3. *Optimal incremental algorithm.* Presort the points by their x coordinate, so that $p \notin Q$ at each step. Now try to arrange the search for tangent lines in such a manner that the total work over the life of the algorithm is $O(n)$. This then provides an $O(n \log n)$ algorithm.[22]

3.8. DIVIDE AND CONQUER

The final two-dimensional hull algorithm we consider achieves optimal $O(n \log n)$ time by a method completely different than Graham's algorithm: divide and conquer. This is the only technique known to extend to three dimensions and achieve the same asymptotically optimal $O(n \log n)$ time complexity. It is therefore well worth studying, even though in two dimensions it is relatively complicated.

3.8.1. Divide-and-Conquer Recurrence Relation

"Divide and conquer" is a general paradigm for solving problems that has proved very effective in computer science. The essence is to partition the problem into two (nearly) equal halves, solve each half recursively, and create a full solution by "merging" the two half solutions. When the recursion reduces the original problem down to very small subproblems, they are usually quite easy to solve. All the work therefore lies in the merge step.

Let $T(n)$ be the time complexity of a divide-and-conquer hull algorithm for n points. If the merge step can be accomplished in linear time, then we have the recurrence relation $T(n) = 2T(n/2) + O(n)$: The two problems of half size take $2T(n/2)$ time, and the merge takes $O(n)$ time. As we mentioned before, this has solution $O(n \log n)$.

Exercises 3.8.5[1] and [2] ask for exploration of the recurrence relation when the merge step is less efficient; the conclusion is that the merge must be $O(n)$ to achieve optimality.

3.8.2. Algorithm Description

The divide-and-conquer technique was first applied to the convex hull problem by Preparata & Hong (1977), whose goal was to create an efficient algorithm for three dimensions.

To keep the explanation simple, we assume two types of nondegeneracy: No three points are collinear, and no two points lie on a vertical line. The outline of their algorithm is as follows:

1. Sort the points by x coordinate.
2. Divide the points into two sets $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ containing the left $\lceil n/2 \rceil$ points, and $\mathcal{B}$ the right $\lfloor n/2 \rfloor$ points.
3. Compute the convex hulls $A = \mathcal{H}(\mathcal{A})$ and $B = \mathcal{H}(\mathcal{B})$ recursively.
4. Merge A and B: Compute conv $\{A \cup B\}$.

[22]This idea is due to Edelsbrunner (1987, p. 144).

The sorting step (1) guarantees the sets $\mathcal{A}$ and $\mathcal{B}$ will be separated by a vertical line (by our assumption that no two points lie on a vertical), which in turn guarantees that A and B will not overlap. This simplifies the merge step. Steps 2, 3, and 4 are repeated at each level of the recursion, stopping when $n \leq 3$; if $n = 3$ the hull is a triangle by our assumption of noncollinearity.

All of the work in this divide-and-conquer algorithm resides in the merge step. For this algorithm it is tricky to merge in linear time: The most naive algorithm would only achieve $O(n^2)$, which as we mentioned is not sufficient to yield $O(n \log n)$ performance overall.

We will use a and b as indices of vertices of A and B respectively, with the vertices of each ordered counterclockwise and numbered from 0. All index arithmetic is modulo the number of vertices in the relevant polygon, so that expressions like $a+1$ and $a-1$ can be interpreted as the next and previous vertices around A's boundary, respectively. The goal is to find two tangent lines, one supporting the two hulls from below, and one supporting from above. From these tangents, conv$\{A \cup B\}$ is easily constructed in $O(n)$ time. We will only discuss finding the lower tangent; the upper tangent can be found analogously.

Let the lower tangent be $T = ab$. The difficulty is that neither endpoint of T is known ahead of time, so a search has to move on both A and B. Note that the task in the incremental algorithm was considerably easier because only one end of the tangent was unknown; the other was a single, fixed point. Now a naive search for all possible a endpoints and all possible b endpoints would result in a worst-case quadratic merge step, which is inadequate. So the search must be more delicate.

The idea of Preparata and Hong is to start T connecting the rightmost point of A to the leftmost point of B, and then to "walk" it downwards, first advancing on one hull, then on the other, alternating until the lower tangent is reached. See Figure 3.11. Pseudocode is displayed in Algorithm 3.9. Note that $a-1$ is clockwise on A, and $b+1$ is counterclockwise on B, both representing downward movements. $T = ab$ is a lower tangent at a if both $a-1$ and $a+1$ lie above T; this is equivalent to saying that both of these points are left or on ab. A similar definition holds for lower tangency to B. And again we assume for simplicity of exposition that lines are always tangent at a point, rather than along an edge.

```
Algorithm: LOWER TANGENT
a ← rightmost point of A.
b ← leftmost point of B.
while T = ab not lower tangent to both A and B do
    while T not lower tangent to A do
        a ← a − 1
    while T not lower tangent to B do
        b ← b + 1
```

Algorithm 3.9 Lower tangent.

An example is shown in Figure 3.11. Initially, $T = (4, 7)$; note that T is tangent to both A and B, but it is only a lower tangent for A. The A loop does not execute, but the

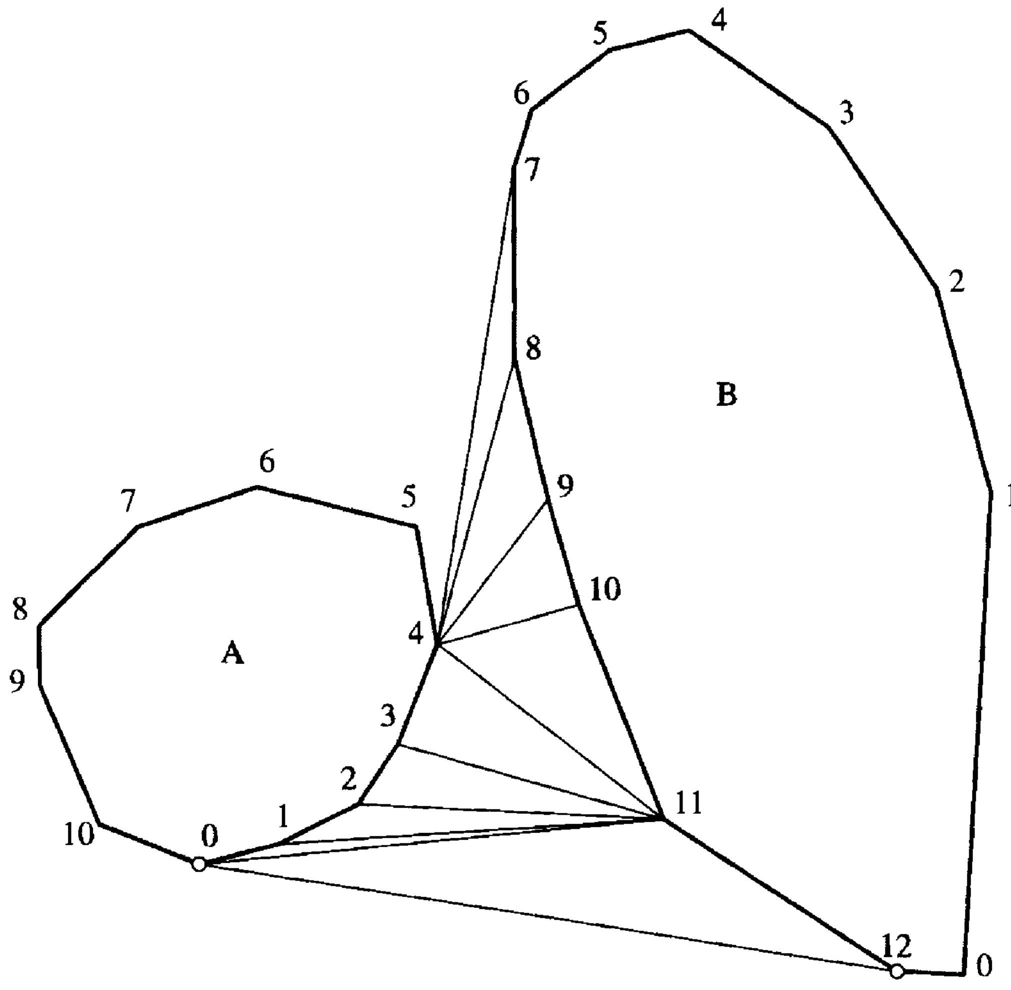

FIGURE 3.11 Finding the lower tangent: from (4, 7) to (0, 12).

B loop increments b to 11, at which time $T = (4, 11)$ is a lower tangent for B. But now T is no longer a lower tangent for A, so the A loop decrements a to 0; now $T = (0, 11)$ is a lower tangent for A. This is not a lower tangent for B, so b is incremented to 12. Now $T = (0, 12)$ is a lower tangent for both A and B, and the algorithm stops. Note that $b = 12$ is not the lowest vertex of B: 0 is slightly lower.

3.8.3. Analysis

Neither the time complexity nor the correctness of the hull-merging algorithm are evident. Certainly when the outermost while loop terminates, T is tangent to both A and B. But two issues remain:

1. The outer loop must terminate: It is conceivable that establishing tangency at a always breaks tangency at b and vice versa.
2. There are four mutual tangents (see Figure 3.12), and the algorithm as written should only find one, the one that supports both A and B to its left.

Lemma 3.8.1. *The lower tangent $T = ab$ touches both A and B on their lower halves: Both a and b lie on the closed chain from the leftmost vertex counterclockwise to the rightmost vertex, of A and B respectively.*

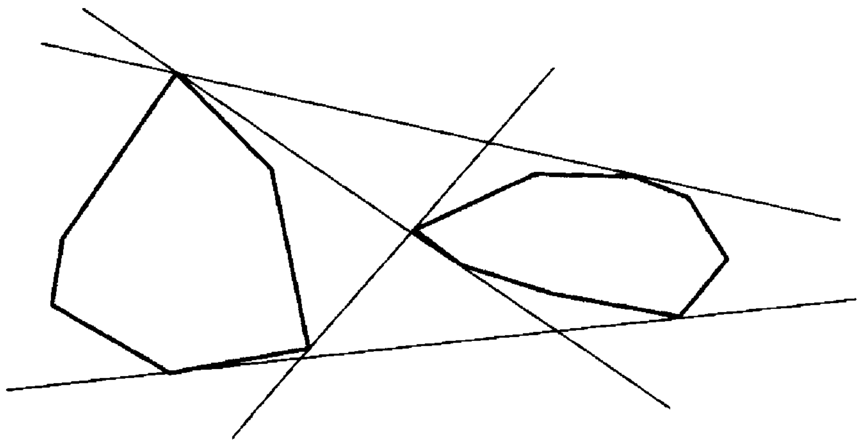

FIGURE 3.12 Four mutual tangents.

Proof. This is a consequence of the horizontal separation of A and B. Let L be a vertical line separating them. Then if B is very high above A, T approaches being parallel to L, and a approaches the rightmost vertex of A. Similarly if B is very low below A, a approaches the leftmost vertex of A. Between these extremes, a lies on the lower half of A. □

Since a starts at the rightmost vertex, and is only decremented (i.e., moved clockwise), the inner A loop could only iterate infinitely if a could pass the leftmost vertex. However, the next lemma will show this is not possible.

Lemma 3.8.2. *Throughout the life of Algorithm 3.9, ab does not meet the interior of A or of B.*

Proof. The proof is by induction. The statement is true at the start of the algorithm.

Suppose it is true after some step, and suppose that a is about to be decremented, which only happens when T is not a lower tangent to A. The new tangent $T' = (a-1, b)$ could only intersect the interior of A if b is left of $(a-1, a)$; see Figure 3.13. But b could not also be left of $(a, a+1)$, for then T would have intersected A, which we assumed

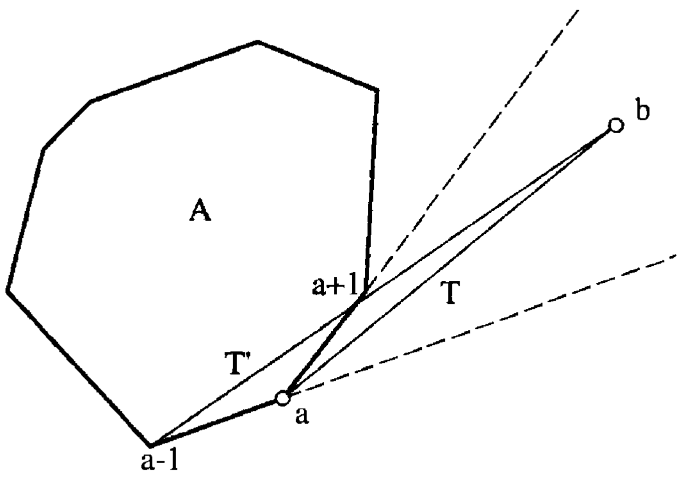

FIGURE 3.13 If T' intersects A, then T must be a lower tangent at a.

by induction was not the case. So if the next step would cause intersection, T must be tangent at a, and the next step would not be taken. □

Now because T does not meet the interior of A, and because b is right of L (the line separating A and B), a cannot advance (clockwise) beyond the leftmost vertex of A. Similar reasoning applies to B. Therefore both loops must terminate eventually, since they each move the indices in one direction only, and there is a limit to how far the indices can move. Now everything we need for correctness follows. Because the loops terminate, they must terminate with double tangency, which clearly must be the lower tangent.

The loops can only take linear time, since they only advance, never back up, and the number of steps is therefore limited by the number of vertices of A and B. Therefore the merge step is linear, which implies that the entire algorithm is $O(n \log n)$.

3.8.4. Output-Size Sensitive Optimal Algorithm

As a function of n, this asymptotic time complexity cannot be improved, in view of the lower bound (Section 3.6). But recall that the gift wrapping and QuickHull algorithms are output-size sensitive, running in time $O(nh)$ for hulls of size h. Examining the lower bound proof more closely shows it to establish $\Omega(n \log h)$. The divide-and-conquer algorithm does not match this more refined bound, which opened the possibility that it was not "the ultimate planar convex hull algorithm." Kirkpatrick & Seidel (1986) published a paper with this title (suffixed by '?'), seven years after the Preparata and Hong algorithm. They introduced a novel variation of the divide-and-conquer paradigm they called "marriage-before-conquest" that leads to an $O(n \log h)$ time complexity. The algorithm constructs the upper and lower hulls separately. Rather than conquering the subproblems after dividing, it finds an upper tangent of the two sets prior to finding their hulls recursively. Finding this "bridge" in linear time is tricky, but knowing it permits all the points vertically underneath the bridge and between its endpoints to be discarded before recursing. This partial discard is the key to achieving the lower asymptotic time complexity. However, it is questionable whether this theoretical improvement is worth the additional programming complexity in practical situations.

3.8.5. Exercises

1. *Recurrence relation with $O(n^2)$ merge.* Solve the recurrence relation $T(n) = 2T(n/2) + O(n^2)$.
2. *Recurrence relation with $O(n \log n)$ merge.* Solve the recurrence relation $T(n) = 2T(n/2) + O(n \log n)$.
3. *Degeneracies.* Remove all assumptions of nondegeneracy from the divide-and-conquer algorithm by considering the following possibilities:
   a. Several points lie on the same vertical line.
   b. A tangent line T is collinear with an edge of A and/or B.
   c. The recursion bottoms out with three collinear points.
4. *Merge without sorting* (Toussaint 1986). If the sorting step of the divide-and-conquer algorithm is skipped, the hulls to be merged might intersect. Design an algorithm that can merge two arbitrarily located hulls of n and m vertices in $O(n+m)$ time. This then provides an alternative $O(n \log n)$ divide-and-conquer algorithm.