1 Introduction

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. This is why I've included this topic in this course.

Some applications of CG:

```
Computer Graphics
  images creation
  hidden surface removal
  illumination

Robotics
  motion planning

Geographic Information Systems
  Height of mountains
  vegetation
  population
  cities, roads, electric lines

CAD/CAM computer aided design/computer aided manufacturing

Computer chip design and simulations

Scientific Computation
  Blood flow simulations
  Molecular modeling and simulations
```

Basic algorithmic design approaches:

- Divide-and-Conquer
- Line-Sweep (typically in 2D)
- Random Incremental

In this course there will be three lectures on computational geometry covering the following topics:

- Geometric primitives
- Convex hull in 2D
- Sweep line algorithm for intersecting a set of segments
- Two algorithms for the point location problem

1.1 Representations

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, Big Number
Point	Pair of Reals
Line	Pair of Points, An Equation
Line Segment	Pair of Endpoints
Triangle	Triple of points
Etc	

1.2 Using Points to Generate Objects

Suppose $P_1, P_2, \ldots P_k \in \mathbb{R}^d$. Below are several ways to use these points to generate more complex objects.

Linear Combination

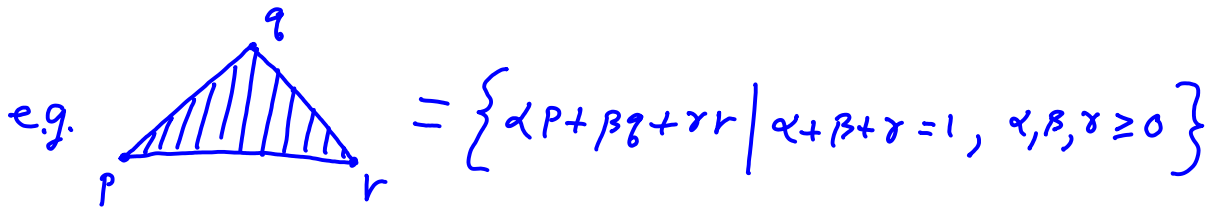
$$\text{Subspace} = \sum \alpha_i P_i \qquad \text{where} \qquad \alpha_i \in \mathbb{R}$$

Affine Combination

$$\text{Plane} = \sum \alpha_i P_i \qquad \text{where} \qquad \sum \alpha_i = 1, \;\; \alpha_i \in \mathbb{R}$$

Convex Combination

$$\text{Body} = \sum \alpha_i P_i \qquad \text{where} \qquad \sum \alpha_i = 1, \;\; \alpha_i \geq 0 \;\; \alpha_i \in \mathbb{R}$$



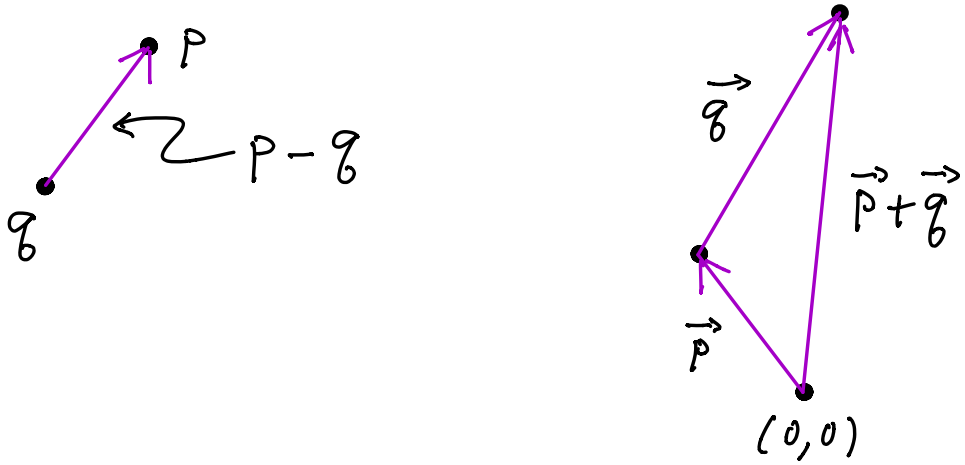
2 Primitive Operations

Basics

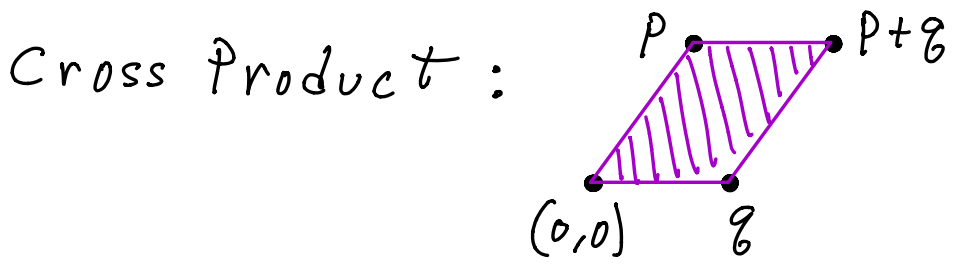
I'll be giving integer implementations of these primitives in ocaml. Let's start with some basic operations on vectors in 2D. The code below defines vector addition subtraction, cross product, dot product and the sign of a number.

```
let (--) (x1,y1) (x2,y2) = (x1-x2, y1-y2)
let (++) (x1,y1) (x2,y2) = (x1+x2, y1+y2)
let cross (x1,y1) (x2,y2) = (x1*y2) - (y1*x2)
let dot (x1,y1) (x2,y2) = (x1*x2) + (y1*y2)
let sign x = compare x 0
  (* returns -1 if x<0, 0 if x=0 and 1 if x>0 *)
```

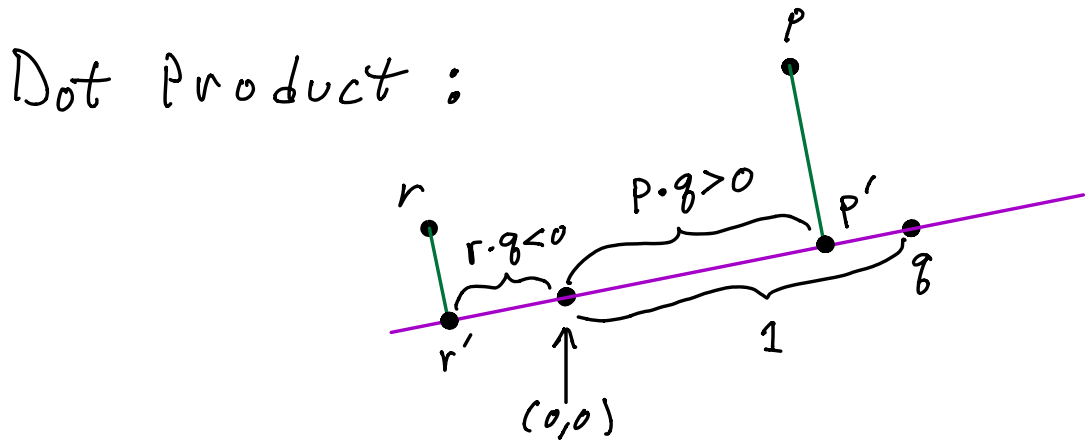
The first two operations are subtraction and addition of vectors, as shown in the following figure. Note that sometimes we view a pair (x, y) as a vector, and sometimes we view it as a point.



The cross product of p and q, denoted $p \times q$, is the signed area of the parallogram with the four vertices $(0, 0), p, q, p+q$. The sign is determined by the "right hand" rule. Alternatively 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.



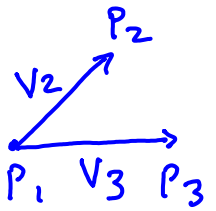
The dot product is a projection operator. Consider the line L through $(0,0)$ and q. Project p to the line L with a perpendicular (shown in green in the following figure.) Call this point p'. Then the value of $p \cdot q$ is the length of p', denoted $|p'|$, times the length of q. That is, $|p'| * |q|$. 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 $|q| = 1$.

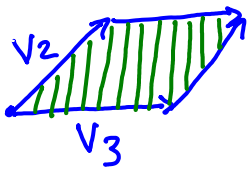


Line Side Test

Given three points P_1, P_2, P_3, the output of the *line side test* is "LEFT" if the point P_3 is to the left of ray $P_1 \rightarrow P_2$, "RIGHT" if the point P_3 is to the right of ray $P_1 \rightarrow P_2$, and "ON" if it is on that ray.

The algorithm is to construct vectors $V_2 = P_2 - P_1$ and $V_3 = P_3 - P_1$. Then take the cross product of V_2 and V_3 and look at its value compared to 0.





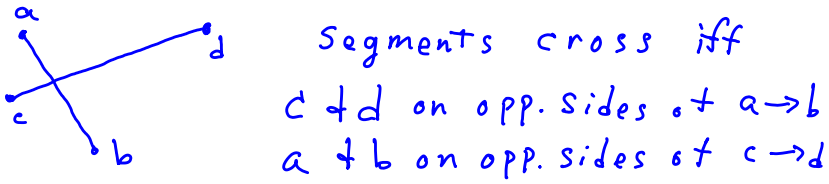
$V_2 \times V_3$ = Signed area of parallelogram
(in this case negative by the right hand rule.)

Here is an implementation of this test in ocaml which returns 1 if p_3 is LEFT of ray $p_1 \rightarrow p_2$, -1 if RIGHT, and 0 if ON.

```
let line_side_test p1 p2 p3 = sign (cross (p2--p1) (p3--p1))
```

Line segment intersection testing

Here we are given two line segments (a, b) and (c, d) (where a, b, c, d are points), and we have to determine if they cross. We can do this using four line-side tests, as illustrated here.




```
let segments_intersect (a,b) (c,d) =
  (line_side_test a b d) * (line_side_test a b c) <= 0 &&
  (line_side_test c d a) * (line_side_test c d b) <= 0
```

By changing the `<=` into a `<`, this can be changed into a strict intersection test, which would require the segments to intersect at a point interior to both of them.

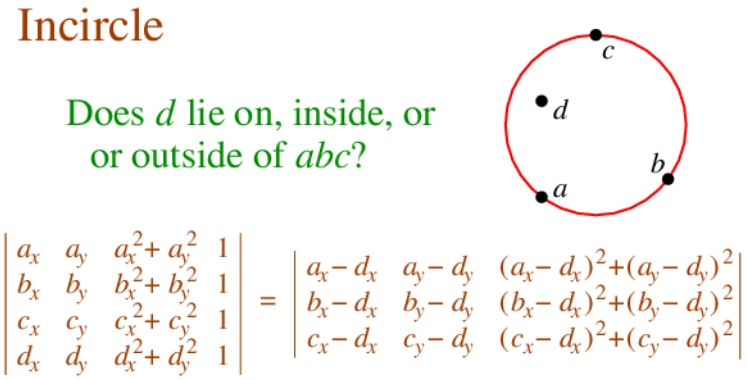
In-circle test

Three non-colinear points determine a circle. The *in-circle test* will tell us the relationship of a fourth point to the circle determined by the other three points. So the test takes points a, b, c, and d as inputs, and returns 1, 0, or -1 as follows:

> This returns 0 if the four points are on the same circle (or straight line.) Suppose I walk forward around the circle with my right hand on the circle from $a \to b \to c$. It returns 1 if d is on the same side of the circle as my body, and -1 otherwise. It's a fourth degree function in the given coordinates.

```
let incircle (ax,ay) (bx,by) (cx,cy) (dx,dy) =
  let det ((a,b,c),(d,e,f),(g,h,i)) =
    a*(e*i - f*h) - b*(d*i - f*g) + c*(d*h - e*g)
  in

  let row ax dx ay dy =
    let a = ax - dx in
    let b = ay - dy in
      (a, b, (a*a) + (b*b))
  in
    sign (det (row ax dx ay dy, row bx dx by dy, row cx dx cy dy))
```



$$\begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix} = \begin{vmatrix} a_x-d_x & a_y-d_y & (a_x-d_x)^2+(a_y-d_y)^2 \\ b_x-d_x & b_y-d_y & (b_x-d_x)^2+(b_y-d_y)^2 \\ c_x-d_x & c_y-d_y & (c_x-d_x)^2+(c_y-d_y)^2 \end{vmatrix}$$

The picture above illustrates a case when the incircle test would return 1. (This figure was taken from `http://www.cs.cmu.edu/~quake/robust.html`)

3 Computing 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.

A point set $A \subseteq \mathbb{R}^d$ 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 A. In other words if when we walk along the straight line between any pair of points in A we find that the entire path is also inside of A, then the set A is convex.

We saw convex sets before when we talked about linear programming. One observation we used at that time is that the intersection of any two convex sets is convex.

Definition: ConvexClosure(A) = smallest convex set containing A

This is well-defined and unique for any point set A. (We won't prove this.) Assuming that the set A is a closed set of points we can define the convex hull of A as follows:

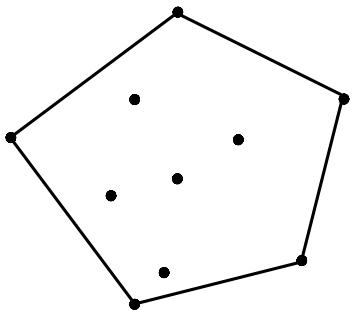
Definition: ConvexHull(A) = boundary of ConvexClosure(A). (A point p is on the boundary of S if for any $\epsilon > 0$ there exists a point within ϵ of p that is inside S and also another point with ϵ of p that is outside of S.)

These definitions are general and apply to any closed set of points.

For our purposes we're only interested in the ConvexClosure(A) and ConvexHull(A) when A is a finite set of points. In this case the ConvexClosure will be a closed polyhedron.

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 $A \in \mathbb{R}^2$, and produces a list L of points from A 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 An $O(n^2)$ Algorithm for 2D Convex Hulls

First we give a trivial $O(n^3)$ algorithm for convex hull. The idea is that a directed segment between a pair of points (P_i, P_j) is on the convex hull iff all other points P_k are to the left of the ray from

P_i to P_j. 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.

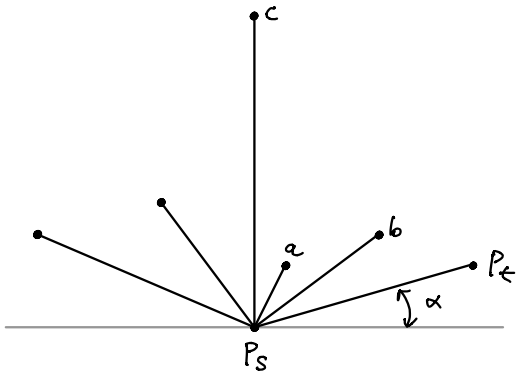
Here's the pseudo-code for this algorithm.

Slow-Hull$(P_1, P_2, \ldots, P_n)$:
    For each distinct pair of indices (i, j) do
        if for all $1 \leq k \leq n$ and $k \neq i$ and $k \neq j$
        it is the case that P_k is to the left of segment (P_i, P_j)
        Then output (i, j).
    done

(To make this into a proper convex hull algorithm, a final pass would be required to turn this list of pairs of indices into an ordered list of points in counterclockwise order.)

To get this to run in $O(n^2)$ time we just have to be a bit more organized.

The first observation is that if we take the point with the lowest y-coordinate, this point must be on the contex 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 $O(n^2)$.

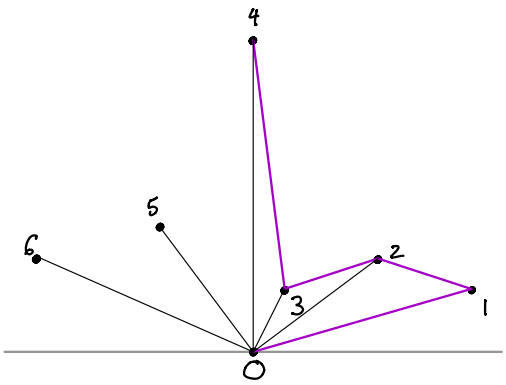
Actually we don't need to compute angles. The line-side-test can be used for this instead. For example look at what happens after we've found P_s and P_t. We process possibilities for the next point in any order. Say we start from a in the figure. Then we try b, and note that b is on the right side of 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

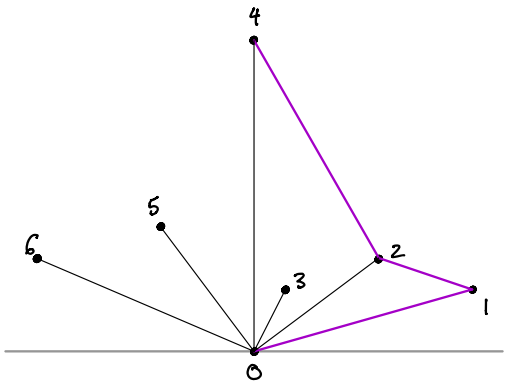
We can convert this into an $O(n \log n)$ algorithm with a slight tweek. Instead of processing the points in an arbitrary order, we process them in order of increasing angle with respect to point p_s.

Let's relabel the points so that P_0 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.

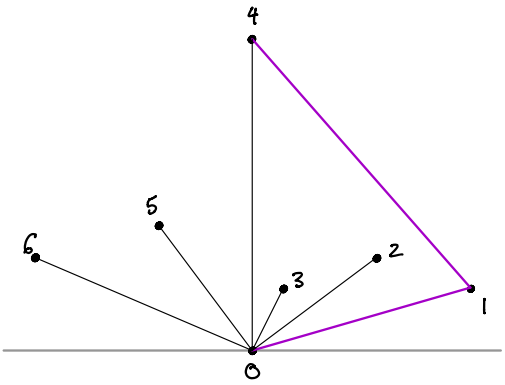
The Graham Scan works as follows. We maintain a "chain" of points that starts with $P_0, P_1, \ldots$. This chain has the property that each step is always a left turn with respect to the previous element of the chain. We try to extend the chain by taking the next point in the sorted order. If this has 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. Here's an example of the algorithm.



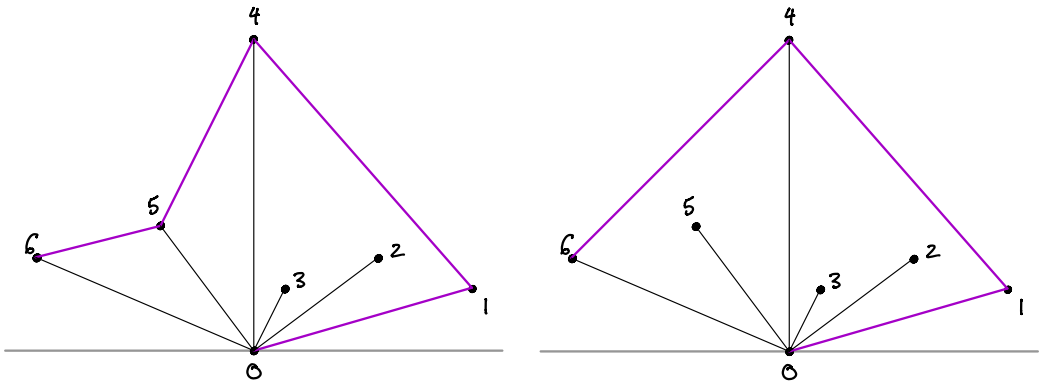
At this point we've formed the chain P_0, P_1, P_2, P_3, P_4. But the last step (from P_3 to P_4) is a right turn. So we delete P_3 from the chain. Now we have:



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.

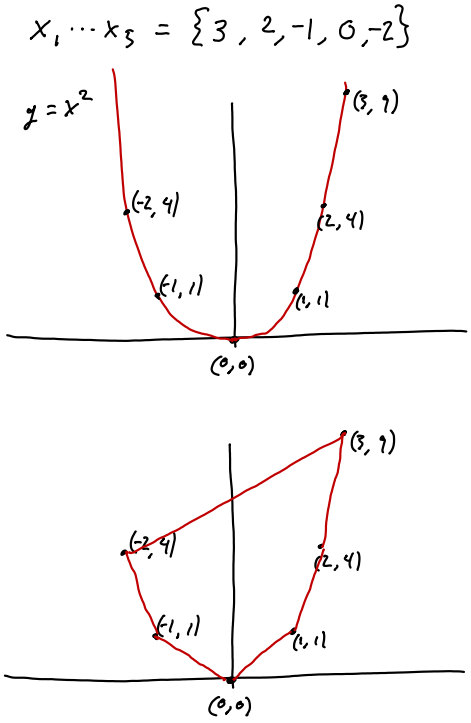
Complete ocaml code for the graham scan is at the end of these notes.

3.3 Lower bound for computing the convex hull

Suppose the input to a sorting problem is $X_1, \ldots, X_n$. Consider computing the convex hull of the following set of points:

$$(X_1, X_1^2), \ldots, (X_n, X_n^2)$$

All of these points are on the convex hull (they're on a parabola). Thus they are returned in the order they appear along the parabola. No matter which convex hull algorithm is used, the points can be reflected and/or cyclically shifted so that their x coordinates are in sorted order. Thus, they can be sorted by computing a convex hull followed by $O(n)$ additional work. Thus any comparison based convex hull algorithm must make $\Omega(n \log n)$ comparisons. The figure below illustrates this phenomenon.



3.4 Ocaml code for the Graham Scan convex hull algorithm

```
let sq x = x * x
let (--) (x1,y1) (x2,y2) = (x1-x2, y1-y2)
let cross (x1,y1) (x2,y2) = (x1*y2) - (y1*x2)
let len2 (x,y) = (sq x) + (sq y)

type side = LEFT | ON | RIGHT

let line_side_test p1 p2 p3 =
  let cp = cross (p2--p1) (p3--p1) in
  if cp > 0 then LEFT else if cp < 0 then RIGHT else ON

let graham_convex_hull points =
  (* This algorithm takes as input a list of points, and returns a
     list of points that is the convex hull (in clockwise order) of
     the input.  Duplicates are allowed in the input, and the output
     contains no three colinear points.

     graham_convex_hull [(0,0);(0,2);(2,2);(2,0);(1,1);(1,2);(1,2)]
     returns this list: [(0,2) (2,2) (2,0) (0,0)]
 *)

  let base = List.fold_left min (List.hd points) points in
  let points = List.filter (fun pt -> pt <> base) points in

  let compare_points pi pj =
    match line_side_test base pi pj with
      | ON -> compare (len2 (base -- pi)) (len2 (base -- pj))
      | LEFT -> -1
      | RIGHT -> 1
  in

  let points = List.sort compare_points points in

  let rec scan chain points =
    let (c1,c2,chainx) = match chain with
      | c1::((c2::_) as chainx) -> (c1,c2,chainx)
      | _ -> failwith "chain must have length at least 2"
    in
    match points with [] -> chain
      | pt::tail ->
        match line_side_test c2 c1 pt with
          | ON ->
            if len2 (pt--c2) > len2 (c1--c2)
            then scan (pt::chainx) tail
            else scan chain tail
          | LEFT -> scan (pt::chain) tail
          | RIGHT -> scan chainx points
  in

  match points with
    | (p1::((_::(_::_)) as rest)) -> scan [p1;base] rest;
    | _ -> List.rev (base::points)
```