**Convex Hull**

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

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

**Def.** A polyhedral set in $\mathbb{E}^d$ 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 $\mathbb{E}^d$ 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 \( S = \{ q_1, \ldots, q_n \} \subset \mathbb{E}^d \)

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 \( \text{conv}(S) \).

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:

We write: \( \text{Problem 2} \preceq \text{Problem 1} \)

and 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:

\[ \text{The sorting is linear-time transformable to 2D Convex Hull} \]

\[ \text{Sort} \preceq \text{Problem 1} \]

Corollary: Finding the convex hull of \( N \) points in the plane requires \( \Omega(N \log N) \) time. \( \Omega \) means "at least", so a
Given $N$ real numbers $x_1, \ldots, x_N$.

e.g. 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 $x > 0$.]

\[
\text{SORT}_{\leq N} CH
\]

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

It turns 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 1983 paper


That provides an algorithm with time complexity $O(N \log N + N^{d-1})$. 
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 \( \text{conv}(S) \).
Then \( E \) is the smallest subset of \( S \) such that \( \text{conv}(E) = \text{conv}(S) \).
Moreover, \( E \) consists precisely of the vertices of \( \text{CH}(S) \).

This yields a rather straightforward (but inefficient) method for computing \( \text{CH}(S) \), 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: Intermior to the triangle, 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 internal point.
So, we first find a point $q$ interior to $E$. Again, the way we do this is find two adjacent vertices, then scan around the remaining vertices until we have 3 affixing indices on $q$, 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 again is $O(N \log N)$ 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 cood system so $f = \text{origin} = \text{internal point}$.
(Actually, it'd be an external point.)

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

$$p_1 < p_2 \text{ iff (1) } \text{arg}(p_1) < \text{arg}(p_2)$$

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

As before we can compute to compute (1) (quadrant information). If we need to compute (2), then we really just need to compare $x$ or $y$ coördinate to decide which $|p_i|$ is greater, and the points are colinear, (so no steps required)

In order to access the sorted vector 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 even order, examining triples of consecutive vertices, eliminating those for which the angle is "reflex" (or "right turn").

E.g., eliminate $p_2$

\[ \begin{array}{c}
\text{keep } p_3 \\
\text{not on our hull only}
\end{array} \]
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 to start over and consider $p_0 p_1 p_3$.

Deciding whether to keep or eliminate $p_2$ is based on a cross product or a similar quantity, namely the sign of the 2D cross product:

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

pos $\rightarrow$ eliminate
neg $\rightarrow$ keep

**Note:** If $p_0$, $p_2$, and $p_3$ are collinear,

Since the origin is an interior point and since the $p_i$ are sorted, we must have

$$\cdot p_3 \quad \cdot p_2$$

$$\cdot p_2 \quad \cdot p_3$$

$$\cdot p_1 \quad \cdot p_2$$

So we should eliminate $p_2$.

**Step:** Start after a complete cycle (i.e., keep track of whether every vertex has been tested).

**Complexity:** The scan runs in $O(n)$ time since it either advances one vertex or eliminates one vertex per cycle, and the sort step takes $O(n \log n)$ time.

So overall: $O(n \log n)$ time,

$O(n)$ space.

i.e. optimal.

(Again, notice the intimate use of sorting.)
Ex:  

Progression of algorithm:  

<table>
<thead>
<tr>
<th>Step</th>
<th>Current Sorted List</th>
<th>Triple under consideration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8 )</td>
<td>( P_1, P_2, P_3 \rightarrow \text{elim } P_2 )</td>
</tr>
<tr>
<td>2.</td>
<td>( P_1, P_3, P_4, P_5, P_6, P_7, P_8 )</td>
<td>( P_3, P_4, P_5 \rightarrow \text{keep } P_3 )</td>
</tr>
<tr>
<td>3.</td>
<td>( P_1, P_3, P_4, P_5, P_6, P_7, P_8 )</td>
<td>( P_3, P_4, P_5 \rightarrow \text{keep } P_3 )</td>
</tr>
<tr>
<td>4.</td>
<td>( P_1, P_3, P_5, P_6, P_7, P_8 )</td>
<td>( P_1, P_3, P_5 \rightarrow \text{keep } P_3 )</td>
</tr>
<tr>
<td>5.</td>
<td>( P_1, P_3, P_5, P_6, P_7, P_8 )</td>
<td>( P_3, P_5, P_6 \rightarrow \text{keep } P_5 )</td>
</tr>
<tr>
<td>6.</td>
<td>( P_1, P_3, P_5, P_6, P_7, P_8 )</td>
<td>( P_5, P_6, P_7 \rightarrow \text{keep } P_5 )</td>
</tr>
<tr>
<td>7.</td>
<td>( P_1, P_3, P_5, P_6, P_7, P_8 )</td>
<td>( P_3, P_5, P_6 \rightarrow \text{keep } P_5 )</td>
</tr>
<tr>
<td>8.</td>
<td>( P_1, P_3, P_5, P_7, P_8 )</td>
<td>( P_5, P_7, P_8 \rightarrow \text{keep } P_7 )</td>
</tr>
<tr>
<td>9.</td>
<td>( P_1, P_3, P_5, P_7, P_8 )</td>
<td>( P_5, P_7, P_8 \rightarrow \text{keep } P_8 )</td>
</tr>
<tr>
<td>10.</td>
<td>( P_1, P_3, P_5, P_7, P_8 )</td>
<td>( P_5, P_7, P_8 \rightarrow \text{keep } P_8 )</td>
</tr>
</tbody>
</table>

output line 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 if convex hull algorithms,

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

Jarvis’s March

This algorithm focuses on edges rather than vertices.

The line segment \( l \) defined by two points of \( S \) is an edge 
of \( \text{conv}(S) \) if all points of \( S \) lie in one half-space 
determined by \( l \).

This gives a simple \( O(N^3) \) algo:

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 boundary.
At each stage we add a new point, i.e., a new hull 
edge, by picking the point with the smallest polar angle, 
within the nearest point on interior.

\( T_n \)
We start the march off with the "lexicographically lowest" point of $S$ (specifically, the point satisfying $\max \min \tilde{z}(x,y) \in S\tilde{z}$).

The march takes us to the "lexicographically highest" point of $S$ (min max), while tracing 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.

**Complexity:** $O(hN)$ time

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

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

But, in practice, my run times are

$[O(N)$ 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 $N$ and pick the one of least polar angle).

$*$ We can compute these two points ahead of time in $O(N)$ time

$*$ Simply start and stop the marches at these points, rather than let the first march go beyond the top and continue.
Recall Quicksort:

Given array of $N$ numbers.

1. Split array in "half" so 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 distribution
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 $[N-1, 1]$. Worst $O(N^2)$ case...
Quickhull

First, pick l and r to be points with min and max x values. Split \( S \) into \( S^{(1)} \) and \( S^{(2)} \) using the line through \( l \) and \( r \) (Note: \( S^{(1)} \) and \( S^{(2)} \) are sets of all points in \( S \) on one side.

The recursive step expects a set \( \hat{S} \) and two points \( \hat{p}_l, \hat{p}_r \) such that \( \hat{S} \) lies wholly to one side of the line \( \hat{p}_l \hat{p}_r \) (incl the line), and indeed includes all points of \( S \) on that side of the line.

\[ \begin{array}{c}
\text{Example:}
\end{array} \]

The algorithm finds a point \( h \) such that the triangle \( \hat{p}_l \hat{h} \hat{p}_r \) has maximum area.

(If there are several such triangles pick the \( h \) that maximizes the angle at \( \hat{p}_l \).)

Then \( h \in \text{conv}(S) \), since no points of \( S \) lie "above" \( h \) and bounded by the normal to \( \hat{p}_l \hat{p}_r \).

We then construct two sets \( \hat{S}^{(1)} \) and \( \hat{S}^{(2)} \) consisting of those points of \( \hat{S} \) that are not in the interior of the triangle (points interior to the triangle are discarded).
Cannot be element of $CH(S)$.

$\hat{S}^{(1)}$ consists of points to one side of the line $\ell_{\bar{p}^2 \bar{h}}$.

$\hat{S}^{(2)}$ consists of points to the other side of the line $\ell_{\bar{p}^2 \bar{h}}$.

We call the algorithm recursing on

$$(\hat{S}^{(1)}, \bar{p}^2, \bar{h}) \times (\hat{S}^{(2)}, \bar{h}, \bar{p}_r).$$

**Note:** By construction of $\bar{h}$, $\hat{S}^{(1)} \cap \hat{S}^{(2)}$ consists only of $\bar{h}$.

We can train $\hat{S}^{(1)}$ on all points lying on or to the left of the directed edge $(\bar{p}^2, \bar{h})$.

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

[Given that $\hat{S}$ lies on or to the left of $(\bar{p}^2, \bar{p}_r)$]

**Complexity:** At each stage, determining $\bar{h}$ requires $O(N)$ time.

Similarly, determining $\hat{S}^{(1)}$ or $\hat{S}^{(2)}$ requires $O(N)$ time.

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

However, if the splitting process is nicely behaved, can get $O(N \log N)$.

**Note:** Termination of recursion: If $\hat{S} = \emptyset$, return $\hat{S}$. In that case return $\emptyset$, the previous cells concatenate returned value (while removing duplicate $h$ values).
Formally: \( \text{Quick Hull}(S, p, p_r) \):

If \( S = \emptyset \), \( p \notin S \), then return the list \( p \); 

Else: determine \( h, h', S_1, S_2 \). 

\[
\begin{align*}
\text{vert}^{(1)} & \leftarrow \text{QuickHull} \left( S^{(1)}, P_l, h \right) \\
\text{vert}^{(2)} & \leftarrow \text{QuickHull} \left( S^{(2)}, h, P_r \right)
\end{align*}
\]

return \( \text{append}(\text{vert}^{(1)}, \text{vert}^{(2)}, \emptyset) \)

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 lie wholly to one side of it: \( p \leftarrow \text{fake pt} \).

**Ex:**

![Diagram of points and lines](image)

**Calls:**

\[
\begin{array}{c|c|c}
\text{Computation} & \text{Returned Values} \\
\hline
Q(S, p_1, p_3) & h \leftarrow p_2, S^{(1)} \leftarrow \{p_1, p_2, p_3\}, S^{(2)} \leftarrow \{p_2, p_3, p_4\} & (p_4, p_3), (p_3, p_2) \\
Q(S^{(1)}, p_2, p_3) & (p_3, p_2) & (p_4, p_3, p_2) \\
Q(S^{(2)}, p_3, p_2) & (p_2, p_1) & (p_4, p_3, p_2) \\
Q(S^{(3)}, p_3, p_1) & (p_4, p_3, p_2) & (p_4, p_3, p_2) \\
\end{array}
\]

Similarly \( \text{Q}(S, p_1, p_4) \) returns:

\[
(p_4, p_3, p_2, p_1, p_3, p_4)
\]

And so the initial call to \( \text{Q}(S, p_1, p_4) \) on which these results to get:

\[
(p_4, p_3, p_2, P, p_9, p_5, p_4)
\]
Mergehull → Divide and Conquer Approach.

Given an input set of points, S, suppose we split S into two roughly same size sets S₁ and S₂.

If we now recursively compute \( CH(S₁) \) & \( CH(S₂) \), how long does it take to compute \( CH(S) \) from this data?

We know \( CH(S₁ \cup S₂) = CH(CH(S₁) \cup CH(S₂)) \)

So this leads us to consider the following problem:

**Problem:** Given two convex polygons \( P₁ \) & \( P₂ \), find the convex hull of their union.

We will shortly exhibit an algorithm that solves this problem in time \( O(N) \), where \( N \) is total number of points.

Consequently, we get the following recurrence for the running time of Mergehull:

\[ T(N) ≤ 2T(N/2) + O(N) \]

\[ \Rightarrow T(N) = O(N \log N) \]

**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 \( L \).

**Notes.** If \( P₁ \) & \( P₂ \) are two polygons (possibly overlapping) such that one is not entirely contained in the other, then \( P₁ \) & \( P₂ \) share common supporting lines. At least 2; not more than 2+min(\#edges \( P₁ \), \#edges \( P₂ \)).
Here is one way to compute the supporting line of $P_1 \cup P_2$:

Compute $CH(P, UP_2)$

Now scan the vertex list of $CH(P, UP_2)$. Any pair of consecutive vertices in this list arising from both $P_1$ and $P_2$ (i.e., one vertex is in $P_1$'s vertex list, the other in $P_2$'s) defines a supporting line.

Note: Given that we can compute $CH(P, UP_2)$ in time $O(N)$, this algorithm also runs in time $O(N)$, since the scan is linear.

Here is an algorithm for computing $CH(P, UP_2)$, given convex $P_1 \cup P_2$:

There are two main steps:

1. (We will elaborate on this shortly)
   
   Create a list of vertices sorted around some point interior to $CH(P, UP_2)$ that includes all the vertices of $conv(P, UP_2)$, plus possibly some exterior ones.

2. Use the scanning portion of Graham's Scan to eliminate from this list all vertices interior to $CH(P, UP_2)$.

Notice that this algorithm looks a lot like Graham's Scan. In particular, Step 2 is the same and runs in time $O(N)$.

Step 1 is different in that we will use the already sorted nature of $P_1 \cup P_2$ to create a sorted list in time $O(N)$, rather than $O(N \log N)$.
So let's expand Step 1:

1.1: \( p \in \text{some point internal to } P_1 \)

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 \( p \):

![Diagram showing P1 and P2]

1.4: Determine two wedges and discard all the vertices of \( P_2 \) that lie strictly inside the wedge facing \( p \). / * All such vertices are interior to \( CH(P, UP_2) \) * /

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.

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Step</th>
<th>Combin.</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(N) )</td>
<td>1.1</td>
<td>( O(N) )</td>
<td>usual reason.</td>
</tr>
<tr>
<td>( O(N) )</td>
<td>1.2</td>
<td>( O(N) )</td>
<td>Since ( P_2 ) is a polygon.</td>
</tr>
<tr>
<td>( O(N) )</td>
<td>1.3</td>
<td>( O(N) )</td>
<td>merge of two ( O(N) ) lists</td>
</tr>
<tr>
<td>( O(N) )</td>
<td>1.4</td>
<td>( O(N) )</td>
<td>scan vertex of ( P_2 ). Find extreme angle w.r.t ( p )</td>
</tr>
<tr>
<td>( O(N) )</td>
<td>1.5</td>
<td>( O(N) )</td>
<td>merge of two ( O(N) ) lists</td>
</tr>
</tbody>
</table>
Dynamic Convex Hull

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

**Complexity:**
1. There exists an algorithm that allows addition of new points with an update time of \( O(\log N) \).
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 \( \text{conv}(P \cup \{q\}) \) 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 \( \text{CH}(P \cup \{q\}) = P \). Otherwise, we eliminate from \( P \) all vertices inside the wedge ordered at \( q \) and add in the edge from the support line, \( \text{CH}(P \cup \{q\}) \).
Eg:

```
V
P

lines

added
eliminated
vertices

CH(P U Eq)
```

We classify vertices of P relative to q as follows:

```
Concave  Reflex  Supporting
```

(Note: we assume no three vertices of P are collinear. Hence these classifications can be computed in constant time.)

For each we the implicit ordering of the vertices of P.

The trick in the algorithm for computing CH(P U Eq) is to find the two supporting vertices of P related to q (if they exist) quickly.

We will then maintain the polygon P as a height-balanced tree that supports add and delete operations in O(log N) time, with the vertices sorted in CCW order about P. For each of our similar data structure,

...
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 made of $P$, we will have different minimum and maximum elements in $T(P)$.

But, it doesn't matter. Pick one of these.

Now, let $r$ be the root of $T(P)$ and $m$ the minimum vertex of $T(P)$. (i.e., $v_1, v_2, v_3$ in the example above.)

We will be concerned with the relationship between $r, m, q, t$ specifically, we are interested in design of the cross-product $(m-q) \times (r-t)$.

There are two cases really!

- Positive cross-product
- Negative cross-product
- (or zero)
The following 8 cases are of interest (to cover all possibilities):

<table>
<thead>
<tr>
<th>Case</th>
<th>Cross Product Sign</th>
<th>Classification of m</th>
<th>Classification of r</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>concave</td>
<td>concave</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>concave</td>
<td>not concave</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>not concave</td>
<td>reflex</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>not concave</td>
<td>not reflex</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>reflex</td>
<td>reflex</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>reflex</td>
<td>not reflex</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>not reflex</td>
<td>concave</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>not reflex</td>
<td>not concave</td>
</tr>
</tbody>
</table>

The idea is to use these local classifications to search T(p) efficiently, in order to locate potential vertices.

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

Consider the case:

1. In this case we know that we must recursively search the right subtree for the support vertex which \text{not exist if } q \text{ is inside } \mathcal{P} \text{. (which turns out to be inside } \mathcal{P})

In the case we know that we must recursively search the right subtree for the support vertex, which turns out to be inside \mathcal{P}.

Circle represents the polygon \mathcal{P}, \text{min} as sorted, \text{begin at } m \& \text{running counterclockwise.}
In this case q must lie outside $\mathcal{F}$, and the 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, and 7 are variations on 1:

3:

Both support vertices lie in $\text{LEFT}[r]$. Both support vertices lie in $\text{RIGHT}[r]$. They exist, lie in $\text{LEFT}[r]$.

4:

One support vertex in $\text{LEFT}[r]$. One support vertex in $\text{RIGHT}[r]$. The other is either in $\text{RIGHT}[r] U S_{r}$ or in $\text{LEFT}[r] U S_{r}$.
Let \( v_r \) be the support vertex defining the left support line \( L \), and let \( v_r \) be the support vertex defining the right support line.

We must now delete the vertex "between" \( v_r \) and \( v_r \), then add in \( q \). The result is \( CH(P \cup \{q\}) \) (or worse case...).

In order to delete the vertex, we turn it to a balanced tree or define a constant-time queue and use the efficient \texttt{SPLIT} \texttt{SPLIT} operation.

There are two cases:

1) If \( v_r < v_r \) in the tree ordering, then we split the tree \( T(P) \) once at \( v_r \), retaining all vertices \( \leq v_r \), and we split the tree \( T(P) \) again at \( v_r \), retaining all vertices \( \geq v_r \). Finally we splice these together and add in \( q \).

E.g.:

\begin{align*}
&\begin{array}{c}
V_o \\
T_1 & T_2 \\
& W
\end{array} \\
\rightarrow \\
\begin{array}{c}
V_r \\
& W \\
T_1 & T_2 \\
& V_r
\end{array}
\end{align*}
2) If \( V_r \neq V_p \) in the tree order, then we again split at \( V_r \) and \( V_p \), but this time we don't need to split. We do add in \( q \).

\[
\begin{array}{c}
\text{Fig.} \\
\begin{array}{c}
\text{V}_p \\
\rightarrow \\
\text{V}_r \\
\end{array}
\end{array}
\]

(plus rebalancing at each step)

**Complexity:**

Finding \( V_r \) and \( V_p \) 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 vertex between \( V_r \) or \( V_p \) requires at most a constant number of SPLIT, SPLICE, and INSERT operations, each of which requires \( O(\log N) \) time.

\( O(\log N) \) total time.

Note: The tree picture above are just examples. Other possibilities exist.
Vertex classification with \( q \): \( V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \), reflex, supporting, concave, concave, supporting, reflex.

Suppose \( T(P) \) is

Then the search proceeds as follows:

**SEARCH for \( v_p + v_r \) in \( T(P) \)**

\[ r = v_4 \]
\[ m = v_1 \]
\[ \text{cross-product} > 0 \]

**Case 4**

**Search for \( v_p \) in \( \text{RIGHT}[v_4] \)**

\[ r = v_6 \]
\[ m = v_5 \]
\[ \text{cross-product} < 0 \]

**Case 8**

**Search for \( v_p \) in \( \text{LEFT}[v_4] \)**

\[ r = v_2 \text{ which is supporting} \]

so search ends

\[ v_p = v_5 \]

**Search for \( v_p \) in \( \text{LEFT}[v_6] \)**

\[ r = v_5 \text{ which is supporting} \]

so search ends

\[ v_p = v_5 \]
Delete vertices between $v_5$ and $v_2$, and add vertex $v_7$.

Note: $v_7 \cap V_p$, so two splits plus one insertion ($f_7$) are required.

1. Split at $v_4 = v_5$, retaining all vertices $\leq v_p$.
   
   Result tree:
   
   \[
   \begin{array}{c}
   \text{result tree:} \\
   \text{\includegraphics[width=0.3\textwidth]{tree1.png}}
   \end{array}
   \]

2. Split at $v_4 = v_2$, retaining all vertices $\geq v_p$.
   
   Result tree:
   
   \[
   \begin{array}{c}
   \text{result tree:} \\
   \text{\includegraphics[width=0.3\textwidth]{tree2.png}}
   \end{array}
   \]

3. Insert $f_7$ into the result of step 2.
   
   Final tree:
   
   \[
   \begin{array}{c}
   \text{Final tree:} \\
   \text{\includegraphics[width=0.3\textwidth]{final_tree.png}}
   \end{array}
   \]

   (which is still balanced)

The resulting order is: $v_2 \ v_3 \ v_4 \ v_5 \ v_7$. Which does indeed specify $CH(\{V_0, V_3\})$. 

Applications of Convex Hull

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

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

There is an $O(N^2)$ algorithm: examine all pairs of points, but what is an optimal algorithm?

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

- The computation of the diameter of a finite set of $N$ pts in $\mathbb{R}^d$ requires $O(N \log N)$ time (in the algebraic computation tree model).
  - It works by reducing set disjointness (of real numbers) to set diameter in 2D: maps sets to points on $y$ axis.

The connection to convex hull is given by:

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

And then we have the following help:

- The diameter of a convex figure (in 2D) is the greatest distance between parallel line 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.)
Indeed we see that the max occurs for support contact either with the pair of points \((P_1, P_2)\) or the pair \((P_3, P_4)\).

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 gives us an \(O(N)\) algorithm for detecting the maximum diameter of a convex polygon.

We simulate the rotation of a parallel jaw gripper around \(P\). Specifically, we start with two unique \(P\) with extreme \(y\)-coordinates and let \(\theta = 0\). Suppose the two verts are \(P_1 \equiv P_3\). Consider the angle \(\theta_{i+1}\) of the edge \(P_iP_{i+1}\). We advance \(\theta\) to the min of \(\theta_{i+1}\), then switch the appropriate support vertex (either \(P_{i+1} \equiv P_i\) or \(P_i \equiv P_{i+1}\)) and repeat until \(\theta\) is changed by \(\pi\). We encounter \(O(N)\) pairs of support verts, update them this way. The max separation over those pairs is the diameter of \(P\).
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 is \( O(N) \).

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