
Elementary Geometric Methods

☐ Computers are being used more and more to solve large-scale problems that are inherently geometric. Geometric objects such as points, lines and polygons are the basis of a broad variety of important applications and give rise to an interesting set of problems and algorithms.

Geometric algorithms are important in design and analysis systems modeling physical objects ranging from buildings and automobiles to very large-scale integrated circuits. A designer working with a physical object has a geometric intuition that is difficult to support in a computer representation. Many other applications directly involve processing geometric data. For example, a political “gerrymandering” scheme to divide a district up into areas of equal population (and that satisfy other criteria such as putting most of the members of the other party in one area) is a sophisticated geometric algorithm. Other applications abound in mathematics and statistics, fields in which many types of problems can be naturally set in a geometric representation.

Most of the algorithms we’ve studied have involved text and numbers, which are represented and processed naturally in most programming environments. Indeed, the primitive operations required are implemented in the hardware of most computer systems. We’ll see that the situation is different for geometric problems: even the most elementary operations on points and lines can be computationally challenging.

Geometric problems are easy to visualize, but that can be a liability. Many problems that can be solved instantly by a person looking at a piece of paper (example: is a given point inside a given polygon?) require non-trivial computer programs. For more complicated problems, as in many other applications, the method of solution appropriate for computer implementation may well be quite different from the method of solution appropriate for a person.

One might suspect that geometric algorithms would have a long history because of the constructive nature of ancient geometry and because useful applications are

so widespread, but actually much of the work in the field has been quite recent. Nonetheless, the work of ancient mathematicians is often useful in the development of algorithms for modern computers. The field of geometric algorithms is interesting to study because of its strong historical context, because new fundamental algorithms are still being developed, and because many important large-scale applications require these algorithms.

Points, Lines, and Polygons

Most of the programs we'll study operate on simple geometric objects defined in a two-dimensional space, though we will consider a few algorithms for higher dimensions. The fundamental object is a *point*, which we consider to be a pair of integers—the “coordinates” of the point in the usual Cartesian system. A *line* is a pair of points, which we assume are connected together by a straight line segment. A *polygon* is a list of points: we assume that successive points are connected by lines and that the first point is connected to the last to make a closed figure.

To work with these geometric objects, we need to decide how to represent them. Usually we use an array representation for polygons, though a linked list or some other representation can be used when appropriate. Most of our programs will use the straightforward representations

```
struct point { int x, y; char c; };  
struct line { struct point p1, p2; };  
struct point polygon[Nmax];
```

Note that points are restricted to have integer coordinates. A floating point representation could also be used. Using integer coordinates leads to slightly simpler and more efficient algorithms, and is not as severe a restriction as it might seem. As mentioned in Chapter 2, working with integers when possible can be a very significant timesaver in many computing environments, because integer calculations are typically much more efficient than floating-point calculations. Thus, when we can get by with dealing only with integers without introducing much extra complication, we will do so.

More complicated geometric objects will be represented in terms of these basic components. For example, polygons will be represented as arrays of points. Note that using arrays of lines would result in each point on the polygon being included twice (though that still might be the natural representation for some algorithms). Also, it is useful in some applications to include extra information associated with each point or line; we can do this by adding an *info* field to the records.

We'll use the sets of points shown in Figure 24.1 to illustrate the operation of several geometric algorithms. The sixteen points on the left are labeled with single letters for reference in explaining the examples, and have the integer coordinates

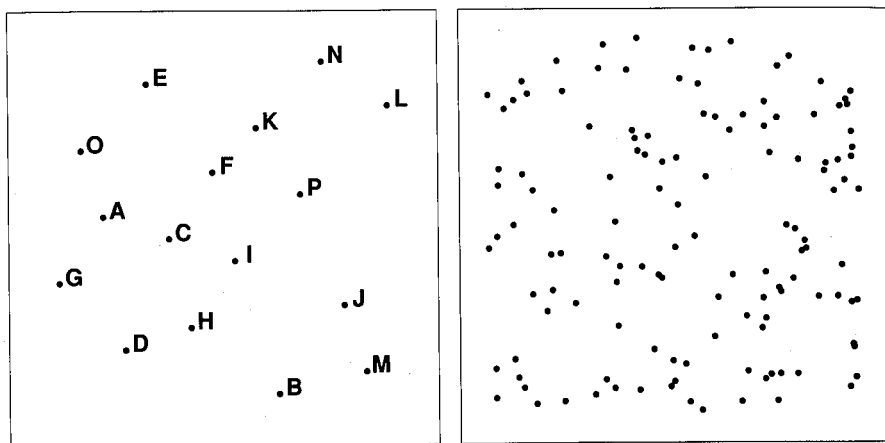


Figure 24.1 Sample point sets for geometric algorithms.

shown in Figure 24.2. (The labels we use are assigned in the order in which the points are assumed to appear in the input.) The programs usually have no reason to refer to points “by name”; they are simply stored in an array and are referred to by index. The order in which the points appear in the array may be important in some of the programs: indeed, it is the goal of some geometric algorithms to “sort” the points into some particular order. On the right in Figure 24.1 are 128 points, randomly generated with integer coordinates between 0 and 1000.

A typical program maintains an array p of points and simply reads in N pairs of integers, assigning the first pair to the x and y coordinates of $p[1]$, the second pair to $p[2]$, etc. When p represents a polygon, it is sometimes convenient to maintain “sentinel” values $p[0]=p[N]$ and $p[N+1]=p[1]$.

Line Segment Intersection

As our first elementary geometric problem, we’ll consider determining whether or not two given line segments intersect. Figure 24.3 illustrates some of the situations that can arise. In the first case, the line segments intersect. In the second, the endpoint of one segment is on the other segment: We’ll consider this an intersection

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
x	3	11	6	4	5	8	1	7	9	14	10	16	15	13	3	12
y	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12	10

Figure 24.2 Coordinates of points in small sample set (on the left in Figure 24.1).

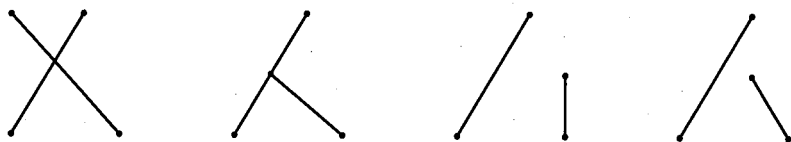


Figure 24.3 Testing whether line segments intersect: four cases.

by assuming the segments to be “closed” (endpoints are part of the segments); thus, line segments having a common endpoint intersect. In both the last two cases in Figure 24.3, the segments do not intersect, but the cases differ when we consider the intersection point of the lines defined by the segments. In the fourth case this intersection point falls on one of the segments; in the third it does not. Or, the lines could be parallel (a special case of this that frequently turns up is when one or both of the segments is a single point).

The straightforward way to solve this problem is to find the intersection point of the lines defined by the line segments and then check whether this intersection point falls between the endpoints of both of the segments. Another easy method is based on a tool that we’ll find useful later, so we’ll consider it in more detail. Given three points, we want to know whether, in traveling from the first to the second to the third, we turn counterclockwise or clockwise. For example, for points A, B, and C in Figure 24.1 the answer is yes, but for points A, B, and D the answer is no. This function is straightforward to compute from the equations for the lines as follows:

```
int ccw(struct point p0,
        struct point p1,
        struct point p2 )
{
    int dx1, dx2, dy1, dy2;
    dx1 = p1.x - p0.x; dy1 = p1.y - p0.y;
    dx2 = p2.x - p0.x; dy2 = p2.y - p0.y;
    if (dx1*dy2 > dy1*dx2) return +1;
    if (dx1*dy2 < dy1*dx2) return -1;
    if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
    if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2))
        return +1;
    return 0;
}
```

To understand how the program works, first suppose that all of the quantities $dx1$, $dx2$, $dy1$, and $dy2$ are positive. Then note that the slope of the line connecting $p0$ and $p1$ is $dy1/dx1$ and the slope of the line connecting $p0$ and $p2$ is $dy2/dx2$. Now, if the slope of the second line is greater than the slope of the first, a “left” (counterclockwise) turn is required in the journey from $p0$ to $p1$ to $p2$; if less, a “right” (clockwise) turn is required. Comparing slopes in the program is slightly inconvenient because the lines could be vertical ($dx1$ or $dx2$ could be 0): we multiply by $dx1 * dx2$ to avoid this. It turns out that the slopes need not be positive for this test to work properly—checking this is left as an instructive exercise for the reader.

But there is another crucial omission in the description above: it ignores all cases when the slopes are the same (the three points are collinear). In these situations, one can envision a variety of ways to define *ccw*. Our choice is to make the function three-valued: rather than the standard nonzero or zero return value we use 1 and -1, reserving the value 0 for the case where $p2$ is *on* the line segment *between* $p0$ and $p1$. If the points are collinear, and $p0$ is between $p2$ and $p1$, we take *ccw* to be -1; if $p2$ is between $p0$ and $p1$, we take *ccw* to be 0; and if $p1$ is between $p0$ and $p2$, we take *ccw* to be 1. We’ll see that this convention simplifies the coding for functions that use *ccw* in this and the next chapter.

This immediately gives an implementation of the *intersect* function. If both endpoints of each line are on different “sides” (have different *ccw* values) of the other, then the lines must intersect:

```
int intersect(struct line l1, struct line l2)
{
    return ((ccw(l1.p1, l1.p2, l2.p1)
            *ccw(l1.p1, l1.p2, l2.p2)) <= 0)
        && ((ccw(l2.p1, l2.p2, l1.p1)
            *ccw(l2.p1, l2.p2, l1.p2)) <= 0);
}
```

This solution seems to involve a fair amount of computation for such a simple problem. The reader is encouraged to try to find a simpler solution, but should be warned to be sure that the solution works on all cases. For example, if all four points are collinear, there are six different cases (not counting situations where points coincide), only four of which are intersections. Special cases like this are the bane of geometric algorithms: they cannot be avoided, but we can lessen their impact with primitives like *ccw*.

If many lines are involved, the situation becomes much more complicated. In Chapter 27, we’ll see a sophisticated algorithm for determining whether any two of a set of N lines intersect.

Simple Closed Path

To get the flavor of problems dealing with sets of points, let's consider the problem of finding a path through a set of N given points that doesn't intersect itself, visits all the points, and returns to the point at which it started. Such a path is called a *simple closed path*. One can imagine many applications for this: the points might represent homes and the path the route that a mailman might take to get to each of the homes without crossing his path. Or we might simply want a reasonable way to draw the points using a mechanical plotter. This problem is elementary because it asks only for any closed path connecting the points. The problem of finding the best such path, called the *traveling salesman problem*, is much, much more difficult, and we'll look at it in some detail in the last few chapters of this book. In the next chapter, we'll consider a related but much easier problem: finding the shortest path that surrounds a set of N given points. In Chapter 31, we'll see how to find the best way to "connect" a set of points.

An easy way to solve the elementary problem at hand is the following. Pick one of the points to serve as an "anchor." Then compute the angle made by drawing a line from each of the points in the set to the anchor and then out in the positive horizontal direction (this is part of the polar coordinate of each point with the anchor point as origin). Next, sort the points according to that angle. Finally, connect adjacent points. The result is a simple closed path connecting the points, as shown in Figure 24.4 for the points in Figure 24.1. In the small set of points, B is used as the anchor: if the points are visited in the order

B M J L N P K F I E C O A H G D B

then a simple closed polygon will be traced out.

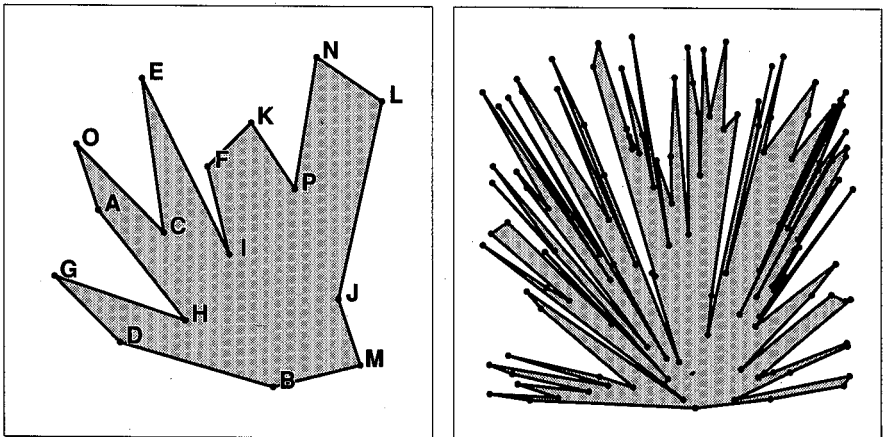


Figure 24.4 Simple closed paths.

If dx and dy are the distances along the x and y axes from the anchor point to some other point, then the angle needed in this algorithm is $\tan^{-1} dy/dx$. Although the arctangent is a built-in function in C (and some other programming environments), it is likely to be slow and leads to at least two annoying extra conditions to compute: whether dx is zero and which quadrant the point is in. Since the angle is used only for the sort in this algorithm, it makes sense to use a function that is much easier to compute but has the same ordering properties as the arctangent (so that when we sort, we get the same result). A good candidate for such a function is simply $dy/(dy + dx)$. Testing for exceptional conditions is still necessary, but simpler. The following program returns a number between 0 and 360 that is *not* the angle made by $p1$ and $p2$ with the horizontal but which has the same order properties as that angle.

```
float theta(struct point p1, struct point p2)
{
    int dx, dy, ax, ay;
    float t;
    dx = p2.x - p1.x; ax = abs(dx);
    dy = p2.y - p1.y; ay = abs(dy);
    t = (ax+ay == 0) ? 0 : (float) dy/(ax+ay);
    if (dx < 0) t = 2-t; else if (dy < 0) t = 4+t;
    return t*90.0;
}
```

In some programming environments it may not be worthwhile to use such programs instead of standard trigonometric functions; in others it may lead to significant savings. (In some cases it may be worthwhile to change `theta` to have an integer value, to avoid using floating point numbers entirely.)

Inclusion in a Polygon

The next problem we'll consider is a natural one: given a point and polygon represented as an array of points, determine whether the point is inside or outside the polygon. A straightforward solution to this problem immediately suggests itself: draw a long line segment from the point in any direction (long enough so that its other endpoint is guaranteed to be outside the polygon) and count the number of lines from the polygon that it crosses. If the number is odd, the point must be inside; if it is even, the point is outside. This is easily seen by tracing what happens as we come in from the endpoint on the outside: after the first line, we are inside, after the second we are outside, etc. If we do this an even number of times, the point at which we end up (the original point) must be outside.

The situation is not quite so simple, however, because some intersections might occur right at the vertices of the input polygon. Figure 24.5 shows some of the situations that must be handled. The first is a straightforward “outside” case; the second is a straightforward “inside” case; in the third, the test line leaves the polygon at a vertex (after touching two other vertices; and in the fourth, the test line coincides with an edge of the polygon before leaving. In some cases where the test line intersects a vertex it should count as one intersection with the polygon; in other cases it should count as none (or two). The reader may be amused to try to find a simple test to distinguish these cases before reading further.

The need to handle cases where polygon vertices fall on the test lines forces us to do more than just count the line segments in the polygon intersecting the test line. Essentially, we want to travel around the polygon, incrementing an intersection counter whenever we go from one side of the test line to another. One way to implement this is to simply ignore points that fall on the test line, as in the following program:

```
int inside(struct point t, struct point p[], int N)
{
    int i, count = 0, j = 0;
    struct line lt, lp;
    p[0] = p[N]; p[N+1] = p[1];
    lt.p1 = t; lt.p2 = t; lt.p2.x = INT_MAX;
    for (i = 1; i <= N; i++)
    {
        lp.p1 = p[i]; lp.p2 = p[j];
        if (!intersect(lp, lt))
        {
            lp.p2 = p[j]; j = i;
            if (intersect(lp, lt)) count++;
        }
    }
    return count & 1;
}
```

This program uses a horizontal test line for ease of calculation (imagine the diagrams in Figure 24.5 as rotated 45 degrees). The variable *j* is maintained as the index of the last point on the polygon known not to lie on the test line. The program assumes that *p*[1] is the point with the smallest *x* coordinate among all the points with the smallest *y* coordinate, so that if *p*[1] is on the test line, then *p*[0] cannot be. The same polygon can be represented by *N* different *p* arrays, but as this illustrates it is sometimes convenient to fix a standard rule for *p*[1]. (For example, this same rule is useful for *p*[1] as the “anchor” for the procedure

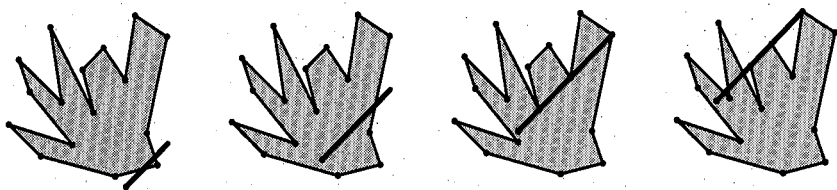


Figure 24.5 Cases to be handled by a point-in-polygon algorithm.

suggested above for computing a simple closed polygon.) If the next point on the polygon that is not on the test line is on the same side of the test line as the j th point, then we need not increment the intersection counter (`count`); otherwise we have an intersection. The reader may wish to check that this algorithm works properly for the cases in Figure 24.5.

If the polygon has only three or four sides, as is true in many applications, then such a complex program is not called for: a simpler procedure based on calls to `ccw` will be adequate. Another important special case is the *convex polygon*, to be studied in the next chapter, which has the property that no test line can have more than two intersections with the polygon. In this case, a procedure like binary search can be used to determine in $O(\log N)$ steps whether or not a point is inside.

Perspective

From the few examples given, it should be clear that it is easy to underestimate the difficulty of solving a particular geometric problem with a computer. There are many other elementary geometric computations that we have not treated at all. For example, a program to compute the area of a polygon makes an interesting exercise. However, the problems we've looked at have provided some basic tools that will be useful in later sections for solving the more difficult problems.

Some of the algorithms we'll study involve building geometric structures from a given set of points. The "simple closed polygon" is an elementary example of this. We will need to decide upon appropriate representations for such structures, develop algorithms to build them, and investigate their use in particular applications. As usual, these considerations are intertwined. For example, the algorithm used in the *inside* procedure in this chapter depends in an essential way on the representation of the simple closed polygon as an ordered set of points (rather than as an unordered set of lines).

Many of the algorithms we'll study involve *geometric search*: we want to know which points from a given set are close to a given point, or which points fall in a given rectangle, or which points are closest to one another. Many of the algorithms appropriate for such search problems are closely related to the search algorithms studied in Chapters 14–17. The parallels will be quite evident.


Few geometric algorithms have been analyzed to the point that precise statements can be made about their relative performance characteristics. As we've already seen, the running time of a geometric algorithm can depend on many things. The distribution of the points themselves, the order in which they appear in the input, and whether trigonometric functions are used can all significantly affect the running time of geometric algorithms. As usual in such situations, however, we do have empirical evidence that suggests good algorithms for particular applications. Also, many of the algorithms are derived from complexity studies and are designed for good worst-case performance.



Exercises

1. Give the value of `ccw` for the three cases when two of the points are identical (and the third is different), and for the case when all three points are identical.
2. Give a quick algorithm for determining whether two line segments are parallel, without using any divisions.
3. Give a quick algorithm for determining whether four line segments form a square, without using any divisions.
4. Given an array of `lines`, how would you test to see whether they form a simple closed polygon?
5. Draw the simple closed polygons that result from using A, C, and D in Figure 24.1 as “anchors” in the method described in the text.
6. Suppose that we use an arbitrary point for the “anchor” in the method described in the text for computing a simple closed polygon. Give conditions which such a point must satisfy for the method to work.
7. What does the `intersect` function return when called with two copies of the same line segment?
8. Does `inside` call a vertex of the polygon inside or outside?
9. What is the maximum value achievable by `count` when `inside` is executed on a polygon with `N` vertices? Give an example supporting your answer.
10. Write an efficient program for determining whether a given point is inside a given quadrilateral.

Finding the Convex Hull

 Often, when we have a large number of points to process, we're interested in the boundaries of the point set. People looking at a diagram of a set of points plotted in the plane, have little trouble distinguishing those on the "inside" of the point set from those lying on the edge. This distinction is a fundamental property of point sets; in this chapter we'll see how it can be precisely characterized by looking at algorithms for separating out the "natural boundary" points.

The mathematical way to describe the natural boundary of a point set depends on a geometric property called *convexity*. This is a simple concept that the reader may have encountered before: a *convex polygon* has the property that any line connecting any two points inside the polygon must itself lie entirely inside the polygon. For example, the "simple closed polygon" that we computed in the previous chapter is decidedly nonconvex; on the other hand, any triangle or rectangle is convex.

Now, the mathematical name for the natural boundary of a point set is the *convex hull*. The convex hull of a set of points in the plane is defined to be the smallest convex polygon containing them all. Equivalently, the convex hull is the shortest path surrounding the points. An obvious property of the convex hull that is easy to prove is that the vertices of the convex polygon defining the hull are points from the original point set. Given N points, some of them form a convex polygon within which all the others are contained. The problem is to find those points. Many algorithms have been developed to find the convex hull; in this chapter we'll examine some of the important ones.

Figure 25.1 shows our sample sets of points for Figure 24.1 and their convex hulls. There are 8 points on the hull of the small set and 15 points on the hull of the large set. In general, the convex hull can contain as few as three points (if the three points form a large triangle containing all the others) or as many as all the points (if they fall on a convex polygon, then the points comprise their own convex hull). The number of points on the convex hull of a "random" point set

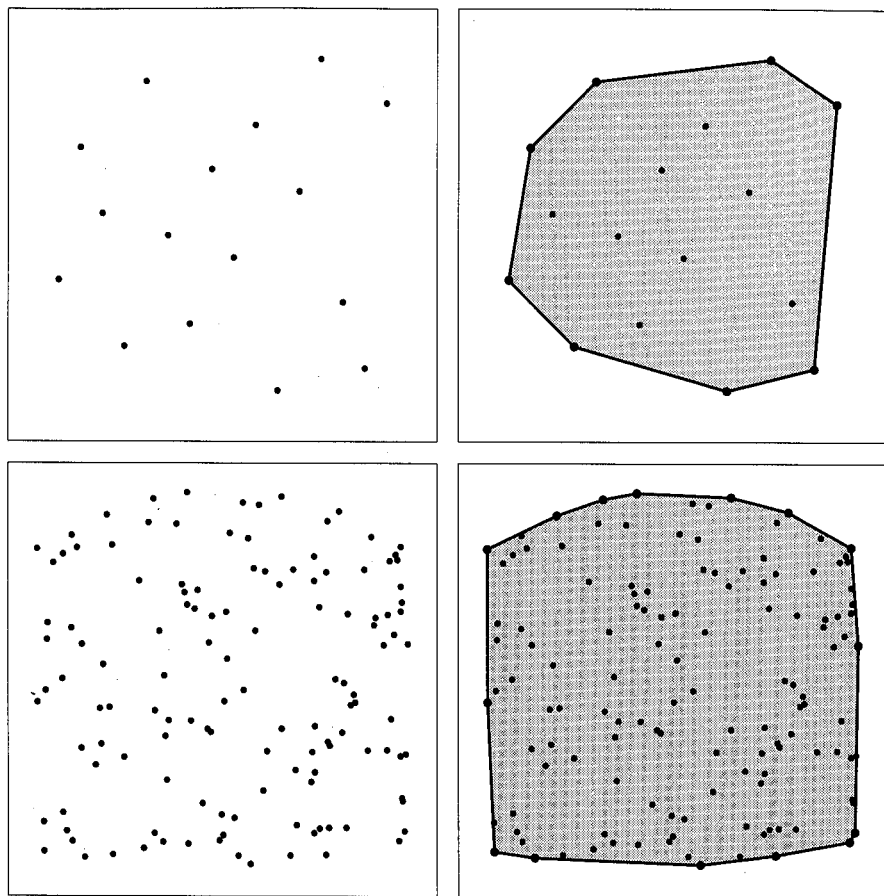


Figure 25.1 Convex hulls of the points of Figure 24.1.

falls somewhere in between these extremes, as we will see below. Some algorithms work well when there are many points on the convex hull; others work better when there are only a few.

A fundamental property of the convex hull is that any line outside the hull, when moved in any direction towards the hull, hits it at one of its vertex points. (This is an alternate way to define the hull: it is the subset of points from the point set that could be hit by a line moving in at some angle from infinity.) In particular, it's easy to find a few points guaranteed to be on the hull by applying this rule for horizontal and vertical lines: the points with the smallest and largest x and y coordinates are all on the convex hull. This fact is used as the starting point for the algorithms we consider.

Rules of the Game

The input to an algorithm for finding the convex hull is of course an array of points; we can use the `point` type defined in the previous chapter. The output is a polygon, also represented as an array of points with the property that tracing through the points in the order in which they appear in the array traces the outline of the polygon. On reflection, this might appear to require an extra ordering condition on the computation of the convex hull (why not just return the points on the hull in any order?), but output in the ordered form is obviously more useful, and it has been shown that the unordered computation is no easier to do. For all of the algorithms that we consider, it is convenient to do the computation *in place*: the array used for the original point set is also used to hold the output. The algorithms simply rearrange the points in the original array so that the convex hull appears in the first M positions, in order.

From the description above, it may be clear that computing the convex hull is closely related to sorting. In fact, a convex hull algorithm can be used to sort, in the following way. Given N numbers to sort, turn them into points (in polar coordinates) by treating the numbers as angles (suitably normalized) with a fixed radius for each point. The convex hull of this point set is an N -gon containing all of the points. Now, since the output must be ordered in the order in which the points appear on this polygon, it can be used to find the sorted order of the original values (remember that the input was unordered). This is not a formal proof that computing the convex hull is no easier than sorting, because, for example, the cost of the trigonometric functions required to convert the numbers into points on the polygon must be considered. Comparing convex hull algorithms (which involve trigonometric operations) to sorting algorithms (which involve comparisons between keys) is a bit like comparing apples to oranges, but even so it has been shown that any convex hull algorithm must require about $N \log N$ operations, the same as sorting (even though the operations allowed are likely to be quite different). It is helpful to view finding the convex hull of a set of points as a kind of “two-dimensional sort,” since frequent parallels to sorting algorithms arise in the study of algorithms for finding the convex hull.

In fact, the algorithms we’ll study show that finding the convex hull is no harder than sorting either: there are several algorithms that run in time proportional to $N \log N$ in the worst case. Many of the algorithms tend to use even less time on actual point sets, because their running time depends on how the points are distributed and on the number of points on the hull.

As with all geometric algorithms, we have to pay some attention to degenerate cases that are likely to occur in the input. For example, what is the convex hull of a set of points all of which fall on the same line segment? Depending upon the application, this could be all the points or just the two extreme points, or perhaps any set including the two extreme points would do. Though this seems an extreme example, it would not be unusual for more than two points to fall on one of the line

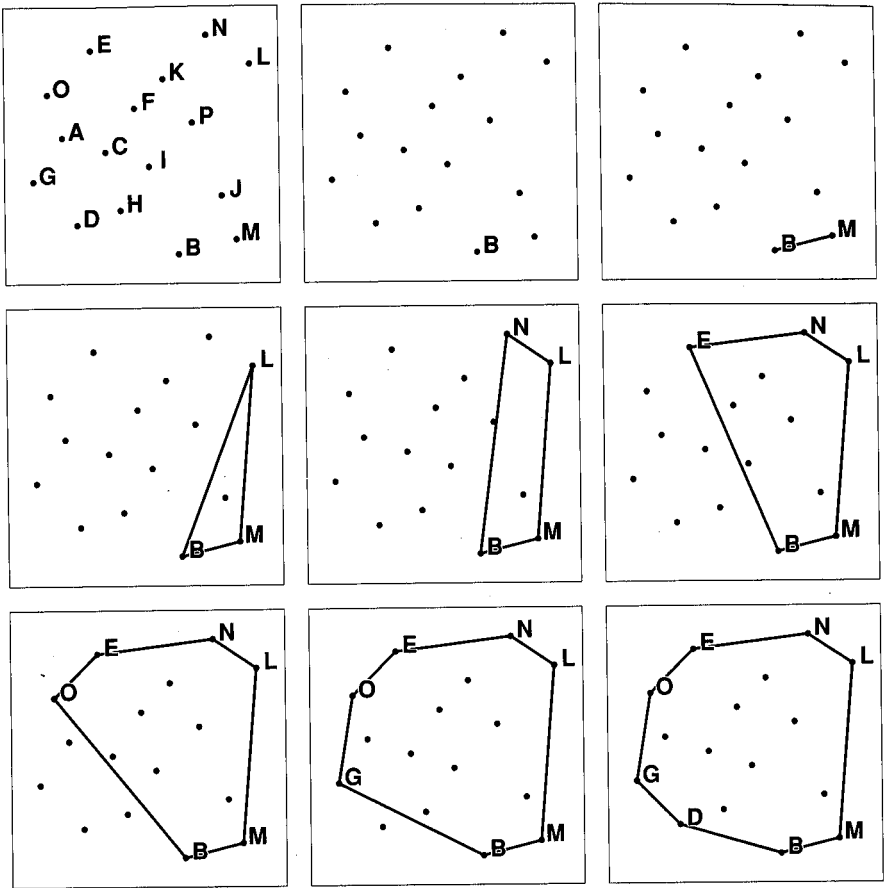


Figure 25.2 Package-wrapping.

segments defining the hull of a set of points. In the algorithms below, we won't insist that points falling on a hull edge be included, since this generally involves more work (though we will indicate how this could be done when appropriate). On the other hand, we won't insist that they be omitted either, since this condition could be tested afterwards if desired.

Package-Wrapping

The most natural convex hull algorithm, which parallels how a human would draw the convex hull of a set of points, is a systematic way to "wrap up" the set of points. Starting with some point guaranteed to be on the convex hull (say the one with the smallest y coordinate), take a horizontal ray in the positive direction and

“sweep” it upward until hitting another point; this point must be on the hull. Then anchor at that point and continue “sweeping” until hitting another point, etc., until the “package” is fully “wrapped” (the beginning point is included again). Figure 25.2 shows how the hull is discovered in this way for our sample set of points. Point B has the minimum y coordinate and is the starting point. Then M is the first point hit by the sweeping ray, then L, etc.

Of course, we don’t actually need to sweep through all possible angles; we just do a standard find-the-minimum computation to find the point that would be hit next. For each point to be included on the hull, we need to examine each point not yet included on the hull. Thus, the method is quite similar to selection sorting—we successively choose the “best” of the points not yet chosen, using a brute-force search for the minimum. The actual data movement involved is depicted in Figure 25.3: the Mth line of the table shows the situation after the Mth point is added to the hull.

The following program finds the convex hull of an array p of N points, represented as described at the beginning of Chapter 24. The basis for this implementation is the function `theta` developed in the previous chapter, which takes two points `p1` and `p2` as arguments and can be thought of as returning the angle between `p1`, `p2` and the horizontal (though it actually returns a more easily computed number with the same ordering properties). Otherwise, the implementation follows directly from the discussion above. A sentinel is needed for the find-the-minimum computation: though we normally might try to arrange things so that `p[0]` would be used, it is more convenient in this case to use `p[N+1]`.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
B	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P
B	M	C	D	E	F	G	H	I	J	K	L	A	N	O	P
B	M	L	D	E	F	G	H	I	J	K	C	A	N	O	P
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P
B	M	L	N	E	O	G	D	I	J	K	C	A	H	F	P

Figure 25.3 Data movement in package-wrapping.

```

int wrap(struct point p[], int N)
{
    int i, min, M;
    float th, v;
    struct point t;
    for (min = 0, i = 1; i < N; i++)
        if (p[i].y < p[min].y) min = i;
    p[N] = p[min]; th = 0.0;
    for (M = 0; M < N; M++)
    {
        t = p[M]; p[M] = p[min]; p[min] = t;
        min = N; v = th; th = 360.0;
        for (i = M+1; i <= N; i++)
            if (theta(p[M], p[i]) > v)
                if (theta(p[M], p[i]) < th)
                    { min = i; th = theta(p[M], p[min]); }
        if (min == N) return M;
    }
}

```

First, the point with the smallest y coordinate is found and copied into $p[N+1]$ in order to stop the loop, as described below. The variable M is maintained as the number of points so far included on the hull, and v is the current value of the “sweep” angle (the angle from the horizontal to the line between $p[M-1]$ and $p[M]$). The `for` loop puts the last point found into the hull by exchanging it with the M th point, and uses the `theta` function from the previous chapter to compute the angle from the horizontal made by the line between that point and each of the points not yet included on the hull, searching for the one whose angle is smallest among those with angles greater than v . The loop stops when the first point (actually the copy of the first point put into $p[N+1]$) is encountered again.

This program may or may not return points which fall on a convex hull edge. This situation is encountered when more than one point has the same `theta` value with $p[M]$ during the execution of the algorithm; the implementation above returns the point first encountered among such points, even though there may be others closer to $p[M]$. When it is important to find points falling on convex hull edges, we can achieve this by changing `theta` to take into account the distance between the points given as its arguments and assign the closer point a smaller value when two points have the same angle.

The major disadvantage of package-wrapping is that in the worst case, when all the points fall on the convex hull, the running time is proportional to N^2 (like that of selection sort). On the other hand, the method has the attractive feature that it generalizes to three (or more) dimensions. The convex hull of a set of points

in k -dimensional space is the minimal convex polytope containing them all, where a convex polytope is defined by the property that any line connecting two points inside must itself lie inside. For example, the convex hull of a set of points in 3-space is a convex three-dimensional object with flat faces. It can be found by “sweeping” a plane until the hull is hit, then “folding” faces of the plane, anchoring on different lines on the boundary of the hull, until the “package” is “wrapped.” (Like many geometric algorithms, it is rather easier to explain this generalization than to implement it!)

The Graham Scan

The next method we’ll examine, invented by R. L. Graham in 1972, is interesting because most of the computation involved is for sorting: the algorithm includes a sort followed by a relatively inexpensive (though not immediately obvious) computation. The algorithm starts by constructing a simple closed polygon from the points using the method of the previous chapter: sort the points using as keys the theta function values corresponding to the angle from the horizontal made from the line connecting each point with an ‘anchor’ point $p[1]$ (the one with the lowest y coordinate), so that tracing $p[1], p[2], \dots, p[N], p[1]$ gives a closed polygon. For our example set of points, we get the simple closed polygon of the previous chapter. Note that $p[N], p[1]$, and $p[2]$ are consecutive points on the hull; by sorting, we’ve essentially run the first iteration of the package-wrapping procedure (in both directions).

Computation of the convex hull is completed by proceeding around, trying to place each point on the hull and eliminating previously placed points that couldn’t possibly be on the hull. For our example, we consider the points in the order B M J L N P K F I E C O A H G D; the first few steps are shown in Figure 25.4. At the beginning, we know because of the sort that B and M are on the hull. When J is encountered, the algorithm includes it on the trial hull for the first three points.

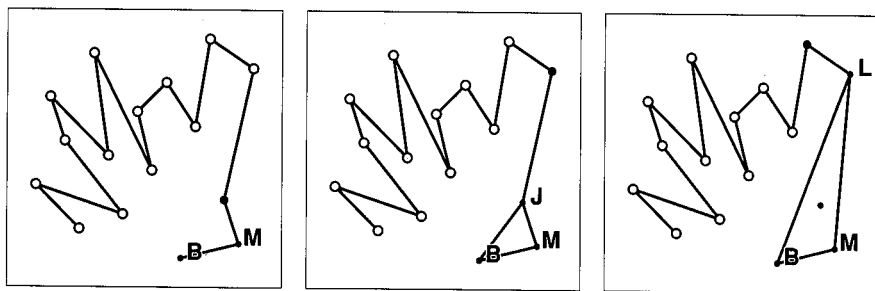


Figure 25.4 Start of Graham scan.

Then, when L is encountered, the algorithm finds out that J couldn't be on the hull (since, for example, it falls inside the triangle BML).

In general, testing which points to eliminate is not difficult. After each point has been added, we assume that we have eliminated enough points that what we have traced out so far could be part of the convex hull on the basis of the points so far seen. As we trace around, we expect to turn left at each hull vertex. If a new point causes us to turn *right*, then the point just added must be eliminated, since there exists a convex polygon containing it. Specifically, the test for eliminating a point uses the *ccw* procedure of the previous chapter, as follows. Suppose we have determined that $p[1], \dots, p[M]$ are on the partial hull determined on the basis of examining $p[1], \dots, p[i-1]$. When we come to examine a new point $p[i]$, we eliminate $p[M]$ from the hull if $\text{ccw}(p[M], p[M-1], p[i])$ is nonnegative. Otherwise, $p[M]$ could still be on the hull, so we don't eliminate it.

Figure 25.5 shows the completion of this process on our sample set of points. The situation as each new point is encountered is diagrammed: each new point is added to the partial hull so far constructed and is then used as a "witness" for the elimination of (zero or more) points previously considered. After L, N, and P are added to the hull, P is eliminated when K is considered (since NPK is a right turn), then F and I are added, leading to the consideration of E. At this point, I must be eliminated because FIE is a right turn, then F and K must be eliminated because KFE and NKE are right turns.. Thus more than one point can be eliminated during the "backup" procedure, perhaps several. Continuing in this way, the algorithm finally arrives back at B.

The initial sort guarantees that each point is considered in turn as a possible hull point, because all points considered earlier have a smaller *theta* value. Each line that survives the "eliminations" has the property that all points so far considered are on the same side of it, so that when we get back to $p[N]$, which also must be on the hull because of the sort, we have the complete convex hull of all the points.

As with the package-wrapping method, points on a hull edge may or may not be included, though there are two distinct situations that can arise with collinear points. First, if there are two points collinear with $p[1]$, then, as above, the sort using *theta* may or may not get them in order along their common line. Points out of order in this situation will be eliminated during the scan. Second, collinear points along the trial hull can arise (and not be eliminated).

Once the basic method is understood, the implementation is straightforward, though there are a number of details to attend to. First, the point with the maximum *x* value among all points with minimum *y* value is exchanged with $p[1]$. Next, *shellsort* is used to rearrange the points (any comparison-based sorting routine would do), modified as necessary to compare two points using their *theta* values with $p[1]$. After the sort, $p[N]$ is copied into $p[0]$ to serve as a sentinel in case $p[3]$ is not on the hull. Finally, the scan described above is performed. The following program finds the convex hull of the point set $p[1], \dots, p[N]$:

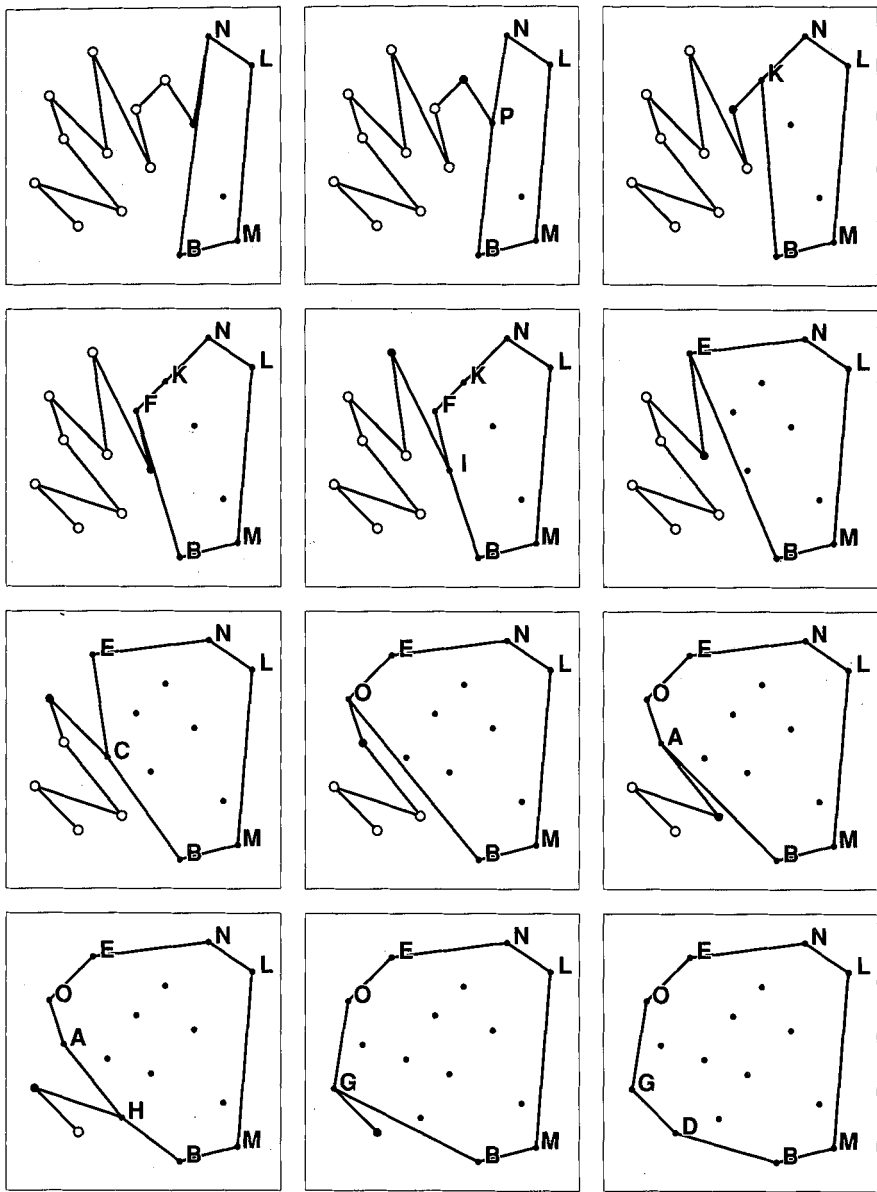


Figure 25.5 Completion of Graham scan.

```

int grahamscan(struct point p[], int N)
{
    int i, min, M;
    struct point t;
    for (min = 1, i = 2; i <= N; i++)
        if (p[i].y < p[min].y) min = i;
    for (i = 1; i <= N; i++)
        if (p[i].y == p[min].y)
            if (p[i].x > p[min].x) min = i;
    t = p[1]; p[1] = p[min]; p[min] = t;
    shellsort(p, N);
    p[0] = p[N];
    for (M = 3, i = 4; i <= N; i++)
    {
        while (ccw(p[M], p[M-1], p[i]) >= 0) M--;
        M++; t = p[M]; p[M] = p[i]; p[i] = t;
    }
    return M;
}

```

The loop maintains a partial hull in $p[1], \dots, p[M]$, as described above. For each new i value considered, M is decremented if necessary to eliminate points from the partial hull and then $p[i]$ is exchanged with $p[M+1]$ to (tentatively) add it to the partial hull. Figure 25.6 shows the contents of the p array each time a new point is considered for our example.

The reader may wish to check why it is necessary for the \min computation to find the point with the lowest x coordinate among all points with the lowest y coordinate, the canonical form described in Chapter 24. As discussed above, another subtle point is to consider the effect of the fact that collinear points lead to equal θ values, and may not be sorted in the order in which they appear on the line, as one might have hoped.

One reason that this method is interesting to study is that it is a simple form of *backtracking*, the algorithm design technique of “try something, and if it doesn’t work then try something else” that we’ll revisit in Chapter 44.

Interior Elimination

Almost any convex hull method can be vastly improved by a simple technique which quickly disposes of most points. The general idea is simple: pick four points known to be on the hull, then throw out everything inside the quadrilateral formed by those four points. This leaves many fewer points to be considered by, say, the Graham scan or the package wrapping technique.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	M	J	L	N	P	K	F	I	E	C	O	A	H	G	D
B	M	L	J												
B	M	L	N	J											
B	M	L	N	P	J										
B	M	L	N	K	J	P									
B	M	L	N	K	F	P	J								
B	M	L	N	K	F	I	J	P							
B	M	L	N	E	F	I	J	P	K						
B	M	L	N	E	C	I	J	P	K	F					
B	M	L	N	E	O	I	J	P	K	F	C				
B	M	L	N	E	O	A	J	P	K	F	C	I			
B	M	L	N	E	O	A	H	P	K	F	C	I	J		
B	M	L	N	E	O	G	H	P	K	F	C	I	J	A	
B	M	L	N	E	O	G	D	P	K	F	C	I	J	A	H

Figure 25.6 Data movement in the Graham scan.

The four points known to be on the hull should be chosen with an eye towards any information available about the input points. Generally, it is best to adapt the choice of points to the distribution of the input. For example, if all x and y values within certain ranges are equally likely (a rectangular distribution), then choosing four points by scanning in from the corners (find the four points with the largest and smallest sum and difference of the two coordinates) turns out to eliminate nearly all the points. Figure 25.7 shows that this technique eliminates most points not on the hull in our two example point sets.

In an implementation of the interior elimination method, the “inner loop” for random point sets is the test of whether or not a given point falls within the test quadrilateral. This can be speeded up somewhat by using a rectangle with edges parallel to the x and y axes. The largest such rectangle which fits in the quadrilateral described above is easy to find from the coordinates of the four points defining the quadrilateral. Using this rectangle will eliminate fewer points from the interior, but the speed of the test more than offsets this loss.

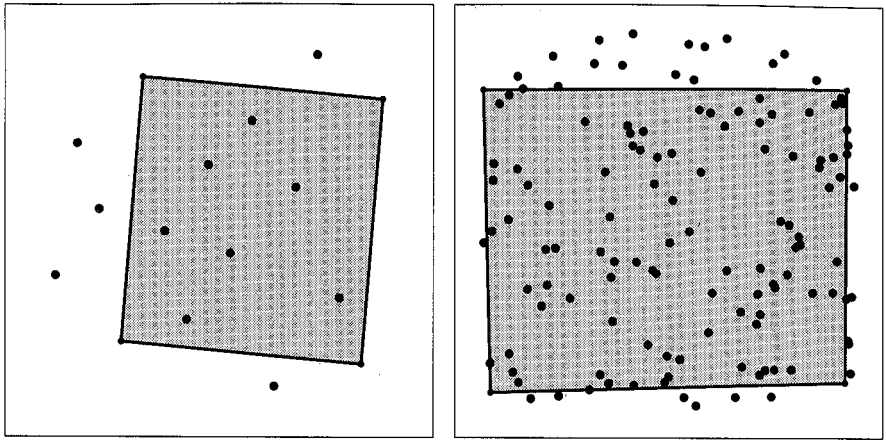


Figure 25.7 Interior elimination.

Performance Issues

As mentioned in the previous chapter, geometric algorithms are somewhat harder to analyze than algorithms in some of the other areas we've studied because the input (and the output) is more difficult to characterize. It often doesn't make sense to speak of "random" point sets: for example, as N gets large, the convex hull of points drawn from a rectangular distribution is extremely likely to be very close to the rectangle defining the distribution. The algorithms we've looked at depend on different properties of the point set distribution and are thus incomparable in practice, because to compare them analytically would require an understanding of very complicated interactions between little-understood properties of point sets. On the other hand, we can say some things about the performance of the algorithms that will help choosing one for a particular application.

Property 25.1 *After the sort, the Graham scan is a linear-time process.*

A moment's reflection is necessary to convince oneself that this is true, since there is a "loop-within-a-loop" in the program. However, it is easy to see that no point is "eliminated" more than once, so the code within that double loop is iterated fewer than N times. The total time required to find the convex hull using this method is $O(N \log N)$, but the "inner loop" of the method is the sort itself, which can be made efficient using techniques of Chapters 8-12. ■

Property 25.2 *If there are M vertices on the hull, then the "package-wrapping" technique requires about MN steps.*

First, we must compute $N - 1$ angles to find the minimum, then $N - 2$ to find the next, then $N - 3$, etc., so the total number of angle computations is $(N - 1) + (N - 2) + \dots + (N - M + 1)$, which is exactly equal to $MN - M(M - 1)/2$. To compare

this analytically with the Graham scan would require a formula for M in terms of N , a difficult problem in stochastic geometry. For a circular distribution (and some others) the answer is that M is $O(N^{1/3})$, and for values of N which are not large $N^{1/3}$ is comparable to $\log N$ (which is the expected value for a rectangular distribution), so this method will compete very favorably with the Graham scan. Of course, the N^2 worst case should always be taken into consideration. ■

Property 25.3 *The interior elimination method is linear, on the average.*

Full mathematical analysis of this method would require even more sophisticated stochastic geometry than above, but the general result is the same as that given by intuition: almost all the points fall inside the quadrilateral and are discarded—the number of points left over is $O(\sqrt{N})$. This is true even if the rectangle is used as described above. This makes the average running time of the whole convex hull algorithm proportional to N , since most points are examined only once (when they are thrown out). On the average, it doesn't matter much which method is afterwards, since so few points are likely to be left. However, to protect against the worst case (when all points are on the hull), it is prudent to use the Graham scan. This gives an algorithm which is almost sure to run in linear time in practice and is guaranteed to run in time proportional to $N \log N$. ■

The average-case result of Property 25.3 holds only for randomly distributed points in a rectangle, and in the worst case nothing is eliminated by the interior elimination method. However, for other distributions or for point sets with unknown properties, this method is still recommended because the cost is low (a linear scan through the points, with a few simple tests) and the possible savings is high (most of the points can be easily eliminated). The method also extends to higher dimensions.

It is possible to devise a recursive version of the interior elimination method: find extreme points and remove points on the interior of the defined quadrilateral as above, but then consider the remaining points as partitioned into subproblems which can be solved independently, using the same method. This recursive technique is similar to the Quicksort-like `select` procedure for selection discussed in Chapter 12. Like that procedure, it is vulnerable to an N^2 worst-case running time. For example, if all the original points are on the convex hull, then no points are thrown out in the recursive step. Like `select`, the running time is linear on the average (though it is not easy to prove this). But because so many points are eliminated in the first step, it is not likely to be worth the trouble to do further recursive decomposition in any practical application.



Exercises

1. Suppose you know in advance that the convex hull of a set of points is a triangle. Give an easy algorithm for finding the triangle. Answer the same question for a quadrilateral.
2. Give an efficient method for determining whether a point falls within a given convex polygon.
3. Implement a convex hull algorithm like insertion sort, using your method from the previous exercise.
4. Is it strictly necessary for the Graham scan to start with a point guaranteed to be on the hull? Explain why or why not.
5. Is it strictly necessary for the package-wrapping method to start with a point guaranteed to be on the hull? Explain why or why not.
6. Draw a set of points that makes the Graham scan for finding the convex hull particularly inefficient.
7. Does the Graham scan find the convex hull of the points that make up the vertices of *any* simple polygon? Explain why or give a counterexample showing why not.
8. What four points should be used for the interior elimination method if the input is assumed to be randomly distributed within a circle (using random polar coordinates)?
9. Empirically compare the Graham scan and the package-wrapping method for large point sets with both x and y equally likely to be between 0 and 1000.
10. Implement the interior elimination method and determine empirically how large N should be before one might expect fifty points to be left after the method is used on point sets with x and y equally likely to be between 0 and 1000.