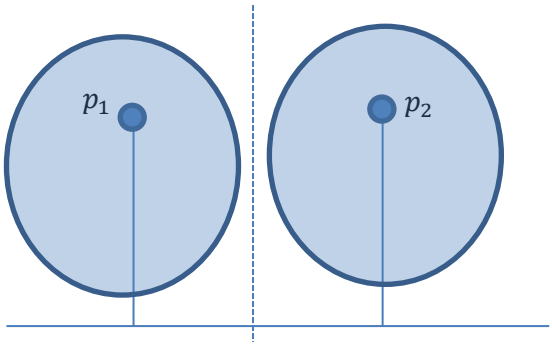
Divide and Conquer

Like merge sort, the basic idea is break down the problem into two subproblems, solve them and then merge the solution. The same here: you break the points in the plane into 2 parts, compute the convex hull for each part and merge the solutions together.



$$x_{p_1} \leq x_{p_2} \leq \cdots \leq x_{p_n} \Rightarrow \Theta(n \lg n)$$

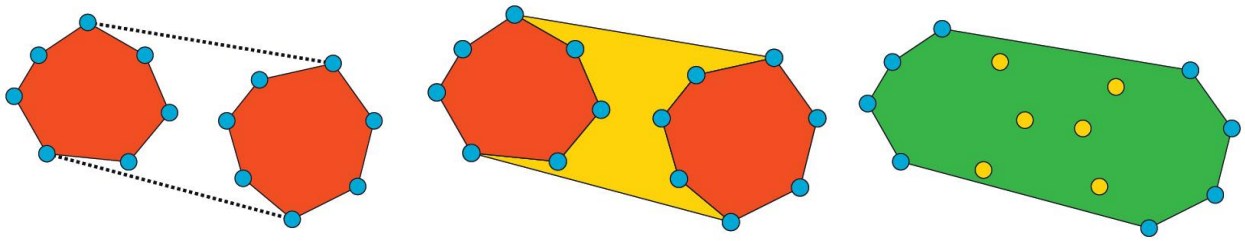
We assume that no two points are on the same perpendicular (otherwise we simply find a new x-axis that satisfies that).

Assume the merge can be done in linear time, the recurrence is:

$$T(n) = \begin{cases} 1, & b \leq 3 \\ n + 2T\left(\frac{n}{2}\right), & otherwise \end{cases}$$

How do we merge?

We need to identify the upper and lower tangents, and then everything between them will disappear:



For any candidate line between a point in A (the left polygon) and a point in B (the right polygon), we can verify whether that line is **cotangent** in constant time: assume the points connecting that line are x, y and the one left to x is a on A and the one right to y on B is b, then $a \to x \to y \to b$ should be in the same orientation (which is verified in constant time).

Note that A, B cannot intersect, because we draw the line to partition the plane that defines A, B.

The merge algorithm:

- We start with initial guess of the right-most point (x-wise) of A and the left-most point of B. Call those points a on A and b on B.
- Let b_2 be the next point after b. If $a \to b \to b_2$ is a right turn, we take b_2 instead of b. Once we moved forward with b, that would happen once and we will never take that point again.
- The same we do backwards with a.

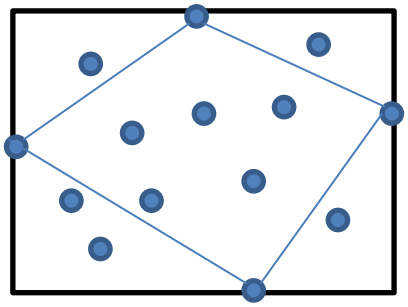
In worst-case we have to go all the way down, iteration over almost all points in A and B, and each orientation check is constant, concluding to a total $O(n)$ time for the merge part (We do the same symmetrically for the upper cotangent). That's the **LowerTangent(Conv(A),Conv(B))** method (or the symmetric UpperTangent).

At the end the total running time wil be $O(n \lg n)$.

Assume we have a chain of a convex hull, and we want to find the tangent from an outside point q to some vertex of the polygon (upper / lower, it is symmetric). We can do that with a binary search:
Let q be a point right to the convex hull, and p_1 the closest point to q, p_k the farthest. We want to find the point between p_1, p_k on the chain, p_i, such that $q \to p_i \to p_{i+1}$ is the first left turn (i.e. $q \to p_{i-1} \to p_i$ is a right turn).

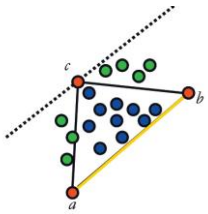
QuickHull

Similar to quicksort, this algorithm is $O(n \lg n)$ expected time and $O(n^2)$.
We want to discard as many points as fast as possible.
For a given convex hull of a set of points, we can identify the 4 points with highest/lowest x/y values, thus getting a box around the convex hull.



Moreover, when we draw the lines between these extreme points, we can discard all the points within the diamond that is formed (the blue object in the drawing). That diamond is called the **support quadrilateral**. We are left to look in the 4 triangles that are left.
Classifying the left points in the triangles into 4 sets is linear – a simple orientation test.
Any point in those triangles is actually witnesses that the current edge (which is initialized to a segment between two extreme points) is not part of the convex hull chain (boundary). We want to "push" that edge farther away towards the corners of the enclosing box.

From this point on we will work only with triangles:



We find the extreme point c that is the farthest apart from the line $[a, b]$ which we know will definitely be on the chain, then create a triangle – everything inside it is discarded, then we need to solve the problem for the green witnesses left – these will be handles in the same triangle manner.

Running time:

$T(n) = \Theta(n) + T(n_1) + T(n_2)$, where n_1, n_2 are the green points above (and $n_1 + n_2 \leq n^2$), and in worst case that's $O(n^2)$ (for the height behavior where we divide by the point with the largest height). The worst case would be when all points sit on a curvature.

Note that if we go **by angle** and not **by height**, the worst-case scenario wouldn't necessarily be the same.

Going by height: calculate which point has the largest height with respect to the segment $[a, b]$.

Going by angle: calculate which point c has the largest angle $\angle abc$.

Gift-Wrapping and Jarvis's March

This is similar to selection sort. If h is the size of the chain, the running time is $O(nh)$, meaning this is an **output-sensitive** algorithm.

At each step, assuming we have the points p_{k-1}, p_k we're looking for point q which maximizes the angle $\angle p_{k-1} p_k q$.

If $h = o(\log n)$, this is better than Graham's scan. But it could be the case that $h = n$, concluding with $O(n^2)$.

The full algorithm is described in the slides.

Chan's Algorithm

This algorithm is also output sensitive, but acts in worst case $O(n \lg n)$ and not $O(n^2)$, otherwise it runs in $O(n \lg h)$. If h is actually $\log n$, then it will run in time $O(n \lg \lg n)$.

The idea of Chan's algorithm:

- Break the problem to subproblems.
- Solve them using Graham's.
- Merge them together with Jarvis's.

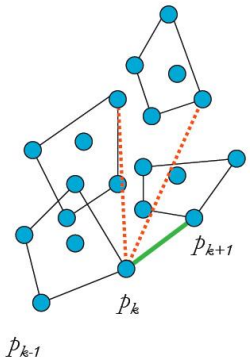
The algorithm:

Partition the points into groups of equal size m, such that we get $r = \lceil n/m \rceil$ groups.

For each group we run Graham's scan, then the running time for each group is $O(m \lg m)$, so the total running time for Graham's scan is $r \cdot m \lg m = \frac{n}{m} \cdot m \lg m = n \lg m$. note that the convex hulls we calculated may overlap.

Now we use Jarvis's march to merge them, addressing each convex hull as a super-node:

We can in time n find the convex hull that is the lowest convex hull (the one with the lowest point). From this point, we compute the tangents to each of the r convex hulls which takes $O(\log m)$ for each convex hull. We know that the tangent with the smallest angle is in the convex hull (the green segment in the figure):



The total running time to calculate the tangents at each step is $O(r \lg m)$, since we need to look at all the convex hulls (r of them) and for each find the tangent in $\lg m$ time.

This process has to be done h times, so the total running time for this phase is $hr \lg m = \frac{hn}{m} \lg m$.

Total running time:

$$O\left(\left(n + \frac{hn}{m}\right) \lg m\right)$$

But, what is m?

If we choose $m = h$ we will get $O(n \lg h)$. But we don't know h…

If we choose $m < h$, then the algorithm will not terminate at p_0 in the gift-wrapping phase. The loop that goes from 1 to m in the wrapping has to go up to m, and we cannot use an arbitrary value for m, otherwise the Jarvis's march phase will cost up to $O(nh)$.

So m has to be of the form $m = h^c$, then the running time will end up being $n \lg h^c$.

We will choose m as follows: we start with a small m and increase it rapidly by squaring it:

For $t = 1,2,\ldots$ do:

- Let $m = min(2^{2^t}, n)$.
- Invoke PartialHull(P,m), returning the result in L.
- If $L \neq$ "try again" then return L.

Why running the iterations above for m still ends up with $O(n \lg h)$ running time?

The last iteration at which we stop is for some t, such that $m = 2^{2^t}$ and so the running time will be $O\left(n \lg 2^{2^t}\right) = O(n \cdot 2^t)$.

The upper bound of t would be $\lg \lg n$, but moreover the algorithm will find the solution correctly when $m \cong h$ and that would be when $2^{2^t} \cong h \Rightarrow t \cong \lg \lg h$. Therefore the total iterations we will run will cost:

$\sum_{t=1}^{\lg \lg h} n2^t = n \sum_{t=1}^{\lg \lg h} 2^t = n(2^{\lg \lg h + 1} - 1) = O(n \lg h)$ - and that's exactly what we had originally.