Computational geometry is the design and analysis of algorithms for geometric problems that arise in low dimensions, typically two or three dimensions. Many elegant algorithmic design and analysis techniques have been devised to attack geometric problems, and these problems have huge applications in many other fields.

Objectives of this lecture

In this lecture, we will

- see some motivating applications of computational geometry
- define some fundamental objects (points, lines, polygons) and common operations on them
- derive an efficient algorithm for the *convex hull* problem, a classic and very fundamental problem in computation geometry

1 Introduction

Computational geometry has applications in, and in some cases is the entire basis of, many fields. E.g.,

- Computer Graphics (rendering, hidden surface removal, illumination)
- Robotics (motion planning)
- Geographic Information Systems (Height of mountains, vegetation, population, cities, roads, electric lines)
- Computer-aided design (CAD) /Computer-aided manufacturing (CAM)
- Computer chip design and simulations
- Scientific Computation (Blood flow simulations, Molecular modeling and simulations)

1.1 Representations and Assumptions

The basic approach used by computers to handle complex geometric objects is to decompose the object into a large number of very simple objects. Examples:

- An image might be a 2D array of dots.
- An integrated circuit is a planar triangulation.
- Mickey Mouse is a surface of triangles

It is traditional to discuss geometric algorithms assuming that computing can be done on ideal objects, such as real valued points in the plane. The following chart gives some typical examples of representations.

Abstract Object	**Representation**
Real Number	Floating-point Number
Point	Pair of Reals
Line	Pair of Points, An Equation
Line Segment	Pair of Endpoints
Triangle	Triple of points

For the purpose of this course, we will assume a rather idealized model of computation, where we assume we can compute on the objects we care about (which might be integers, or sometimes might be arbitrary real numbers) in constant time without any loss of precision. For example, we will ignore things like rounding errors which would be a real-life consideration if you were actually implementing many of these algorithms.

Remark: Rounding errors...

In real life, computational geometry algorithms are often extremely susceptible to rounding errors. Algorithms which might be simple to describe on paper or in pseudocode can end up being very subtle and difficult to implement correctly because of it. In this course, we will mostly sweep this under the rug and focus on the elegance of the algorithms without this pain point in mind.

2 Fundamental Operations

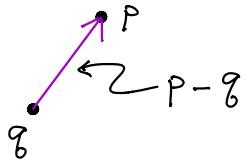
2.1 Operations on points and vectors

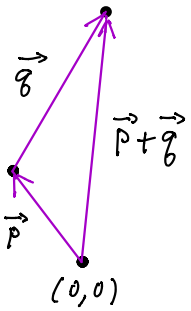
In computational geometry, the most primitive object is a *point*, which we can represent as a tuple of real numbers. We will mostly be focusing on 2D computational geometry, so points will be represented as pairs (x, y) of real numbers. In 2D or 3D space, a vector is a pair of 3-tuple of real numbers, respectively. Wait, that's just the same thing as a point! Indeed, the difference between a point and a vector is often just context, points are typically used to denote locations, and vectors are typically used to denote directions/lengths or differences, but mathematically they are interchangeable.

Addition and subtraction Let's start by defining pointwise addition/subtraction of vectors:

$$(x_1, y_1) + (x_2, y_2) \coloneqq (x_1 + x_2, y_1 + y_2)$$
$$(x_1, y_1) - (x_2, y_2) \coloneqq (x_1 - x_2, y_1 - y_2)$$

The effect of vector addition and subtraction can be visualized as in the following pictures. $p - q$ results in a vector that points from the point q to the point p. Adding $p + q$ results in a vector that points to the location you would get to if you start at $(0, 0)$ then follow p then q (equivalently, q then p).





Scalar multiplication We can define the multiplication of a vector by a scalar. Geometrically, multiplying a vector by a scalar α makes it α times longer.

$$\alpha(x, y) \coloneqq (\alpha x, \alpha y)$$

Magnitude/length The magnitude or length of a vector is given by the formula

$$\|(x, y)\| \coloneqq \sqrt{x^2 + y^2}$$

A particularly useful use case is to compute the distance between two points p and q, given by $\|p - q\|$.

The dot product The dot product operation takes two vectors and returns a real number. It is the first of two kinds of "multiplication" we will define for vectors. It is defined as

$$(x_1, y_1) \cdot (x_2, y_2) = x_1x_2 + y_1y_2$$

Dot products are useful for computing angles.

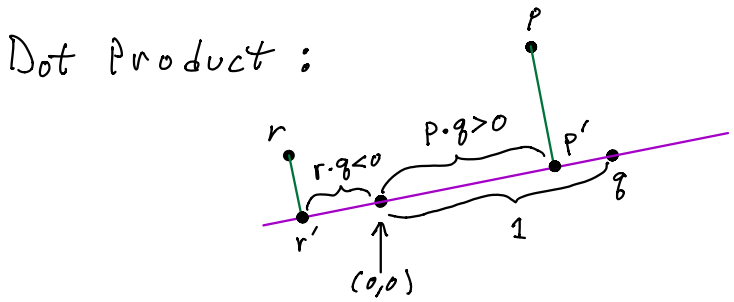
Theorem: Dot-product-angle formula

For any vectors u, v, we have

$$u \cdot v = \|u\|\|v\| \cos(\theta)$$

where θ is the angle between the vectors u and v.

Geometrically, the dot product can be seen as a projection operator. Consider the line L through $(0, 0)$ and q. Project p to the line L with a perpendicular (shown in green in the following figure.) Call this point p'. Then the value of $p \cdot q$ is the length of p', denoted $|p'|$, times the length of q. That is, $|p'| * |q|$. Like the dot product this value is signed. If $(0, 0)$ is not between p' and q then it's positive, otherwise it's negative. The figure below shows the case when $|q| = 1$, which is commonly used to compute projections.



Algorithm: Projection onto a line

Given a point p and a line L, such that L goes through the origin and a point q such that $|q| = 1$, we can compute the projection[a] of p onto the line by

$$p' = (p \cdot q)q$$

[a] The *projection* p' of the point p onto a line is the closest point on the line to p

Exercise: Projection onto a line, but more general

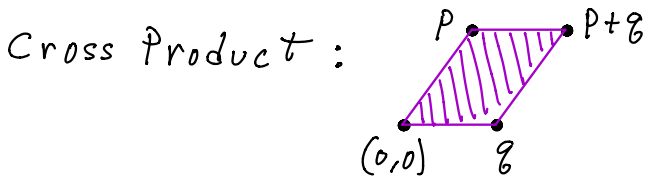
Explain how to compute the projection of a point p in 2D onto any line L defined by two points a and b by using the algorithm above with some extra steps.

The cross product The cross product is the other kind of "multiplication" between two points. In 2D, the cross product is defined as

$$(x_1, y_1) \times (x_2, y_2) \coloneqq \det\left(\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix}\right) = x_1y_2 - y_1x_2$$

where $\det(M)$ is the determinant of M. The cross product of p and q is the signed area of the parallogram with the four vertices $(0, 0), p, q, p + q$. Equivalently, it is twice the signed area of the triangle with vertices

$(0,0), p, q$. The sign is determined by the "right hand" rule. This means that if the angle at $(0,0)$ starting at p, going clockwise around $(0,0)$ and ending at q is less than 180 degrees, the cross product is negative, otherwise it's positive. It's zero of the angle between p and q is 0 or 180 degrees.



There is also a formula analogous to the dot product angle formula for the cross product.

Theorem: Cross-product-angle formula

For any 2D vectors u, v, we have

$$u \times v = \|u\|\|v\| \sin(\theta)$$

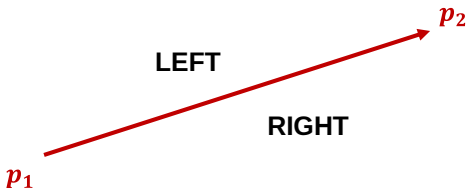
where θ is the angle between the vectors u and v.

Exercise: Point-to-line distance

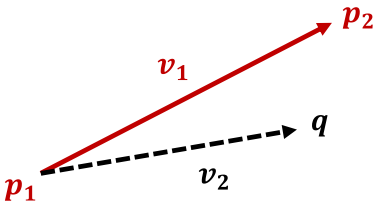
Given a point p in 2D and a line L defined by two points a and b, describe how to compute the distance from p to the line L using the cross product. (One solution is to use the projection algorithm from earlier, but here we want a different solution.)

2.2 Line-side test

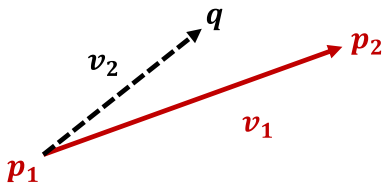
The line-side test is the next most fundamental primitive we will introduce, and turns out to be a critical ingredient in a lot of geometry algorithms. Given three points p_1, p_2, q, the output of the *line side test* is "LEFT" if the point q is to the left of ray $p_1 \to p_2$, "RIGHT" if the point q is to the right of ray $p_1 \to p_2$, and "ON" if it is on that ray.



The algorithm is to construct vectors $v_1 = p_2 - p_1$ and $v_2 = q - p_1$, then take the cross product of v_1 and v_2 and look at its value compared to 0.



$v_1 \times v_2 < 0$



$v_1 \times v_2 > 0$

Notice that if a point q is on the right, then the cross product $v_1 \times v_2$ will be negative, or if q is on the left, it will be negative. If q happens to be on the ray, then the cross product is zero.

Algorithm: Line-side test

```
let v1 = p2 − p1 and v2 = q − p1
if v1 × v2 < 0 then return RIGHT
else if v1 × v2 > 0 then return LEFT
else return ON
```

Exercise: Line segment intersection

Given two line segments defined by the points a and b and the points c and d respectively, determine whether the two of them intersect or not. Hint: Use the line-side test primitive.

2.3 Convex combinations

Since we now know how to add and scale points/vectors, lets see what kinds of other objects we can make just by combining points together in various ways. Our focus will be in 2D, but you could also think about what some of these constructions look like in 3D if you want. We will consider *convex combinations* of points.

Definition: Convex combination

A *convex combination* of the points $p_1, p_2, \ldots p_k \in \mathbb{R}^d$ is a point

$$p' = \sum_{i=1}^{k} \alpha_i p_i,$$

such that $\sum a_i = 1$ and $\alpha_i \geq 0$.

Given this definition, an interesting question to ask is given a set of points, what does the set of *all possible convex combinations* of those points look like? We can start with just two points.

Claim: A convex combination of two points

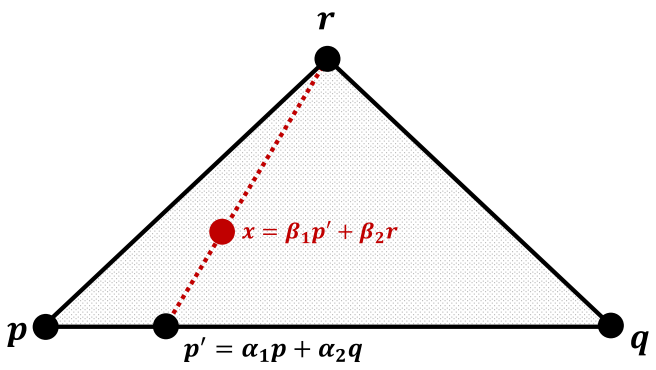
Given two points p and q, any convex combination of p and q is a point on the line connecting p and q. More specifically, the set of all convex combinations of p and q is the line between p and q.

For example, if we take two points p and q and set $\alpha_1 = \alpha_2 = \frac{1}{2}$, we get $\frac{1}{2}p + \frac{1}{2}q$, which is the midpoint of p and q (the middle of the line connecting them). What if we take three points p, q, r?

Claim: A convex combination of three points

The convex combinations of three points p, q, r form the triangle with vertices p, q, r.

How can we justify this? Lets use the fact from above that the convex combinations of two points are the line between them. So, for any point on the line between p and q, we can form a convex combination that achieves that point. Now, for any point inside the triangle pqr, say x, that we want to reach, we can first take a convex combination $p' = \alpha_1 p + \alpha_2 q$ such that p' is the intersection of the lines pq and rx. Then take a second convex combination with that point and r, to get $x = \beta_1 p' + \beta_2 r$.

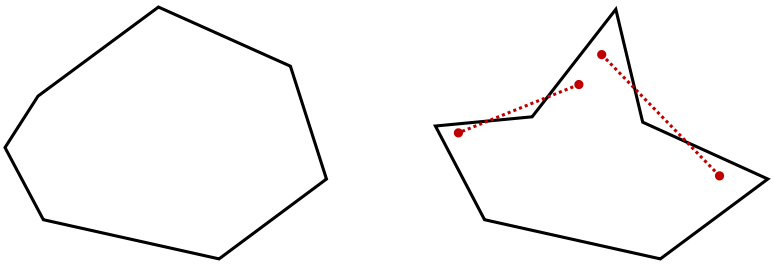


Exercise

Suppose we consider $k > 3$ points $p_1, \ldots, p_k$. Describe the set of points formed by all convex combinations of them. Justify your answer by using the fact above that the convex combinations of three points span a triangle.

3 The Convex Hull

This is the "sorting problem" of computational geometry. There are many algorithms for the problem, and there are often analogous to well-known sorting algorithms. There are also well studied lower bounds! A point set $P \subseteq \mathbb{R}^d$ is *convex* if it is closed under convex combinations. That is, if we take any convex combination of any two points in A, the result is a point in P. In other words, when we walk along the straight line between any pair of points in P we find that the entire path is also inside of P.



Convex **Non-convex**

We saw convex sets before when we talked about linear programming. Based on this definition, we can define the *convex closure* of a set of points P to be the smallest convex set containing P. This is well-defined and unique for any point set P. This brings us to the definition of the object we really care about:

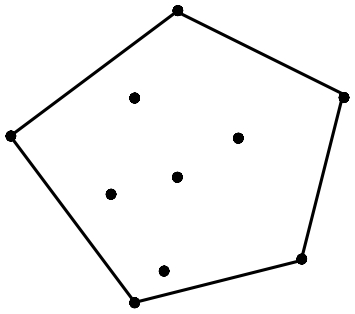
Definition: Convex hull

The *convex hull* of a set of points P is the boundary of the convex closure of P. That is, it is the smallest convex polygon that contains all of the points in P, either on its boundary or interior.

These definitions are general and apply to any closed set of points, including infinite sets, but for our purposes we're only interested in the ConvexClosure(P) and ConvexHull(P) when P is a finite set of points.

A computer representation of a convex hull must include the combinatorial structure. In two dimensions,

this just means a simple polygon in, say counter-clockwise order. (In three dimensions it's a planar graph of vertices edges and faces) The vertices are a subset of the input points. So in this context, a 2D convex hull algorithm takes as input a finite set of n points $P \in \mathbb{R}^2$, and produces a list H of points from P which are the vertices of the ConvexHull(A) in counter-clockwise order. This figure shows the convex hull of 10 points.



Today we're going to focus on algorithms for convex hulls in 2-dimensions. We first present an $O(n^2)$ algorithm, than refine it to run in $O(n \log n)$. To slightly simplify the exposition we're going to assume that no three points of the input are colinear.

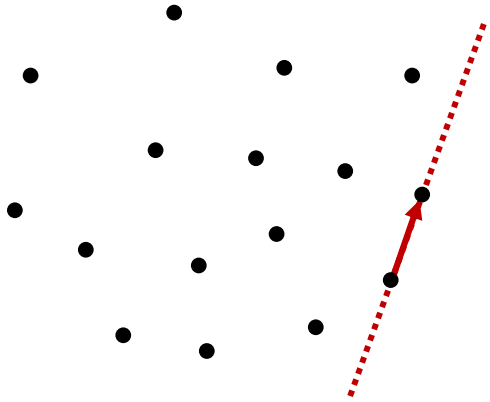
3.1 Warmup: Slow algorithms for 2D Convex Hulls

First we give a trivial $O(n^3)$ algorithm for convex hull. It might be slow, but it starts us off in the right direction by thinking about some useful facts about them.

Claim

A directed segment between a pair of points (p_i, p_j) is on the convex hull if and only if all other points p_k are to the left of the ray through p_i and p_j.

We illustrate this fact with the following "proof by picture".

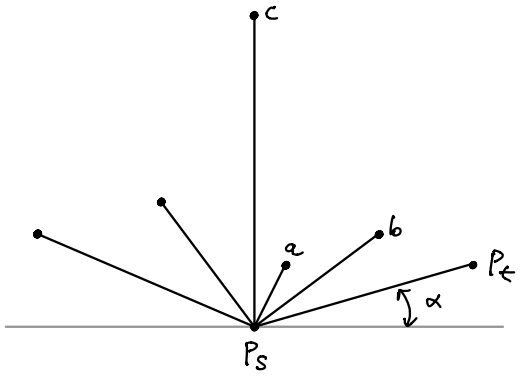


Note that no point to the right of the ray can be in the convex hull because that entire half-plane is devoid of points from the input set. And the points on the segment (p_i, p_j) are in the ConvexClosure of the input points. Therefore the segment is on the boundary of the ConvexClosure. Therefore it is on the convex hull.

An $O(n^3)$-time algorithm Using this observation, we get an $O(n^3)$ algorithm by brute-force. Just try every pair of points p_i, p_j and then line side test every other point p_k. If every p_k is on the left, then add the edge $p_i \to p_j$ to the hull. After adding all of the edges, sort them into counterclockwise order.

An $O(n^2)$-time algorithm To get this to run in $O(n^2)$ time we just have to be a bit more organized. In the first algorithm we just found all of the edges of the hull in an arbitrary order. Lets try to find them in order this time. This will cut down the amount of work we have to do.

The first observation is that if we take the point with the lowest y-coordinate, this point must be on the convex hull. Call it p_s. Suppose we now measure the angle from p_s to all the other points. These angles range from 0 to π. If we take the point p_t with the smallest such angle, then we know that (p_s, p_t) is on the convex hull. The following figure illustrates this.



All the other points must be to the left of segment (p_s, p_t). We can continue this process to find the point p_u which is the one with the smallest angle with respect to (p_s, p_t). This process is continued until all the points are exhausted. The running time is $O(n)$ to find each segment. There are $O(n)$ segments, so the algorithm is $O(n^2)$.

Actually we don't need to compute angles. The line-side-test can be used for this instead. For example look at what happens after we've found p_s and p_t. We process possibilities for the next point in any order. Say we start from a in the figure. Then we try b, and note that b is on the right side of segment (p_t, a) so we jettison a and continue with (p_t, b). But then we then throw out b in favor of c. It turns out that the remaining points are all to the left of segment (p_t, c). Thus $c = p_u$ is the next point on the convex hull.

3.2 Graham Scan, an $O(n \log n)$ Algorithm for 2D Convex Hulls

The $O(n^2)$ algorithm above is actually pretty close. We can convert it into an $O(n \log n)$ algorithm with a slight improvement. We went from $O(n^3)$ to $O(n^2)$ by finding the sides of the hull in order, but we are still searching through the *points* in an arbitrary order to do so. So, instead of processing the points in an arbitrary order, lets process them in order of increasing angle with respect to the point p_s.

Let's relabel the points so that $p_0 = p_s$ is the starting point, and $p_1, p_2 \ldots$ are the remaining points in order of increasing angle with respect to p_0. From the discussion above we know that (p_0, p_1) is an edge of the convex hull. The Graham Scan works as follows.

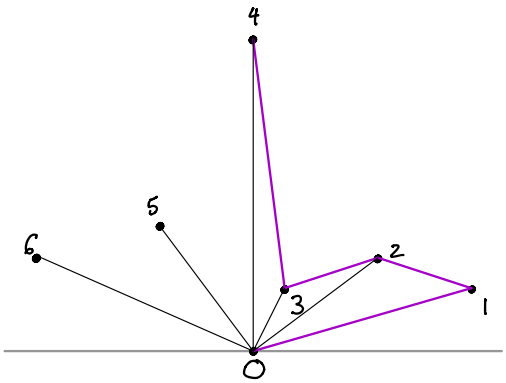
Algorithm: Graham Scan

We maintain a "chain" of points that starts with $p_0, p_1, \ldots$. This chain has the property that each step is always a left turn with respect to the previous element of the chain. We try to extend the chain by taking the next point in the sorted order. If this has a left turn with respect to the current

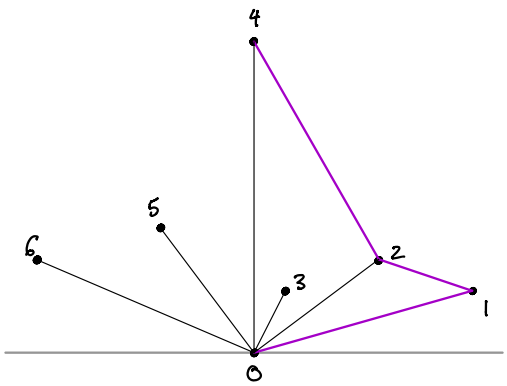
chain, we keep it. Otherwise we remove the last element of the chain (repeatedly) until the chain is again restored to be all left turns.

```
find lowest point p_0
sort points p_1,p_2,... counterclockwise by their angle with p_0
H = [p_0, p_1]
for each point i = 2,3,...
   while LineSideTest(H[-2],H[-1],p_i) is RIGHT
      H.pop()
   H.append(p_i)
return H
```

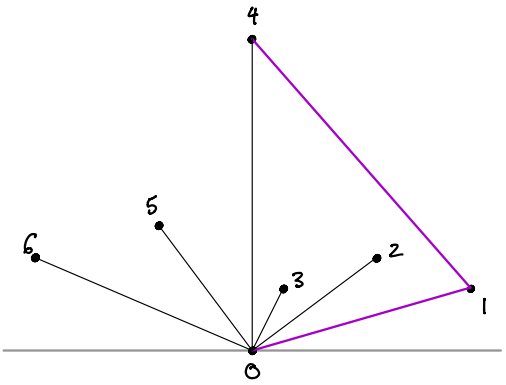
Here's an example of the algorithm.



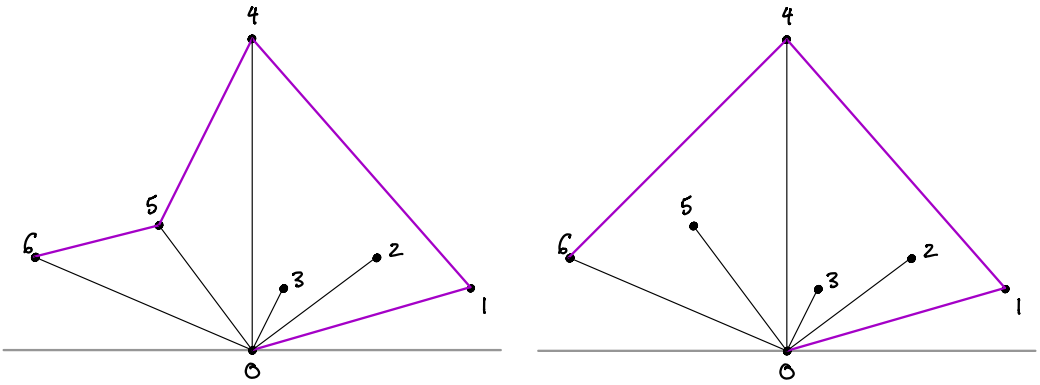
At this point we've formed the chain p_0, p_1, p_2, p_3, p_4. But the last step (from p_3 to p_4) is a right turn. So we delete p_3 from the chain. Now we have:



Now at p_2 we have a right turn, so we remove it, giving:



Now the process continues with points p_5 and p_6. When p_6 is added, p_5 becomes a right turn, so it's removed.



After all the points are processed in this way, we can just add the last segment from p_{n-1} to p_0, to close the polygon, which will be the convex hull.

Each point can be added at most once to the sequence of vertices, and each point can be removed at most once. Thus the running time of the scan is $O(n)$. But remember we already paid $O(n \log n)$ for sorting the points at the beginning of the algorithm, which makes the overall running time of the algorithm $O(n \log n)$.

The reason this algorithm works is because whenever we delete a point we have implicitly shown that it is a convex combination of other points. For example when we deleted p_3 we know that it is inside of the triangle formed by p_0, p_2 and p_4. Because of the presorting p_3 is to the left of (p_0, p_2), and to the right of (p_0, p_4). And because (p_2, p_3, p_4) is a right turn, p_3 is to the left of (p_2, p_4).

At the end the chain is all left turns, with nothing outside of it. Therefore it must be the convex hull.

3.3 Lower bound for computing the convex hull

We said earlier that convex hull is the "sorting" of computational geometry because there are a huge range of algorithms for it that use a wide range of different algorithmic techniques. Also like sorting, it admits a similar lower bound! We won't prove it here because we would need to address too many subtle details of the model, but the result is quite nice!

Theorem: Sidedness lower bound for convex hull

For any algorithm that computes a convex hull using line-side tests, at least $\Omega(n \log n)$ line-side tests are necessary.

The cool thing here is how we measure complexity. Just like how, for sorting, we counted the number of comparisons and showed a lower bound based on that, for convex hull, we can provide a lower bound in terms of the number of *line-side tests* that we need to perform. Just like we can also do sorting outside the comparison model and obtain different bounds, it is also possible to write convex hull algorithms that do not use line-side tests, so this result does not apply to those algorithms. It turns out, however, that the vast majority of known convex hull algorithm are based on line-side tests, so it applies quite widely!