Convex Hull

Def The convex hull of a set S is the smallest convex set containing S. We will denote this by $\mathrm{conv}(S)$.

Sometimes "convex hull" really refers to the boundary of the set $\mathrm{conv}(S)$. We will denote that by $CH(S)$.

Def A polyhedral set in E^d is the intersection of a finite set of closed half-spaces. It is convex. A bounded polyhedral set is called a convex d-polytope.

Thm The convex hull of a finite set of points in E^d is a convex d-polytope. Conversely, a convex d-polytope is the convex hull of a finite set of points.

Def A convex d-polytope is a k-simplex if it is the convex hull of $k+1$ affinely independent points.

Def A convex polytope is often described by its boundary, which consists of faces (themselves convex sets). Sometimes the faces of dimension $d-1$ of a d-polytope are called facets.

Def A d-polytope is called simplicial if each of its facets is a simplex (a $d-1$ simplex).

Problem Statement

Suppose $S = \{q_1, \ldots, q_N\} \subset E^d$

Problem 1: Construct $CH(S)$, with full adjacency information.

Problem 2: Identify the points of S that are vertices of $CH(S)$.

Note: In 2D, problem 1 amounts to a list of the vertices of $CH(S)$ in order as one traverses the boundary of $conv(S)$.

Clearly Problem 1 is asymptotically at least as hard as Problem 2. In particular, if we can solve Problem 1, then we can simply output the vertices to also solve Problem 2.

This is known as a reduction, requiring at most linear reduction time. We write: Problem 2 $\propto_N$ Problem 1

& say: "Problem 2 is linear-time transformable to Problem 1."

> The concept of a reduction is extremely important. First, it allows one to solve a problem by really solving a different problem. Second, it provides a measure of relative difficulty.

Here is an important reduction:

> Thm Sorting is linear-time transformable to 2D Convex Hull (problem 1).
>
> sort $\propto_N$ Problem 1

Corollary: Finding the convex hull of N points in the plane requires $\Omega(N \log N)$ time. "Ω" means "at least", sort of pf

Given N real numbers $x_1, \ldots, x_N$.

CAWLOG all x_i are positive.

Let q_i be the point $q_i = (x_i, x_i^2)$

If we now compute $CH(\{q_i\})$ we get an ordered list of vertices, sorted by their x coordinates.

[Why? Because $y = x^2$ is a monotone function for $x > 0$.]

$\therefore$ SORT $\propto_N$ CH

Note: This then also proves in higher dimensions that Problem 1 is $\Omega(N \log N)$.

It turn out that in 2D one can also do the reduction in the other direction. Therefore Problem 1 is $\Theta(N \log N)$. In other words, convex hull & sorting are equivalent (transformable) problems. Problem 2 is also $\Theta(N \log N)$, as it turns out.

For d dimensions, see Chazelle's 1993 paper "An Optimal Convex Hull Algorithm in Any Fixed Dimension" in Discrete & Computational Geometry, Vol 10, pp. 377–409. That provides an algorithm with time complexity $O(N \log N + N^{\lfloor \frac{d}{2} \rfloor})$.

Convex Hull in 2D

Def A point p of a convex set C is an extreme point if there do not exist two points $a, b \in C$ such that p lies on the open line segment between a & b.

Note: Suppose S is a finite set of points, and consider the set E of extreme points of the convex hull $conv(S)$.
Then E is the smallest subset of S such that $conv(E) = conv(S)$.
Moreover, E consists precisely of the vertices of $CH(S)$.

This yields a rather straightforward (but very inefficient) method for computing $CH(S)$, for finite point sets S:

1. Determine E
2. Sort the points of E to form a convex polygon.

Thm	A point p fails to be an extreme point of a planar convex set C iff it lies within(*) a triangle whose vertices are in C. (*) specifically: Interior to the triangle, or on one of its edges, but not its vertices.

This gives us an $O(N^4)$ algorithm for determining E from S: we test each point of S against all possible triangles.

To sort the points we use our previous point-location ideas:

Thm A ray emanating from an interior point of a bounded convex set intersects the boundary of that set in exactly one point.

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

So, we first find a point q interior to E.
[Again, the way we do this is fix two adjacent vertices, then scan around the remaining vertices until we have 3 affinely indep ones, i.e., not all on the same line. Pick q as the centroid of the triangle

This requires $O(N)$ time.

Next we sort the vertices on angle about q.
To do this we either explicitly compute angles or consider cross products of the form $(p_i - q) \times (p_j - q)$ to determine which angle is greater.

This requires $O(N \log N)$ time.

Notice how SORT enters the picture!

But $O(N^4)$ is yucky overall.
Our complexity results suggest $O(N \log N)$.

Graham's Scan

The reason we get $O(N^4)$ is that we test a point against all triangles to decide whether it is an extreme point. This is not necessary. There is more structure.

Idea: Do the sort first.
Then use that structure to find extreme points in linear time.

Let's choose our coord system, so q = origin = internal point. (Actually, could also q be an extreme pt.)

We now sort the points of S using a dictionary ordering:

$$p_1 < p_2 \text{ iff } (1)\ \arg(p_1) < \arg(p_2)$$
$$\text{or } (2)\ \arg(p_1) = \arg(p_2) \text{ and } |p_1| < |p_2|$$

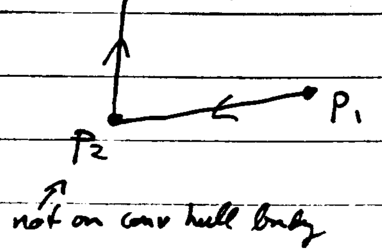
As before we use cross products to compute (1) (& quadrant information). If we need to compute (2), then we really just need to compare x or y coords to decide which $|p_i|$ is greater, since the points are colinear. (so no sqrts required)

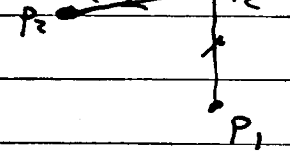
In order to access the sorted vertices quickly we link them into a doubly-linked circular list.

We then start scanning at one vertex known to be extreme, say at the rightmost vertex with the lowest y-coord.

We scan in ccw order, examining triples of consecutive vertices, eliminating those for which the angle is "reflex" (or "right turn").

Eg. eliminate p_2 keep p_3





If we keep p_2 then we advance the scan and consider $p_2 p_3 p_4$,
If we eliminate p_2 then we back up the start vertex & consider $p_0 p_1 p_3$.

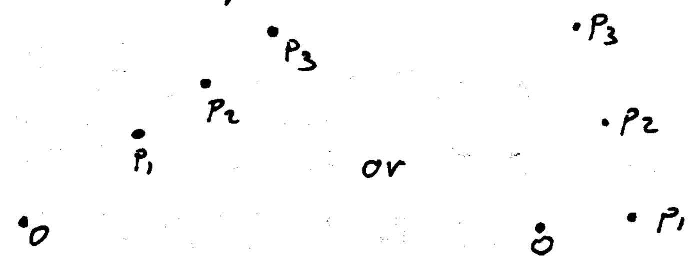
Deciding whether to keep or eliminate p_2 is based on a cross product as usual, namely the sign of the 2D cross product

$$(p_1 - p_2) \times (p_3 - p_2)$$

pos $\leftrightarrow$ eliminate
neg $\leftrightarrow$ keep

<u>Note:</u> If zero then $p_1 p_2 p_3$ are colinear.
Since the origin is an interior point & since the p_i are sorted we must have

(Btw, this means that if several points have the same polar angle, then we could eliminate right at sort time all but the furthest one.)



So we should eliminate p_2.

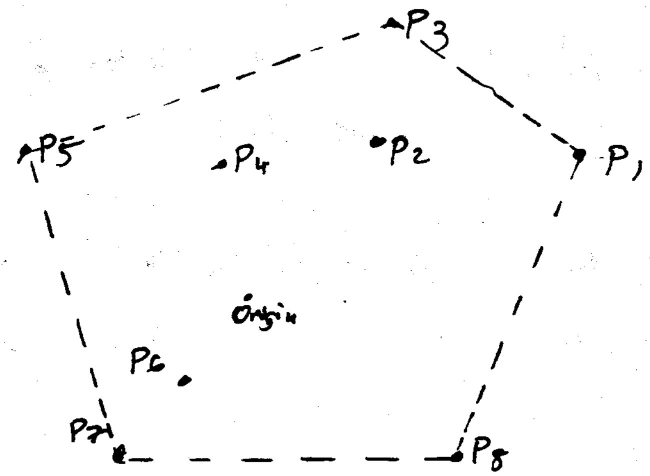
<u>Stop</u> when p_2 = Start after a complete cycle (i.e. keep track of whether every vertex has been teste[d])

<u>Complexity:</u> The scan runs in $O(N)$ time since it either advances one vertex or eliminates one vertex on each step. [so $< 2N$ steps]
Sorting takes $O(N \log N)$ time.
So overall: $O(N \log N)$ time
$O(N)$ space.

$\therefore$ optimal.

(Again, notice the intimate use of sorting.)

Ex:



Progression of algorithm: (The START vertex is p_1 in the example

Step	Current Sorted List	Triple under consideration
1.	$p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8$	$p_1 p_2 p_3 \Rightarrow$ elim p_2
2.	$p_1 p_3 p_4 p_5 p_6 p_7 p_8$	$p_8 p_1 p_3 \Rightarrow$ keep p_1
3.	$p_1 p_3 p_4 p_5 p_6 p_7 p_8$	$p_1 p_3 p_4 \Rightarrow$ keep p_3
4.	$p_1 p_3 p_4 p_5 p_6 p_7 p_8$	$p_3 p_4 p_5 \Rightarrow$ elim p_4
5.	$p_1 p_3 p_5 p_6 p_7 p_8$	$p_1 p_3 p_5 \Rightarrow$ keep p_3
6.	$p_1 p_3 p_5 p_6 p_7 p_8$	$p_3 p_5 p_6 \Rightarrow$ keep p_5
7.	$p_1 p_3 p_5 p_6 p_7 p_8$	$p_5 p_6 p_7 \Rightarrow$ elim p_6
8.	$p_1 p_3 p_5 p_7 p_8$	$p_3 p_5 p_7 \Rightarrow$ keep p_5
9.	$p_1 p_3 p_5 p_7 p_8$	$p_5 p_7 p_8 \Rightarrow$ keep p_7
10.	$p_1 p_3 p_5 p_7 p_8$	$p_7 p_8 p_1 \Rightarrow$ keep p_8

output this as CH(S)

We will look at some other algorithms for a couple reasons
First, we would like to obtain some insights that generalize to higher than 2D.
Second, just as some sorting algorithms are fast even if not worst-case optimal, so too we should expect that of convex hull algorithms.

Some other reasons: permit incremental addition of points to S, allow for parallelism.

Jarvis's March

This algorithm focuses on edges rather than vertices.

> **Th^m** The line segment l defined by two points of S is an edge of conv(S) iff all points of S lie in one half-space determined by l.

This gives a simple $O(N^3)$ alg:

There are $O(N^2)$ possible line segments.
For each we test all N points of S to decide whether the line segment is extremal.
We then hook the resulting line segments together.

Here is a better version:

We start with a point on the hull bndry.

At each stage we add a new point, i.e., a new hull edge, by picking the point with the smallest polar angle, using the current point as origin

We start the march off with the "lexicographically lowest" point of S (specifically the point satisfying

$$\max_x \min_y \{(x,y) \in S\}).$$

The march takes(*) us to the "lexicographically highest" point of S ($\min_x \max_y$), while traversing ccw around the body of conv(S).

Similarly, a symmetric march will take us back down the other side of conv(S).

We then hook up the two march results.

<u>Complexity:</u> $O(hN)$ time

where h = # of vertices of S actually in conv(S)

So worst-case $O(N^2)$.

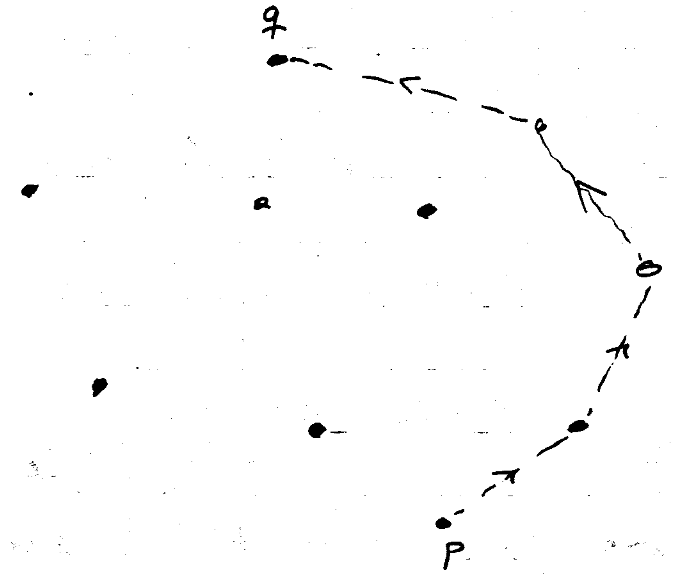
But, in practise may run faster.

[& $O(N)$ space].

The trick is that at each hull vertex the next vertex may be determined in linear time (in the worst case we scan through all N & pick the one of least polar angle).

(*) We can compute these two points ahead of time in $O(N)$ time & simply start & stop the marches at these points, rather than let the first march go beyond the top & get confused.

Ex



Start first march at p, end at q.
Start second march at q, end at p.

Quickhull

Recall Quicksort:

Given array of N numbers.

1. Split array in "half"(*) st all numbers in first half are no larger than all numbers in second half: move two pointers towards each other from extreme ends of array, exchanging elements whenever the inequality condition is violated.
2. Apply recursively to each of the two half arrays, until unit size.

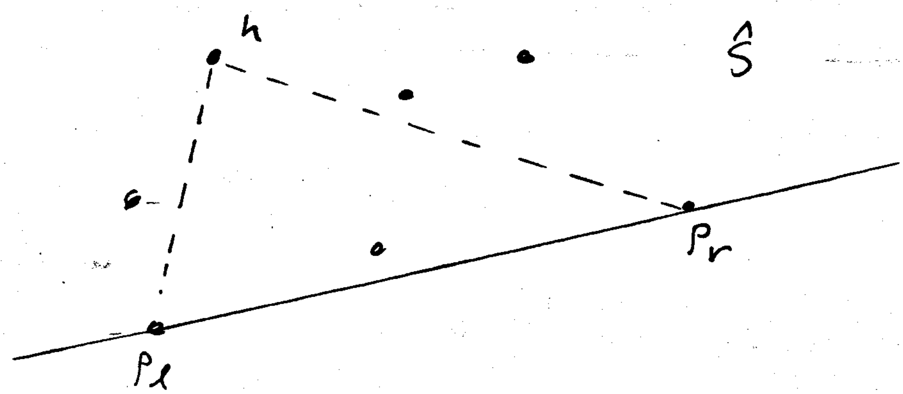
Expected time: $O(N \log N)$ for random distributions
Worst Case: $O(N^2)$

(*) May not split exactly in half: the pointers move one at a time, changing roles as to which one moves whenever there is an exchange of elements. Array is split at the place pointers collide. If sorted initially, the split will be $[N-1, 1]$. Hence $O(N^2)$ worst-

Quickhull

First, [redacted] pick l & r to be points with min & max x value. Split S into $S^{(1)}$ & $S^{(2)}$ using the line through l & r (Note: $S^{(1)} \cap S^{(2)}$ consists of all points in S on this line.)

The recursive step expects a set $\hat{S}$ and two points p_l, p_r st $\hat{S}$ lies wholly to one side of the line $\overline{p_l p_r}$ (incl the line), and indeed includes all points of S on that side of the line. E.g.



The algorithm finds a point $h \in \hat{S}$ st the triangle $p_l h p_r$ has maximum area.
(If there are several such triangles pick the h that maximizes the angle at p_l.)

→ Then $h \in \text{conv}(S)$, since no points of S lie "above" h as measured by the normal to $\overline{p_l p_r}$.

We then construct two sets $\hat{S}^{(1)}$ & $\hat{S}^{(2)}$ consisting of those points of $\hat{S}$ [redacted] that are not in the interior of the triangle (points interior to the triangle clearly cannot be element of $CH(S)$).

$\hat{S}^{(1)}$ consists of points to one side of the line through p_ℓ & h.
$\hat{S}^{(2)}$ – – – – – – – – – pr & h.

We call the algorithm recursively on

$$(\hat{S}^{(1)}, p_\ell, h) \quad \& \quad (\hat{S}^{(2)}, h, pr).$$

<u>Note:</u> by construction of h $\hat{S}^{(1)} \cap \hat{S}^{(2)}$ consists only of h.

We can think of $\hat{S}^{(1)}$ as all points lying on or to the left of the directed edge (p_ℓ, h)

Similarly $\hat{S}^{(2)}$ consists of all points of $\hat{S}$ lying on or to the left of the directed edge (h, pr)

[Given that $\hat{S}$ lies on or to the left of (p_ℓ, pr).]

<u>Complexity:</u> At each stage determining h requires $O(N)$ time. Similarly, determining $\hat{S}^{(1)}$ & $\hat{S}^{(2)}$ requires $O(N)$ time.

$\therefore$ Worst-case running time is $O(N^2)$.
However, if the splitting process is nicely behaved can get $O(N \log N)$.

Note: <u>Termination of recursion:</u> If $\hat{S} = \{p_\ell, pr\}$. In that case return $\{p_\ell, pr\}$. The previous calls concatenate returned value (while removing duplicate h values).

Formally: Quickhull $(\hat{S}, p_\ell, p_r)$:

If $\hat{S} = \{p_\ell, p_r\}$ then RETURN the list $(p_\ell\ p_r)$;

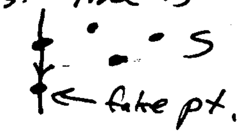
ELSE: determine $\hat{h}$, $\hat{S}^{(1)}$, $\hat{S}^{(2)}$.

$vert^{(1)} \leftarrow$ Quickhull $(\hat{S}^{(1)}, p_\ell, h)$

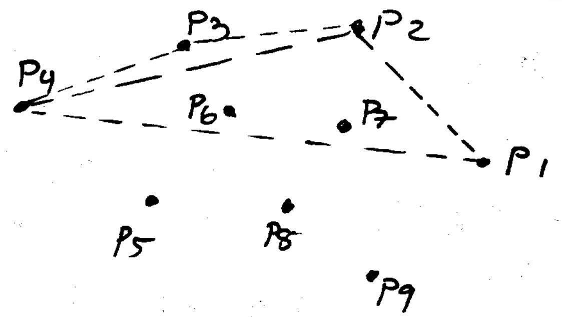
$vert^{(2)} \leftarrow$ Quickhull $(\hat{S}^{(2)}, h, p_r)$

RETURN append $(vert^{(1)}, vert^{(2)} - \{h\})$

Can start the whole process off either by explicitly finding two extreme points as we did above or creating a fake pt st the first line is a vertical line & S lies wholly to one side of it:



Ex S:



let $\hat{S} = \{p_1, p_2, p_3, p_4, p_6, p_7\}$

$\bar{S} = \{p_1, p_4, p_5, p_8, p_9\}$

Calls:	Computation	Returned Values
$Q(\hat{S}, p_4, p_1)$	$h \leftarrow p_2$, $S^{(1)} \leftarrow \{p_4, p_3, p_2\}$, $S^{(2)} \leftarrow \{p_2, p_1\}$	
$Q(S^{(1)}, p_4, p_2)$	$h \leftarrow p_3$, $S^{(1,1)} \leftarrow \{p_4, p_3\}$, $S^{(1,2)} \leftarrow \{p_3, p_2\}$	
$Q(S^{(1,1)}, p_4, p_3)$	none	$(p_4\ p_3)$
$Q(S^{(1,2)}, p_3, p_2)$	none	$(p_3\ p_2)$
		$(p_4\ p_3\ p_2)$
$Q(S^{(2)}, p_2, p_1)$	none	$(p_2\ p_1)$
		$(p_4\ p_3\ p_2\ p_1)$

Similarly $Q(\bar{S}, p_1, p_4)$ returns: $(p_1\ p_9\ p_5\ p_4)$

And so the initial call to $Q(S, -, -)$ combines these results to get

$(p_4\ p_3\ p_2\ p_1\ p_9\ p_5\ p_4)$

Mergehull $\longrightarrow$ Divide and Conquer Approach.

Given an input set of points, S, suppose we split S into two roughly same size sets S_1 & S_2.

If we now recursively compute $CH(S_1)$ & $CH(S_2)$ how long does it take to compute $CH(S)$ from this data?

We know $CH(S_1 \cup S_2) = CH(CH(S_1) \cup CH(S_2))$

So this leads us to consider the following

PROBLEM: Given two convex polygons P_1 & P_2, find the convex hull of their union.

We will shortly exhibit an algorithm that solves this problem in time $O(N)$, where N = total number of points. Consequently, we get the following recurrence for the running time of Mergehull: $T(N) \le 2T(\frac{N}{2}) + O(N)$

$$\Rightarrow T(N) \le O(N \log N)$$

Def A supporting *line* of a convex polygon P is a straight line l passing through a vertex of P such that the interior of P lies entirely on one side of l.

Note: If P_1 & P_2 are two polygons (possibly overlapping) such that one is not entirely contained in the other then P_1 & P_2 share common supporting lines. At least 2; not more than 2 min(# of P_1 vertices, # of vert

Here is one way to compute the supporting lines of P_1 & P_2:

Compute $CH(P_1 \cup P_2)$

Now scan the vertex list of $CH(P_1 \cup P_2)$. Any pair of consecutive vertices in this list arising from both P_1 & P_2 (i.e., one vertex is in P_1's vertex list, the other in P_2's) defines a supporting line.

<u>Note:</u> Given that we can compute $CH(P_1 \cup P_2)$ in time $O(N)$, this algorithm also runs in time $O(N)$, since the scan is linear time

---

Here is an algorithm for computing $CH(P_1 \cup P_2)$, given convex P_1 & P_2

There are two main steps:

1. (we will elaborate on this shortly)
   Create a list of vertices sorted around some point p internal to $CH(P_1 \cup P_2)$ that includes all the vertices of $conv(P_1 \cup P_2)$, plus possibly some extraneous ones.

2. Use the scanning portion of Graham's Scan to eliminate from this vertex list all vertices interior to $CH(P_1 \cup P_2)$.

Notice that this algorithm looks a lot like Graham's Scan. In particular Step 2 is the same & runs in time $O(N)$. Step 1 is different in that we will use the already sorted nature of P_1 & P_2 to create a sorted list in time $O(N)$, rather than $O(N \log N)$

So let's expand Step 1:

1.1: $p \leftarrow$ some point internal to P_1

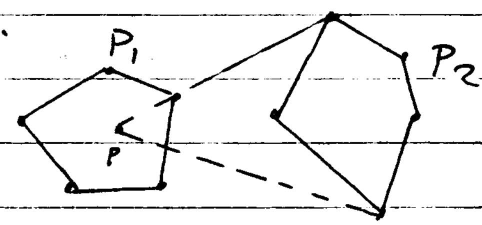
1.2: If p is internal to P_2

Then /* We know the vertices of P_1 are sorted by angle about p, & the vertices of P_2 are sorted by angle about p */

1.3: Merge the vertex lists of P_1 & P_2 into a single list sorted by angle about p. Return the list

Else /* P_2 lies in a wedge when viewed from p:

eg.



*/

1.4: Determine this wedge and discard all the vertices of P_2 that lie strictly inside the wedge facing p /* All such vertices are interior to $CH(P_1 \cup P_2)$. */

1.5: The remaining vertices of P_2 are sorted by angle about p. Merge these with the vertices of P_1 to create a single list sorted by angle about p. Return the list.

Complexity	Step	Complexity	Comment
$O(N)$ time overall	1.1	$O(N)$	Usual reason.
	1.2	$O(N)$	Since P_2 is [redacted] a polygon
	1.3	$O(N)$	merge of two $O(N)$ lists
	1.4	$O(N)$	Scan vertices of P_2. Find extreme angles wrt p
	1.5	$O(N)$	merge of two $O(N)$ lists

Dynamic Convex Hull

So far we have looked at Convex Hull for static sets S. In some cases we may want to add new points to S or remove old ones, then update CH(S). How to do this efficiently?

Complexity: 1. There exists an algorithm that allows addition of new points with an update time of $\Theta(\log N)$

2. There exists an algorithm that allows both addition and deletions, with an update time of $O(\log^2 N)$.

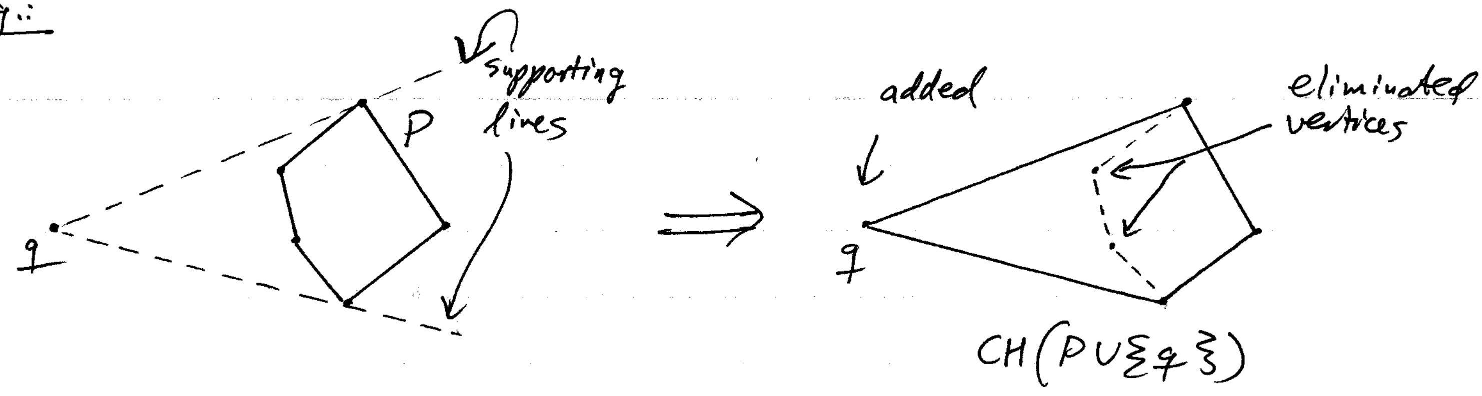
Let's look at 1, the case of dynamic additions.

Here is the basic subproblem:

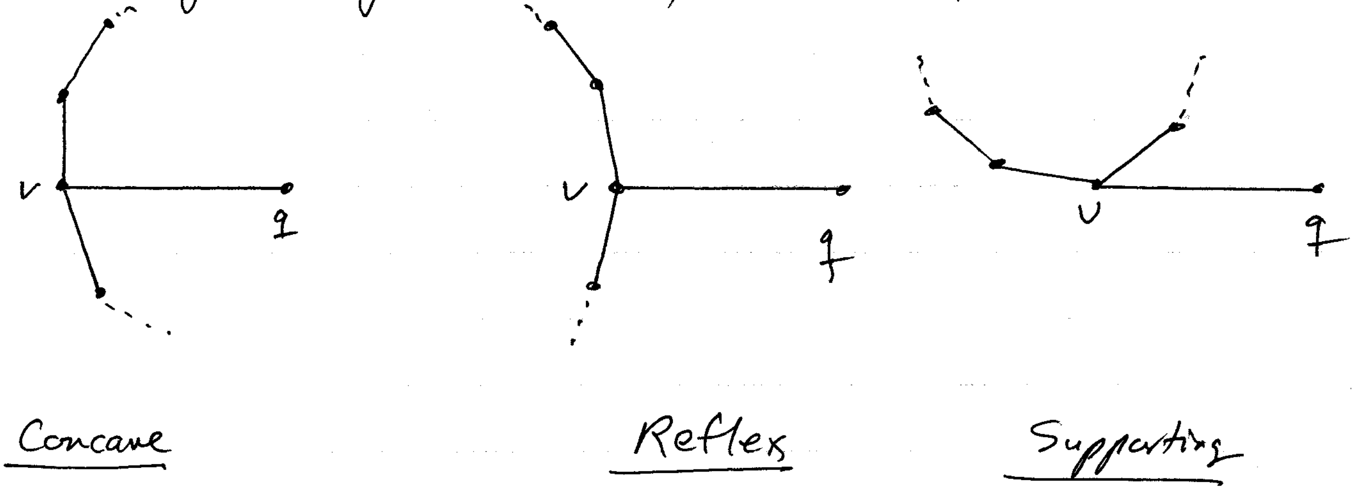
> Given a convex polygon P and a new point q, how do we form $\mathrm{conv}(P \cup \{q\})$ quickly?

The algorithm operates by finding the supporting lines of P that contain q, if these exist.
They do not exist precisely when q is interior to P, in which case $CH(P \cup \{q\}) = P$. Otherwise, we eliminate from P all vertices inside the wedge anchored at q, and add in the edges from the supporting lines, i.e., q adj.
The result is $CH(P \cup \{q\})$.

E.g.:



We classify vertices of P relative to q as follows:

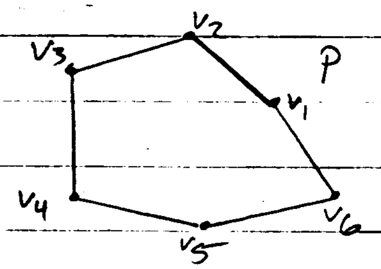


(Note: we assume no three vertices of P are colinear, Hence these classifications can be computed in constant time.)
[or could use the implicit ordering of the vertices of P.]

The trick in the algorithm for computing CH(P ∪ {q}) is to find the two supporting vertices of P relative to q (if they exist) quickly.

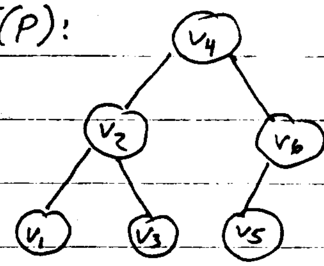
We will then maintain the polygon P as a height-balanced tree that supports add & delete operations in O(log N) time, with the vertices sorted in CCW order about P. [Or could use any similar datastructure.]

Ex



Tree representation
(sorted; balanced)

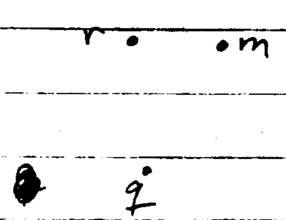
T(P):



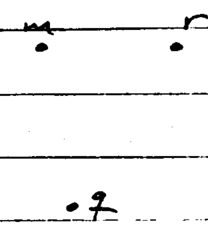
Different sorts are possible of course.
Also note that the "minimum" element in the tree and the "maximum" element are adjacent in the polygon. So, depending on where we break the circular list nature of P, we will have different minimum + maximum elements in T(P). But, it doesn't matter. Pick one of these.

Now, let r be the root vertex of T(P) and m the min vertex of T(P). (i.e., v_4 & v_1 in the example above.)

We will be concerned with the relationship between r, m, & q: specifically, we are interested in the sign of the cross-product $(m-q) \times (r-q)$.
There are two cases really:



positive cross product
(or zero)
?



negative cross product

The following 8 cases are of interest (& cover all possibilities)

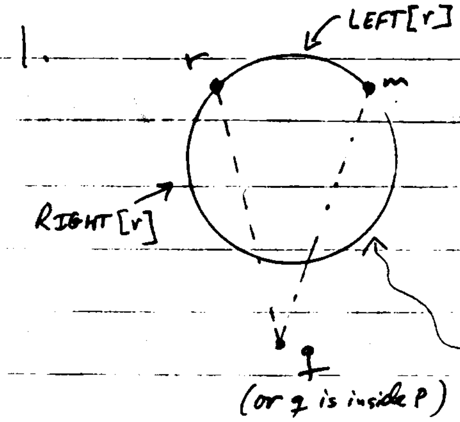
Case	Cross Product Sign	Classification of m	Classification of r
1	+	concave	concave
2	+	concave	not concave
3	+	not concave	reflex
4	+	not concave	not reflex
5	—	reflex	reflex
6	—	reflex	not reflex
7	—	not ~~concave~~ reflex	concave
8	—	not ~~concave~~ reflex	not concave

The idea is to use these local classifications to search T(P) efficiently, in order to locate both vertices of support.

We can represent the tree structure imposed on the polygon by a circle, broken between the min & max elements of the tree/polygon. Also let RIGHT[r] & LEFT[r] denote the right & left subtrees rooted at r.

Consider the cases:

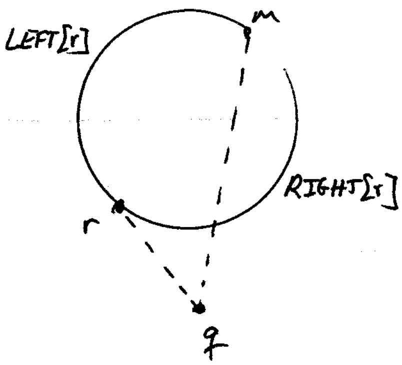
1.



In this case we know that we must recursively search the right subtree for both support vertices (which may not exist if q turns out to be inside P)

circle represents the polygon P, i.e., vertices are sorted, beginning at m & running counterclockwise.

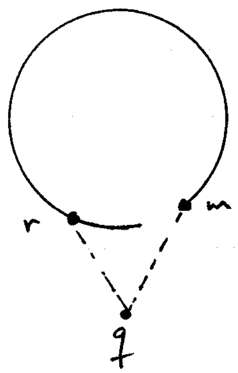
2.



In this case q must lie outside of P, and then both support vertices exist. One lies in the right subtree; the other is either r itself or lies in the left subtree.

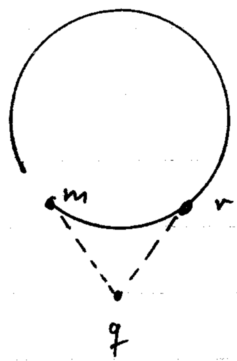
Cases 3, 5, & 7 are variations on 1:

3:



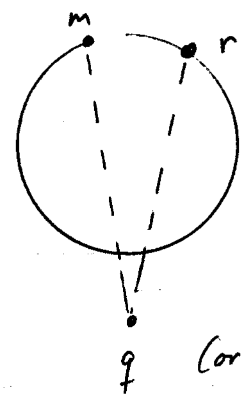
Both support vertices lie in LEFT[r].

5:



Both support vertices lie in RIGHT[r].

7:

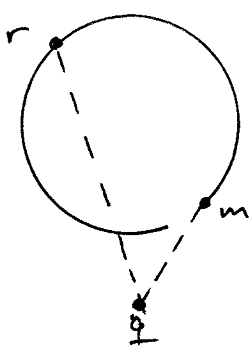


(or q is inside P)

Both support vertices, if they exist, lie in LEFT[r].

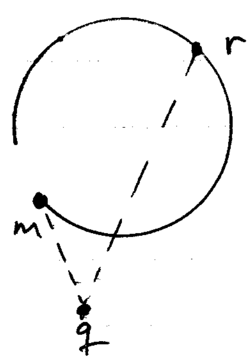
Cases 4, 6, 8 are variations on 2:

4:



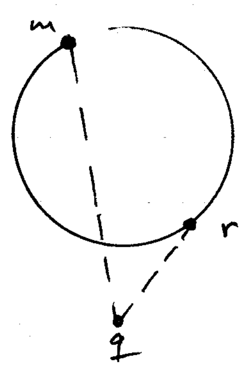
One support vertex in LEFT[r], the other in RIGHT[r] ∪ {r}.

6:

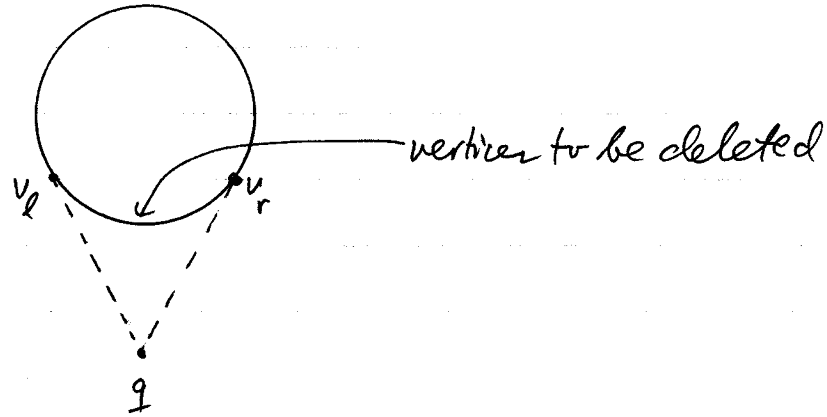


One support vertex in RIGHT[r], the other in LEFT[r] ∪ {r}.

8:



One support vertex in LEFT[r], the other in RIGHT[r] ∪ {r}, Let v_ℓ be the support vertex defining the left support line from q, and let v_r be the support vertex defining the right support line.

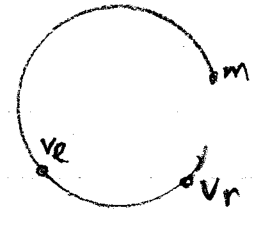


We must now delete the vertices "between" v_ℓ & v_r, then add in q. The result is $CH(P \cup \{q\})$ (or rather conv ....).

In order to delete the vertices we think of the balanced tree as defining a concatenatable queue and use the efficient SPLIT & SPLICE operations.

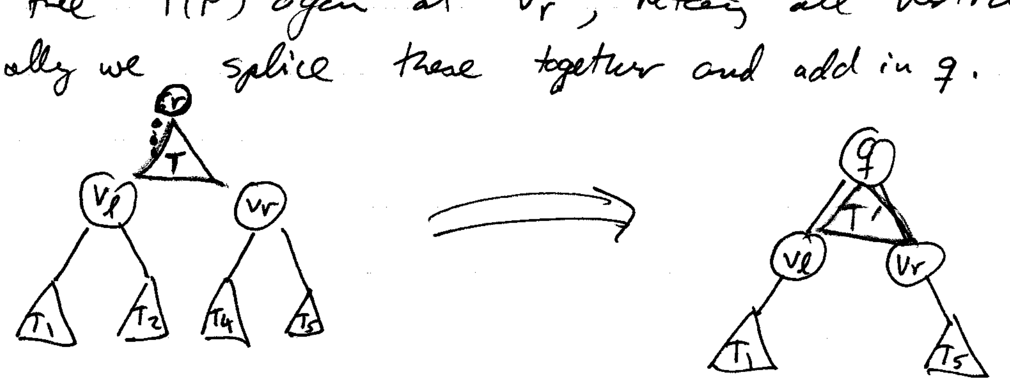
There are two cases:

i.e.,



1) ~~If~~ $v_\ell < v_r$ in the tree ordering, then we split the tree $T(P)$ once at v_ℓ, retaining all vertices $\le v_\ell$, and we split the tree $T(P)$ again at v_r, retaining all vertices $\ge v_r$. Finally we splice these together and add in q.

E.g.

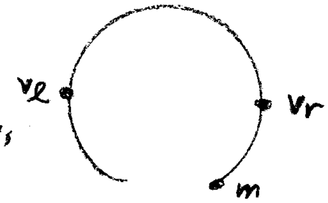


These pictures aren't quite right. Really we need to split using these code, not just drop subtrees.

(plus rebalancing at each step of course)

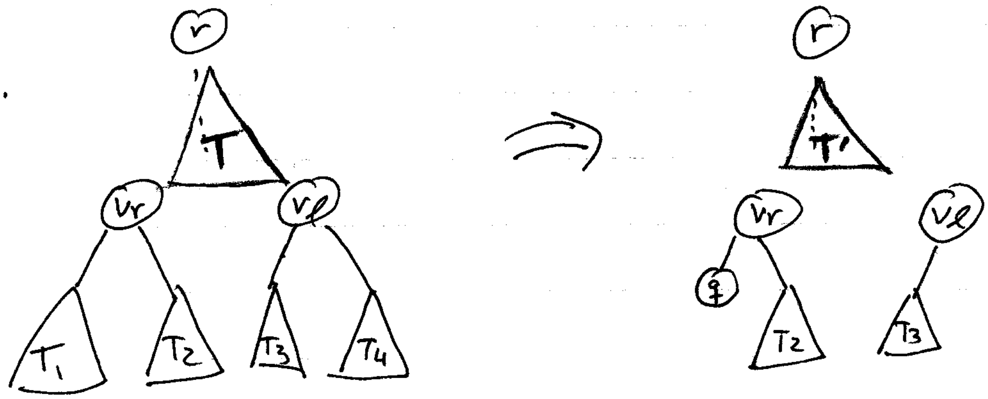
(depending on the location of (r), some or all of the ":" nodes get removed too.

i.e.,



2) If $v_\ell > v_r$ in the tree ordering, then we again split at v_ℓ & v_r, but this time we don't need to splice. We do add in q.

E.g.



here too, one may need to delete ancestors of v_r & v_ℓ

(plus rebalancing at each step)

<u>Complexity:</u>

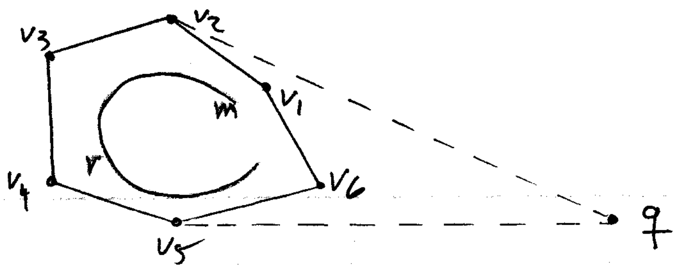
Finding v_r & v_ℓ is accomplished by at most two searches in $T(P)$, starting at r and traversing at most the height of the tree. Hence $O(\log N)$.

Deleting the vertices between v_ℓ & v_r requires at most a constant number of SPLIT, SPLICE, and INSERT operations, each of which requires $O(\log N)$ time. $\therefore$ $O(\log N)$ total time

---

Note: The tree pictures above are just examples. Other possibilities exist.

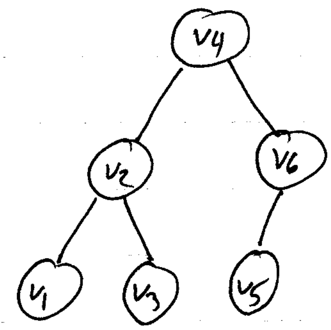
Ex P:



Vertex classification wrt q:

v_1	v_2	v_3	v_4	v_5	v_6
reflex	supporting	concave	concave	supporting	reflex

Suppose T(P) is



Then the search proceeds as follows:

SEARCH for v_ℓ & v_r in T(P)

$$\left.\begin{array}{l} r = v_4 \\ m = v_1 \\ \text{cross-product} > 0 \end{array}\right\} \text{Case } 4$$

LEFTSEARCH (see ⊛)

RIGHTSEARCH (see ⊛)

SEARCH for v_ℓ in RIGHT[v_4]

$$\left.\begin{array}{l} r = v_6 \\ m = v_5 \\ \text{cross-product} < 0 \end{array}\right\} \Rightarrow \text{Case } 8$$

alternatively: r is reflex, so (see ⊛) LEFTSEARCH(LEFT[v_6])

↓

SEARCH for v_ℓ in LEFT [v_6]

$r = v_5$ which is supporting, so search ends

$v_\ell = v_5$

SEARCH for v_r in LEFT [v_4]

$r = v_2$ which is supporting, so search ends

$v_r = v_2$

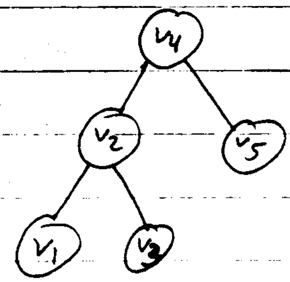
⊛ Actually, just do LEFTSEARCH in T[v_6] and RIGHTSEARCH in T[v_2]. No need to keep doing case analysis. (See P&S for details.)

Delete vertices between v_5 & v_2, and add vertex q:

Note $v_\ell > v_r$, so two splits plus an insertion (of q) are required.

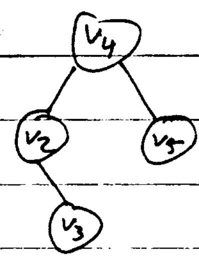
1. $\text{Split}_1^{T(P)}$ at $v_\ell = v_5$, retaining all vertices $\le v_\ell$:

resulting tree:



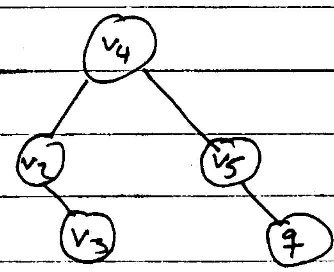
2. $\text{Split}_1^{\text{result}}$ at $v_r = v_2$, retaining all vertices $\ge v_r$:

resulting tree:



3. Insert q into the result of step 2:

Final tree:



(which is still balanced)

The resulting vertex order is: v_2 v_3 v_4 v_5 q which does indeed specify $CH(P \cup \{q\})$.

Applications of Convex Hull

Problem: Set Diameter: Given N points in the plane, find two that are furthest apart. (called the ~~diameter~~ of the set)

This problem arises for instance in clustering algorithms, where one is trying to partition a set of points into some number of clusters, st that the cluster diameters are small.

There is an obvious $O(N^2)$ algorithm: examine all pairs of points.

But what is an optimal algorithm?

The following theorem says it can't be better than $\Theta(N \log N)$:

> Thm The computation of the diameter of a finite set of N pts in E^d requires $\Omega(N \lg N)$ time (in the algebraic computation-tree model).
> Pf works by reducing set disjointness (of real numbers) to SET DIAMETER in 2D: maps sets to points on the circle

The connection to convex hull is given by:

> Thm The diameter of a set is equal to the diameter of its convex hull.

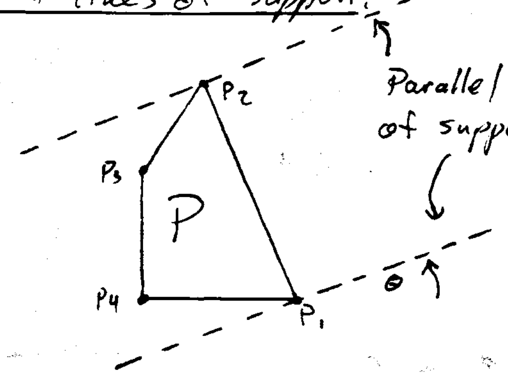
And then we have the following help:

> Thm The diameter of a convex figure (in 2D) is the greatest distance between parallel lines of support.

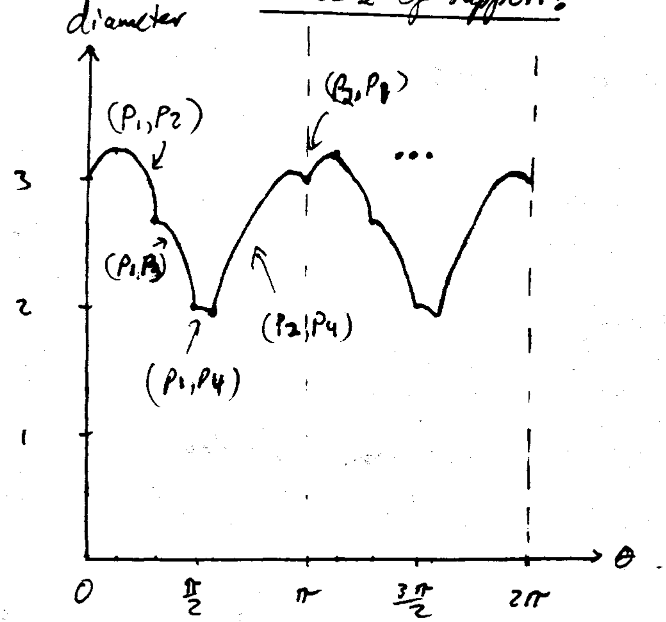
(This is the diameter function of an object, for instance, the separation of a parallel jaw gripper as it maintains contact with an object being rotated.)

Ex

P & lines of support:



Separation between the parallel lines of support:



(Symmetric, i.e., period of π)

Indeed we see that the max occurs for support contact either with the pair of points (p_1,p_2) or the pair (p_2,p_4) [We've exaggerated the difference between the maxima a little in the graph of the diameter function.] The max occurs for (p_1,p_2).

(This is similar to PAS's antipodal pairs algorithm)

This gives us an $O(N)$ algorithm for determining the maximum diameter of a convex polygon:

We simulate the rotation of a parallel jaw gripper around P. Specifically, we start with two vertices of P with extreme y-coords and let $\theta = 0$. Suppose the two vertices are p_i & p_j. Consider the angles θ_i & θ_j of the edges $\overline{p_i p_{i+1}}$ & $\overline{p_j p_{j+1}}$. We advance θ to the min of θ_i & θ_j, then switch the appropriate support vertex (either $p_i \to p_{i+1}$ or $p_j \to p_{j+1}$) & repeat until θ has changed by π. We encounter $O(N)$ pairs of supporting vertices this way. The max separation over these pairs is the diameter of P.

(since one p_i & p_j advance [illegible])

As a result we have:

> Cor: The diameter of a set of N points in the plane can be found in optimal time $\Theta(N \log N)$

Algorithm:
1. Given S, let $P = CH(S)$. Time: $O(N \log N)$
2. Compute diameter of P. Time $O(N)$

In fact there are some special cases.
It turns out that:

I. If S is input as a sequence of vertices describing a simple polygon, then it is possible to compute $CH(S)$ in linear time (& space).

II. If the points of S are chosen randomly (in such a way that the expected number of extreme points in a sample of size N is $O(N^p)$, for some fixed $p<1$) then the expected running time of (a variation of) Mergehull is $O(N)$.

Thus, for problems of class I or II, we can use the linear P-diameter algorithm to compute the diameter of a set of N points in the plane in linear (expected) time.