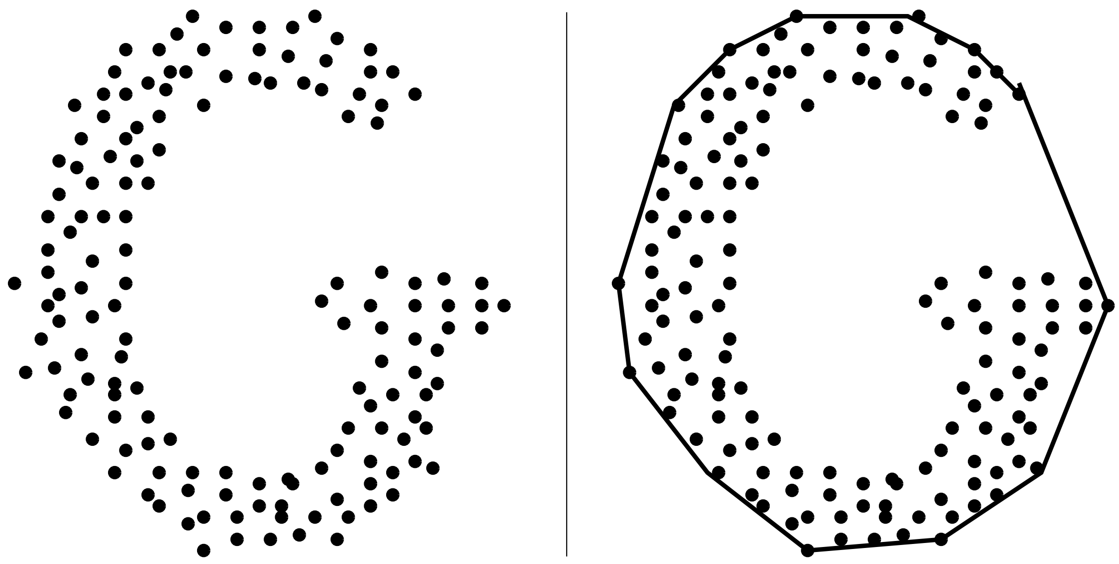

INPUT OUTPUT

17.2 Convex Hull

Input description: A set S of n points in d-dimensional space.

Problem description: Find the smallest convex polygon (or polyhedron) containing all the points of S.

Discussion: Finding the convex hull of a set of points is *the* most important elementary problem in computational geometry, just as sorting is the most important elementary problem in combinatorial algorithms. It arises because constructing the hull captures a rough idea of the shape or extent of a data set.

Convex hull serves as a first preprocessing step to many, if not most, geometric algorithms. For example, consider the problem of finding the diameter of a set of points, which is the pair of points that lie a maximum distance apart. The diameter must be between two points on the convex hull. The $O(n \lg n)$ algorithm for computing diameter proceeds by first constructing the convex hull, then for each hull vertex finding which other hull vertex lies farthest from it. The so-called "rotating-calipers" method can be used to move efficiently from one-diametrically opposed hull vertex to another by always proceeding in a clockwise fashion around the hull.

There are almost as many convex hull algorithms as sorting algorithms. Answer the following questions to help choose between them:

- *How many dimensions are you working with?* – Convex hulls in two and even three dimensions are fairly easy to work with. However, certain valid assumptions in lower dimensions break down as the dimensionality increases. For example, any n-vertex polygon in two dimensions has exactly n edges. However, the relationship between the numbers of faces and vertices becomes more complicated even in three dimensions. A cube has eight vertices and six faces, while an octahedron has eight faces and six vertices. This has implications for data structures that represent hulls—are you just looking for the hull points or do you need the defining polyhedron? Be aware of these complications of high-dimensional spaces if your problem takes you there.

  Simple $O(n \log n)$ convex-hull algorithms are available for the important special cases of two and three dimensions. In higher dimensions, things get more complicated. *Gift-wrapping* is the basic algorithm for constructing higher-dimensional convex hulls. Observe that a three-dimensional convex polyhedron is composed of two-dimensional faces, or *facets*, that are connected by one-dimensional lines or *edges*. Each edge joins exactly two facets together. Gift-wrapping starts by finding an initial facet associated with the lowest vertex and then conducting a breadth-first search from this facet to discover new, additional facets. Each edge e defining the boundary of a facet must be shared with one other facet. By running through each of the n points, we can identify which point defines the next facet with e. Essentially, we "wrap" the points one facet at a time by bending the wrapping paper around an edge until it hits the first point.

  The key to efficiency is making sure that each edge is explored only once. Implemented properly in d dimensions, gift-wrapping takes $O(n\phi_{d-1} + \phi_{d-2} \lg \phi_{d-2})$, where ϕ_{d-1} is the number of facets and ϕ_{d-2} is the number of edges in the convex hull. This can be as bad as $O(n^{\lfloor d/2 \rfloor + 1})$ when the convex hull is very complex. I recommend that you use one of the codes described below rather than roll your own.

- *Is your data given as vertices or half-spaces?* – The problem of finding the intersection of a set of n half-spaces in d dimensions (each containing the origin) is dual to that of computing convex hulls of n points in d dimensions. Thus, the same basic algorithm suffices for both problems. The necessary duality transformation is discussed in Section 17.15 (page 614). The problem of half-plane intersection differs from convex hull when no interior point is given, since the instance may be infeasible (i.e., the intersection of the half-planes empty).

- *How many points are likely to be on the hull?* – If your point set was generated "randomly," it is likely that most points lie within the interior of the hull. Planar convex-hull programs can be made more efficient in practice using the observation that the leftmost, rightmost, topmost, and bottommost points all must be on the convex hull. This usually gives a set of either three or four distinct points, defining a triangle or quadrilateral. Any point inside this region *cannot* be on the convex hull and so can be discarded in a linear sweep through the points. Ideally, only a few points will then remain to run through the full convex-hull algorithm.

This trick can also be applied beyond two dimensions, although it loses effectiveness as the dimension increases.

- *How do I find the shape of my point set?* – Although convex hulls provide a gross measure of shape, any details associated with concavities are lost. The convex hull of the G from the example input would be indistinguishable from the convex hull of O. *Alpha-shapes* are a more general structure that can be parameterized so as to retain arbitrarily large concavities. Implementations and references on alpha-shapes are included below.

The primary convex-hull algorithm in the plane is the *Graham scan*. Graham scan starts with one point p known to be on the convex hull (say the point with the lowest x-coordinate) and sorts the rest of the points in angular order around p. Starting with a hull consisting of p and the point with the smallest angle, we proceed counterclockwise around v adding points. If the angle formed by the new point and the last hull edge is less than 180 degrees, we add this new point to the hull. If the angle formed by the new point and the last "hull" edge is greater than 180 degrees, then a chain of vertices starting from the last hull edge must be deleted to maintain convexity. The total time is $O(n \lg n)$, since the bottleneck is the cost of sorting the points around v.

The basic Graham scan procedure can also be used to construct a nonself-intersecting (or *simple*) polygon passing through all the points. Sort the points around v, but instead of testing angles, simply connect the points in angular order. Connecting this to v gives a polygon without self-intersection, although it typically has many ugly skinny protrusions.

The gift-wrapping algorithm becomes especially simple in two dimensions, since each "facet" becomes an edge, each "edge" becomes a vertex of the polygon, and the "breadth-first search" simply walks around the hull in a clockwise or counter-clockwise order. The 2D gift-wrapping (or *Jarvis march*) algorithm runs in $O(nh)$ time, where h is the number of vertices on the convex hull. I recommend sticking with Graham scan unless you *really* know in advance that there cannot be too many vertices on the hull.

Implementations: The CGAL library (*www.cgal.org*) offers C++ implementations of an extensive variety of convex-hull algorithms for two, three, and arbitrary numbers of dimensions. Alternate C++ implementations of planar convex hulls includes LEDA (see Section 19.1.1 (page 658)).

Qhull [BDH97] is a popular low-dimensional, convex-hull code, optimized for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi vertices, and half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*.

O'Rourke [O'R01] provides a robust implementation of the Graham scan in two dimensions and an $O(n^2)$ implementation of an incremental algorithm for convex hulls in three dimensions. C and Java implementations are both available. See Section 19.1.10 (page 662).

For excellent alpha-shapes code, originating from the work of Edelsbrunner and Mucke [EM94], check out *http://biogeometry.duke.edu/software/alphashapes/*. Ken Clarkson's higher-dimensional convex-hull code *Hull* also does alpha-shapes, and is available at *http://www.netlib.org/voronoi/hull.html*.

Different codes are needed for enumerating the vertices of intersecting half-spaces in higher dimensions. Avis's *lhs* (*http://cgm.cs.mcgill.ca/∼avis/C/lrs.html*) is an arithmetically-robust ANSI C implementation of the Avis-Fukuda reverse search algorithm for vertex enumeration/convex-hull problems. Since the polyhedra is implicitly traversed but not explicitly stored in memory, even problems with very large output sizes can sometimes be solved.

Notes: Planar convex hulls play a similar role in computational geometry as sorting does in algorithm theory. Like sorting, convex hull is a fundamental problem for which many different algorithmic approaches lead to interesting or optimal algorithms. Quickhull and mergehull are examples of hull algorithms inspired by sorting algorithms [PS85]. A simple construction involving points on a parabola presented in Section 9.2.4 (page 322) reduces sorting to convex hull, so the information-theoretic lower bound for sorting implies that planar convex hull requires $\Omega(n \lg n)$ time to compute. A stronger lower bound is established in [Yao81].

Good expositions of the Graham scan algorithm [Gra72] and the Jarvis march [Jar73] include [dBvKOS00, CLRS01, O'R01, PS85]. The optimal planar convex-hull algorithm [KS86] takes $O(n \lg h)$ time, where h is the number of hull vertices that captures the best performance of both Graham scan and gift wrapping and is (theoretically) better in between. Planar convex hull can be efficiently computed *in-place*, meaning without requiring additional memory in [BIK$^+$04]. Seidel [Sei04] gives an up-to-date survey of convex hull algorithms and variants, particularly for higher dimensions.

Alpha-hulls, introduced in [EKS83], provide a useful notion of the shape of a point set. A generalization to three dimensions, with an implementation, is presented in [EM94].

Reverse-search algorithms for constructing convex hulls are effective in higher dimensions [AF96]. Through a clever lifting-map construction [ES86], the problem of building Voronoi diagrams in d-dimensions can be reduced to constructing convex hulls in $(d+1)$-dimensions. See Section 17.4 (page 576) for more details.

Dynamic algorithms for convex-hull maintenance are data structures that permit inserting and deleting arbitrary points while always representing the current convex hull. The first such dynamic data structure [OvL81] supported insertions and deletions in $O(\lg^2 n)$ time. More recently, Chan [Cha01] reduced the cost of such operation of near-logarithmic amortized time.

Related Problems: Sorting (see page 436), Voronoi diagrams (see page 576).