CS584/684 Algorithm Analysis and Design – Spring 2017
Week 2: Computational Geometry

[Much of the material in this lecture is from Preparata and Shamos, *Computational Geometry: An Introduction*, Springer 1985.]

We'll study a collection of geometric algorithms that are both interesting in themselves and offer a chance to review standard sorting and searching algorithms. All problems will be in 2-D; in many cases, there are higher-dimensional analogs, but these are sometimes much harder. So *point* means a point on the plane given by a pair of real coordinates (x, y).

Several of these problems have brute-force $O(n^2)$ solutions, and we will be happy to find $O(n \log n)$ solutions for them instead. It is worth remembering why this is such a big difference. The following table may help:

n	n^2	$n\ log_{10}\ n$
10	100	10
100	$10,000$	200
$1,000$	$1,000,000$	$3,000$
$1,000,000$	$1,000,000,000,000$	$6,000,000$
$1,000,000,000$	$1,000,000,000,000,000,000$	$9,000,000,000$

1 Closest Pair

"Given a collection of n points, find the pair that are closest together."

Here distance is measured by the usual euclidean metric: given points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, the distance between them is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

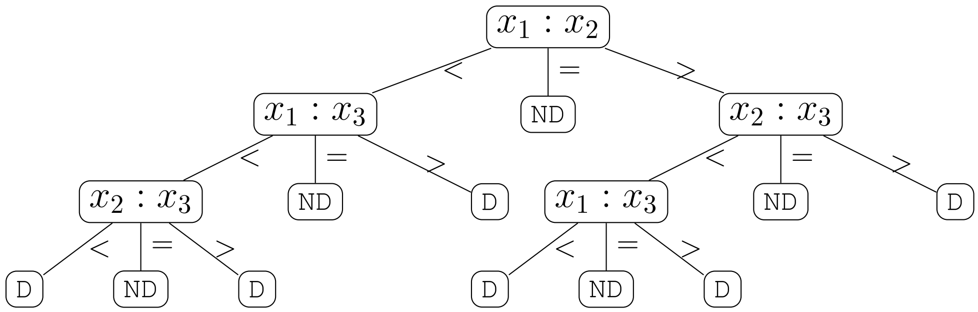
The obvious brute force solution is to compute the distance between every possible pair of points, keeping track of the minimum; this is clearly $O(n^2)$.

We can give a lower bound of $\Omega(n \log n)$ for this problem by *reduction*, as follows.

Excursion: Element Uniqueness

First consider another problem, *Element Uniqueness*: "Given a collection of n real numbers, determine whether or not they are all distinct." We can show that this problem requires $\Omega(n \log n)$ operations under a comparison decision tree model of computation. In this model, we abstract away all the control flow and other details of an algorithm, and just consider the sequence of comparisons made between the inputs. (You have probably seen this technique used to give a lower bound on comparison-based sorting.)

The internal nodes of the tree represent comparison operations between pairs of particular input elements; for simplicity, we'll use three-way comparisons, so an internal node labeled $a : b$ has three children corresponding to the cases $a < b$, $a = b$, and $a > b$. The leaves represent possible outcomes; in this case, we label them "Distinct" or "Not Distinct." Every particular set of inputs induces a path through the tree from the root to a leaf. For example, here is a possible decision tree for checking Element Uniqueness on three elements x_1, x_2, x_3:

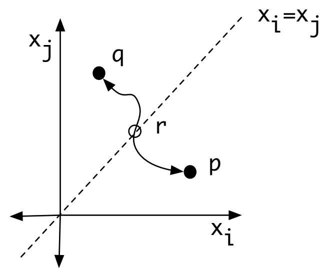


Notice that for some inputs, the tree does not have compare every element with every other element to reach a leaf. For example, for the input $x_1 = 3, x_2 = 2, x_3 = 1$, the tree does not need to compare x_1 directly with x_3.

So now consider decision trees for Element Uniqueness problems of size n acting on an array of inputs $x_1, x_2, \ldots, x_n$. To give a lower bound on the (worst-case) number of comparisons needed to solve Element Uniqueness, it suffices to give a lower bound on the height h of any possible decision tree. In turn, we will derive this from a lower bound on the number of leaves l in the tree.

By a coincidence, the argument is actually geometric!

- Consider an arbitrary decision tree (for any problem, not necessarily Element Uniqueness). View each possible input as a point in the n-dimensional space $\mathbb{R}^n$. Then each node in the decision tree defines a corresponding *hyperplane*, and the outcome of the node's test classifies the input as lying on one side or the other of that hyperplane or on the hyperplane itself. An input reaches a leaf of the tree if it passes all the classification tests of the nodes along the path from the root. Write $W(v)$ for the set of input points that reaches a leaf v. Then $W(v)$ is always a *convex* region of $\mathbb{R}^n$, and hence *path-connected*, meaning that between any two points in $W(v)$ there is a continuous path that remains in $W(v)$.
- Now consider an arbitrary decision tree for Element Uniqueness, and consider an input consisting of n distinct numbers, with $x_1 < x_2 < x_3 < \ldots < x_n$. Then certainly this input should be in $W(v)$ for some leaf v marked "Distinct." Moreover, every *permutation* of the input should also be in $W(v')$ for some leaf v' marked "Distinct." We claim that all of these leaves must be *different* from one other. Why? Suppose, to reach a contradiction, that there are two points p, q, corresponding to two different permutations, with $p, q \in W(v)$ for some single leaf v. Note that each permutation differs from every other permutation by swapping the order of at least two values, i.e. there exist i, j such that $x_i < x_j$ in p but $x_j < x_i$ in q. Then any continuous path between p and q must pass through the hyperplane $x_i = x_j$ at some point r, i.e. some input for which $x_i = x_j$. (The diagram below shows the projections onto the (x_i, x_j) plane.) By (1), $r \in W(v)$, so it will produce the answer "Distinct." But obviously, r should produce the answer "Not Distinct." So we have a contradition, and all the $n!$ permutations of the input must correspond to distinct leaves.



- Since a ternary tree of height h can have at most 3^h leaves (why?), we have $n! \leq l \leq 3^h$. Taking logs, we get $h \geq \log_3(n!)$.
- Via Stirling's approximation for $n!$, we know $\log(n!) = \Theta(n \log n)$. So $h = \Omega(n \log n)$, and we have our lower bound.

Warning: in all lower bound arguments, we have to be very careful about our model of computation. For example, nothing we have shown here precludes a $o(n)$ algorithm for Element Uniqueness based on something other than comparisons, such as bin sorting.

[End of Excursion]

We can *reduce* Element Uniqueness to Closest Pair as follows: associate each element x_i with the point $p_i = (x_i, 0)$. Now the x_i are unique iff the closest pair of points p_i, p_j are a non-zero distance apart. So we cannot solve Closest Pair in better than $\Omega(n \log n)$ time, because if we could, then we could also solve Element Uniqueness in better than $\Omega(n \log n)$ time. This establishes the lower bound.

So now, let's try to find an algorithm that actually achieves this lower bound. Can we just reduce Closest Pair to sorting ($O(n \log n)$) by projecting onto a line, e.g. the x-axis? No: we lose too much information from the other coordinate.

Picture from Preparata&Shamos:

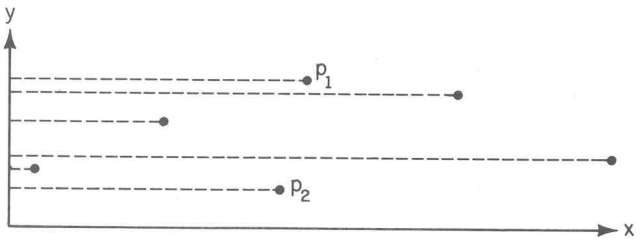


Figure 5.8 The failure of projection methods. Points p_1 and p_2 are the closest pair but are farthest in y-distance.

So let's try to find a divide-and-conquer algorithm that *will* scale up to 2-D. Still, to get some intuition, we can start in the 1-D case, by supposing that all our points are on the x-axis. But our algorithm won't sort. Instead, it performs the following steps:

1. Find the median m of the x coordinates, and partition the points into two groups L and R containing points less than, resp. greater than, m.
2. Recursively find the the closest pair within L; let δ_L be the distance between that pair. Do the same with the R, calculating δ_R. Let $\delta = min(\delta_L, \delta_R)$.
3. Now it may be that the algorithm has already found the closest pair in the whole group, with distance δ. But it remains possible that the closest pair consists of one element x_L in L and one element x_R in R, with $x_R - x_L < \delta$. A key observation is that, to find such a pair, it suffices to compare points in $L' = L \cap (m - \delta, m]$ and $R' = R \cap [m, m + \delta)$; any other points would be too far apart. L' and R' can be extracted by a linear scan through each set. Then it suffices simply to compare each element of L' with each element of R'. If the closest pair among these is closer than δ it is returned; otherwise the algorithm returns the closest pair from L or R, whichever produced δ.

Picture from Preparata&Shamos:

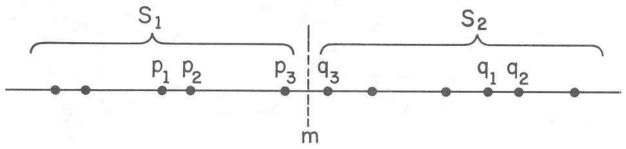


Figure 5.9 Divide-and-conquer in one dimension.

What's the running time of this algorithm? We know step 1 can be done in $O(n)$ time. Step 2 recurses on two problems of size $n/2$. So if step 3 can also be done in $O(n)$ time, we'll have $T(n) = 2T(n/2) + \Theta(n)$, which we know leads to $T(n) = O(n \log n)$, matching the lower bound. But in step 3 the algorithm must compare every element of L' with every element of R' requiring $\Omega(|L'| \cdot |R'|)$ time. So the big question is: how big can L' and R' be?

The second key observation is that L' and R' are very small indeed: in fact, each can contain at most *one* element! Why? Suppose there were two or more elements in, say, L'. Then the distance between them would be less than δ. But by definition, δ is less than or equal to the smallest inter-point distance in L', so we have a contradiction. (The same argument applies to R'.) So the number of comparisons in step 3 is just a constant (1), and the step can indeed be performed in $O(n)$ time.

(In fact, since we are on a line, step 3 can obviously be peformed by comparing the maximum element of L and the minimum element of R, which could also easily be extracted by a linear scan. But our goal here was to find an approach that will scale to 2-D, where "maximum" and "minimum" don't make sense.)

Now let's lift this approach to handle points in two dimensions. The algorithm now looks like this:

1. Take the median m of the x-coordinates and let l be the vertical line $x = m$. Divide the points into sets L and R according to whether they lie to the left or right of l.
2. As before, recursively calculate the closest pairs in L and R, producing δ_L and δ_R, and let $\delta = min(\delta_L, \delta_R)$.
3. As before, we observe that if there is closer pair of points $p_L = (x_L, y_L)$ and $p_R = (x_r, y_R)$ with p_L in L and p_R in R, then we must have $x_L \in (m - \delta, m]$ and $y_L \in [m, m + \delta)$. Graphically, this means the points lie within strips of width δ to the left and right of l. Defining L' to be the points in the left strip and R' to be the points in the right strip, it once again suffices to extract these sets by a linear scan and then compare each point of L' with each point of R'.

However, in 2-D, it is no longer the case that L' and R' have at most one element each. In fact, all n points can lie within the strips!

Picture from Preparata&Shamos:

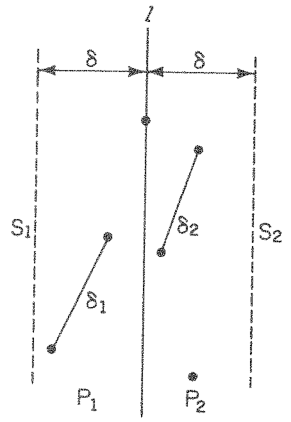


Figure 5.11 All points may lie within δ of l.

To avoid paying $O(n^2)$ time in step 3, the algorithm need to do more. First, it sorts L' and R' by y-coordinate. (This would seem to require $O(n \log n)$ time, which would also break our desired bound, but we'll see a way around that problem shortly.) Next it performs a specialized "merge" of the sorted L' and R', processing the points in L' in increasing y order. The key 2-D observations are that: (a) for any candidate point $p_L \in L'$, it suffices to consider candidate points $p_R \in R'$ such that $y_R \in (y_L - \delta, y_L + \delta)$, since otherwise the points would be at least δ apart; and (b) there can be at most six points in the rectangle bounded by the lines l, $x = l + \delta$, $y = y_L - \delta$, $y = y_L + \delta$, since otherwise at least two of them would be closer than δ. Based on these facts, only six candidate R' points must be checked for each L' point, and they can be found in constant time. (This last point is not completely obvious, but if you think of maintaining a pointer into a sorted R' array while advancing through the L' array, you'll see that the R' pointer will need to oscillate up and down a little, but never by more than six elements. CLRS gives a slightly different algorithm in which L' and R' are combined before sorting; in this scheme, each element in the list need only be compared with the five succeeding elements.)

Picture from Preparata&Shamos:

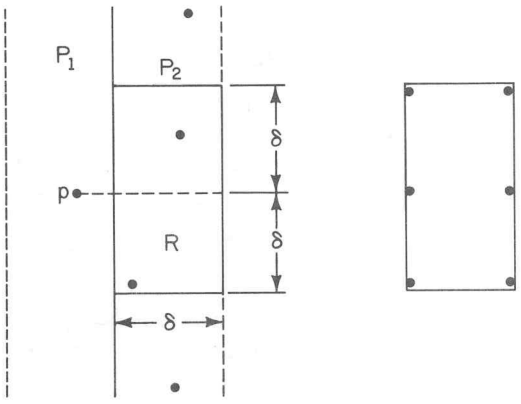


Figure 5.12 For every point in P_1 only a constant number of points in P_2 need to be examined (at most six).

Finally, we need to address the sorting of the L' and R' sets. As noted above, if we sorted them from scratch on each recursive call, we would end up with $O(n \log n)$ time at each recursive level, giving $O(n \log^2 n)$ time overall rather than our desired $O(n \log n)$. The easiest way to fix this is to *presort* the entire array of n

points by y-coordinate before we start the algorithm, at a one-time cost of $O(n \log n)$. Then in step (3), we just *extract* the relevant elements from the sorted list, maintaining the sort order, in linear time.

2 Segment Intersections

"Given a collection of line segments in the plane, determine if any of them intersect."

(Here, we include intersections at endpoints.)

As usual, there is a trivial $O(n^2)$ brute force solution.

And again, it is useful to start with the analogous 1-D problem:

"Given a collection of intervals on the real line, do any of them overlap?"

This has an easy solution: label each of the $2n$ interval endpoints as either L(eft) or R(ight) and then sort them. (If there are duplicate values, put all L-labeled values before any R-labeled values.) The intervals are disjoint iff the labels of the sorted sequence alternate L,R,L,R,. . .,L,R. This algorithm costs $O(n \log n)$ for the sort and $O(n)$ to scan the labels, so $O(n \log n)$ overall.

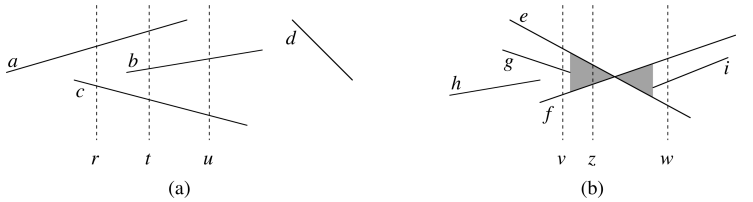
There is also a $\Omega(n \log n)$ lower bound on the 1-D problem, again by reduction from Element Uniqueness: represent each element x_i by the degenerate interval $[x_i, x_i]$; then the elements are disjoint iff the intervals do not overlap. Note that this 1-D lower bound carries over into 2-D as well, since the 1-D problem is just a special case of the 2D one.

Once again, we aim for a 2-D solution with $O(n \log n)$ time by trying to generalize the 1-D solution. One way to conceptualize the 1-D solution is as *sweeping* from left to right along the number line, keeping a *status*: a single integer representing the number of intervals that overlap the "current" value. There is an overlap iff the count ever exceeds 1. The status count changes by +1 or -1 at *event points*: the interval endpoints.

We now sketch a 2-D algorithm. For simplicity, we assume that no three segments share an endpoint, and that there are no vertical segments; these restrictions can be lifted with more or less trouble. We will carry over to the 2-D algorithm the ideas of sweeping while maintaining a status that changes at event points. In the 2-D version:

- We sweep a vertical *line* across the plane from left to right.
- The sweep status data structure keeps track of the segments that intersect the current sweep line, sorted by the y-coordinate of their intersections with the sweep line. We need the ability to insert and delete segments at the correct position and to find the segments just above or below a given segment in the order. A balanced binary search tree, such as a Red-Black tree, will do the job nicely, and provide each of these operations in $O(\log n)$ time. See CLRS Chs. 12-13.
- Sweep event points are still the (x-coordinates of) segment endpoints. An L endpoint event adds the segment to the sweep status data structure; an R endpoint event removes the segment. (The status list should also change at the x-coordinate of any segment intersection, at which point segments swap their position in the list. But as we'll see, our sweep line never gets that far, as long as we're just checking to see whether *some* intersection occurs.)

A picture (CLRS Figure 33.4) may help:



The key geometric observation behind the algorithm is that two segments can only possibly intersect only if they become adjacent in the sweep status ordering at some point. Otherwise, they would always be separated by some other segment (and remember that we are not allowing three segments to intersect at a common point). So to detect segment intersections, it suffices to check each pair of adjacent segments for intersection at the first sweep event where they become adjacent, which is either an L or R endpoint. More specifically:

- When we insert a segment into the sweep data structure at its L endpoint, we check to see whether it intersects either of its immediate neighbors in the order; if so, we immediately end the algorithm with the answer "yes."
- When we remove a segment from the sweep data structure at its R endpoint, we check to see whether its immediate neighbors intersect each other; if so, we immediately end the algorithm with the answer "yes."

There is no need to check for adjacencies that arise as the result of order swapping at an intersection point: we will already have detected the intersection and exited from the algorithm *before* the sweep line passes beyond the intersection point.

[The full algorithm is given on CLRS p. 1025.]

The initial sort takes $O(n \log n)$ time. The scan encounters $2n$ events (corresponding to endpoints), and the action at each event takes $O(\log n)$ time provided that the sweep status data structure steps above can be done in $O(\log n)$ time. So the algorithm is $O(n \log n)$ overall.

3 Convex Hulls

The *convex hull* of a point set S is the smallest convex polygon containing all the points of S. We normally are interested in the presentation of a convex hull as the *ordered* sequence of the vertices of that polygon.

Some key facts:

- Point $p \in S$ is not a vertex of the convex hull of S iff p lies on or in a triangle whose vertices are in $S - \{p\}$.

  Picture from Preparata&Shamos:

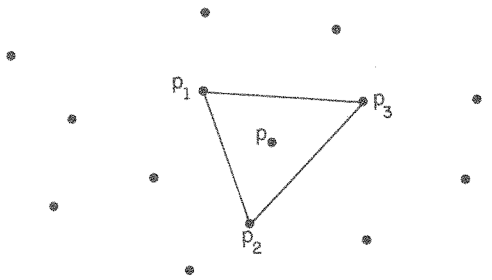


Figure 3.3 Point p is not extreme because it lies inside triangle $(p_1 p_2 p_3)$.

- Consecutive vertices of a convex polygon occur in sorted angular order about any interior point.

  Picture from Preparata&Shamos:

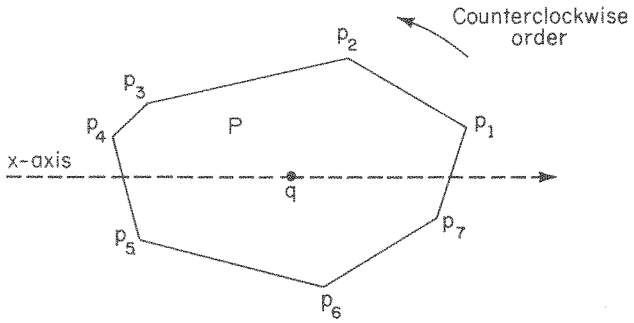


Figure 3.4 The vertices of P occur in sorted order about q.

- The *centroid* (i.e. the arithmetic mean) of the vertices of a convex polygon lies in the interior of the polygon.

The convex hull is a basic tool for many applications. We'll just mention that, among other things, it gives us a way of finding the maximum distance between any two points in a point set, because this distance is realized between two points on the hull.

We can give a brute force method for computing the convex hull (i.e, its vertices, in order) of a point set S of size n.

1. We can determine the (unordered) set of hull vertices of S by repeated use of the triangle test. Since there are $\Omega(n^3)$ triangles and n points to try, this requires $\Omega(n^4)$ time.
2. Now we need to sort the hull vertices into consecutive order. We can do this by sorting by angular order about an interior point. We can find an interior point in $O(n)$ time by taking the centroid of the (unsorted) vertices. Then we can sort using an $O(n \log n)$ method. (There is no need to compute the angles explicitly by conversion to polar coordinates: we just need to know the *relative* angles of pairs of rays, for which testing the cross product suffices.)

This gives an $O(n^4)$ algorithm.

We can give a lower bound on convex hull calculation by reduction from sorting. Given a set of positive values $x_1, \ldots, x_n$, consider the corresponding points (x_i, x_i^2). These lie on the parabola $y = x^2$. The convex

hull of this set in standard consecutive order will be a list of the points sorted by x-coordinate, so we can simply read off the sorted values. Since we have an $\Omega(n \log n)$ lower bound for comparison-based sorting, we get the same lower bound for convex hull.

Picture from Preparata&Shamos:

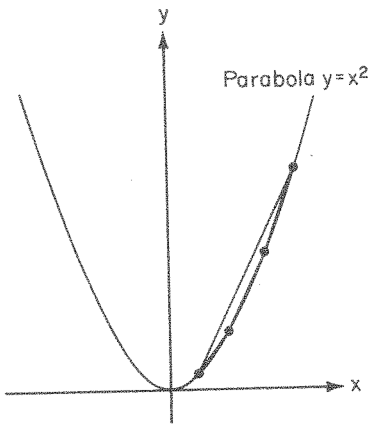


Figure 3.1 Illustration for the proof of Theorem 3.2.

Fortunately, there are plenty of ways to meet the lower bound. The simplest is probably Graham's Scan (1972).

1. Find a known hull vertex point p (e.g. the one with minimum y-value and then minimum x-value in case of ties). This takes $O(n)$ time.

2. Sort all the points in angular order about p in $O(n \log n)$ time, again using any comparison-based worst-case $O(n \log n)$ method. (If two points are on the same ray from p, throw away all but the furthest away.)

3. Process the sorted points in increasing angular order, in consecutive triples p_1, p_2, p_3. If $\angle p_1 p_2 p_3 < \pi$ (is a "right-hand turn"), point p_2 is in the interior of the triangle pp_1p_3, so it cannot be part of the hull: discard it and consider $p_0p_1p_2$. Otherwise, advance to consider $p_2p_3p_4$. Continue until the last point is reached.

   Picture from Preparata&Shamos:

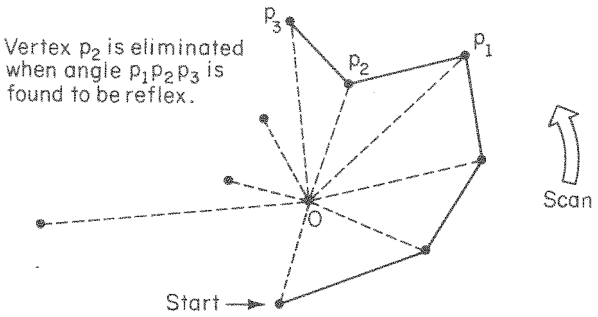


Figure 3.6 Beginning the Graham Scan.

The time analysis for this step is slightly complicated. The algorithm can only perform n advancing steps, but any one advance step may be followed by up to n discard steps in the worst case. Thus a naive analysis would give $O(n^2)$ for the whole scan. The key observation is that the *total* number of discard steps is also bounded by n, so the real bound for the whole scan is $O(2n)$. This is a simple example of *amortized analysis* (specifically what CLRS calls *aggregate analysis*).

The overall performance of Graham's scan is thus dominated by the sorting work in step 2, and is worst-case $O(n \log n)$, matching the lower bound.

Note that the angular sorting in step 2 can be done about an interior point (such as the point O shown in the picture) rather than an hull vertex point. The only requirement on the ordering is that it supports the use of the "right-hand turn" test to discover and reject non-hull-vertex points. In fact, we can also simply sort the points by increasing x-coordinate, which is equivalent to sorting by angle about a point notionally at $y = \infty$. Note the the points with minimum and maximum x-coordinates must be on the convex hull, and they partition it into upper and lower *chains*. Performing step 3 of the Graham scan on the points from left-to-right produces the lower chain, and performing it from right-to-left (equivalent to the order produced by sorting about a point notionally at $y = -\infty$) produces the upper chain. Concatenating the two chains gives the entire hull.

The connection between convex hull computation and sorting is really quite close. So it is natural to ask if other sorting methods have complex hull analogs—and indeed they do. Here is one based on MergeSort, which we'll call MERGEHULL:

1. Divide the point set arbitrary into two equal-size subsets.
2. Recursively compute the convex hull of each subset.
3. Combine the two resulting subset hulls to obtain the overall hull.

As usual, this gives an $O(n \log n)$ algorithm if the combination step can be done in $O(n)$ time. That is, we are given two convex polygons P_1 and P_2 with a combined total of n vertices, and wish to compute the convex hull of their union in $O(n)$ time. We now sketch how this is possible (Shamos, 1978).

1. Pick a point p in the interior of P_1 (the centroid, which can be computed in $O(n)$ time, will do nicely).
2. Determine whether p is also in the interior of P_2. This can be done in $O(n)$ time, by, for example, checking whether p is on the same side of each edge taken in turn.
3. If p is internal to P_2, then the vertices of P_1 and of P_2 are each in sorted angular order about p. We may thus merge them in $O(n)$ time to obtain a single list of the vertices of both P_1 and P_2 sorted about p.

   Picture from Preparata&Shamos:

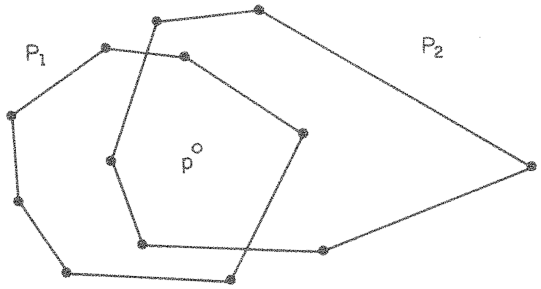


Figure 3.12 Point p lies inside P_2. Since p lies inside both polygons, the vertices of P_1 and P_2 occur in sorted order about p and can be merged in linear time.

4. Alternatively, suppose p is *not internal* to P_2. Now consider the polar angles of the rays from p to each vertex of P_2, and let u and v be the vertices of P_2 for which the angles are minimal and maximal. (Then P_2 lies in $\angle upv$.) u and v partition the vertices of P_2 into two chains that are monotonic in polar angle about p, one increasing and the other decreasing. Of these two, the one convex towards p can be immediately discarded, since its vertices are internal to the convex hull of the union. The other chain of P_2 and the vertices of P_1 are each sorted in angular order about p. Again, they can be merged in $O(n)$ time to form a single list sorted about p.

   Picture from Preparata&Shamos:

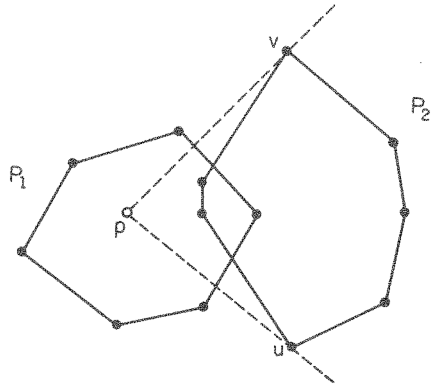


Figure 3.13 Point p is external to P_2. As seen from p, polygon P_2 lies in a wedge defined by vertices u and v, which partition P_2 into two chains of vertices. One may be discarded, the other merged with the vertices of P_1 in linear time.

5. The Graham scan can now be applied to the list obtained from step 3 or 4 to produce the hull in $O(n)$ time.