $Q[1]$, either by brute force or using the modified binary search employed in Chan's algorithm. These two lines partition P into an outer convex chain P^+, whose vertices are sorted in counterclockwise order around $Q[1]$, and an inner convex chain P^-, whose vertices are sorted in clockwise order around $Q[1]$.



Now we sort the vertices of $P \cup Q$ in counterclockwise around $Q[1]$ by merging P^+, the reversal of P^-, and the rest of Q in $O(n)$ time. The resulting array of points describes a simple polygon R with the correct preconditions for the three-penny algorithm. (See the first non-solution to problem 2!)

Finally, the three-penny algorithm computes the convex hull of R in $O(n)$ time.

Let R be the resulting convex hull, stored as usual as an array of vertices in counterclockwise order. Again, if necessary, we rotate the array R so that $R[1]$ is the leftmost vertex of R. Finally, we confirm that $R[i] = Q[i]$ for every index i, in $O(n)$ time. ■

Solution (dual sweep): Because Q is convex, P is contained in Q if and only if every vertex of P is contained in Q. We can express this condition more explicitly as follows. Let $\Delta(p, q, r)$ denote the usual orientation determinant

$$\Delta(p,q,r) := \det \begin{bmatrix} 1 & p.x & p.y \\ 1 & q.x & q.y \\ 1 & r.x & r.y \end{bmatrix}$$

Then P is inside Q if and only if, for every vertex p of P and every (counterclockwise) edge qr of Q, we have $\Delta(p, q, r) > 0$. Testing every vertex-edge pair by brute force would take $O(n^2)$ time.

But we don't need to check every vertex-edge pair! If we fix the edge qr, the function $p \mapsto \Delta(p, q, r)$ is bitonic (and strictly so if we assume general position). So

to prove that $\Delta(p,q,r) > 0$ for *every* vertex p, it's enough to find the single local minimum vertex p and verify that $\Delta(p,q,r) > 0$.

```
NESTED(P[0..m−1], Q[0..n−1]):
  p ← P[0]
  i ← 1
  p′ ← P[1]
  for j ← 0 to n−1
      q ← Q[j]
      q′ ← Q[j+1 mod n]
      while Δ(p,q,q′) > Δ(p′,q,q′)
          p ← P[i]
          i ← i+1 mod m
          p′ ← P[i]
      if Δ(p,q,q′) < 0
          return FALSE
  return TRUE
```

The algorithm clearly considers each edge qq' of Q exactly once. Somewhat more subtly, it also considers each vertex p of P at most twice: at most once in the first iteration of the for loop, and at most once thereafter. We prove this as follows:

Let $n(qq')$ denote the outward unit normal vector of any edge qq' of Q. The vertex p that minimizes $\Delta(p,q,q')$ also maximizes the dot product $p \cdot n(qq')$. During the algorithm, the vector $n(qq')$ makes one complete counterclockwise rotation. It follows that the vertex p minimizing $\Delta(p,q,q')$ makes one complete counterclockwise rotation around P.

We conclude that the algorithm runs in $O(m+n)$ time. ■

Rubric: 10 points. These are not the only correct solutions. These are not even the only ways to implement these high-level strategies in linear time.

- Max 8 points for $O(n \log n)$-time algorithm (for example, computing the convex hull of $P \cup Q$ from scratch and comparing the result to Q)
- Max 6 points for $O(n^2)$-time algorithm (for example, performing an independent point-in-polygon test for each vertex of P)