Today, I'll finish presenting the modification by Andrew ('79) of the well known Graham's scan ('72) algorithm that runs in $O(n \log n)$ time.

Let $p_1, \ldots, p_n$ be the points after sorting them left to right.

We'll focus on finding the upper chain vertices in order from left to right. The algorithm for the lower hull is symmetric. After computing both, we can concatenate the lower chain list with the reversal of the upper chain list to compute the whole convex hull.

Like I said, we'll be doing incremental construction, meaning we add points to a collection one-by-one, maintaining the upper chain of the points added so far. Let $P_i := \{p_1, \ldots, p_i\}$. After adding point $p_i$ to our collection to change $P_{i-1}$ into $P_i$, we'll want to update the upper chain of $P_{i-1}$ to become the upper chain of $P_i$.

So what would updating the upper chain look like? We'll use a couple observations:

1. The upper chain of $P_i$ must contain $p_1$ and $p_i$, because they are the farthest points to the left and right, respectively.

2. Let $< p_{(i-1)}, \ldots, p_{(i_m)} >$ denote the upper chain vertices of $P_i$ in order from left to right. For all $1 \leq j \leq m$, all points of $P_i$ lie on or below each line $<p_{(i_j)} p_{(i_{j-1})}>$ and each consecutive triple $< p_{(i_j)} p_{(i_{j-1})} p_{(i_{j-2})} >$ of upper chain vertices forms a left-hand turn.

Point 1. above motivates us to store upper chain vertices in a stack $S$ with vertices further to the right higher in the stack. That way, we can quickly add each new $p_i$ as we go but also quickly remove the last few that no longer appear to belong to the upper chain. We'll let $S[t]$ denote the top of the stack and $S[t-1]$ denote the point second-from-the-top.

To use point 2., we'll need a way to quickly determine if three points make a left-hand or right-hand turn. Let $\text{orient}(p, q, r) > 0$ if the three points makes a counterclockwise triangle / left-hand turn. We could say they have positive orientation. Similarly, we have negative orientation and zero orientation for the case when the points share a line.
If we remember linear algebra, remember physics, or just trust Mount, orientation is easy to compute as

$$\text{orient}(p, q, r) = \text{sign} \left( \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix} \right).$$

The Details

- OK, we have all the high-level ideas in place. As is often the case, we can sort of reason our way through the details.
- $P_{i-2}$ is the line segment $-p_1 p_2$. Easy enough...
- Now, suppose we have the stack $S$ storing the upper chain of $P_{i-1} = \{p_1, \ldots, p_{i-1}\}$. We want to add and remove points so $S$ now stores the upper chain of $P_i = P_{i-1} \cup \{p_i\}$. We know $p_1$ needs to stay in $S$ and (eventually) we’ll want to add $p_i$.
- First off, consider any point $q$ in $P_{i-1}$ but not in $S$. Point $q$ lies strictly below the upper chain, and it will continue to do so as the upper chain extends to $p_i$, so $q$ won’t need to be added to $S$.
- From point 2. above, we know the last three points of the upper chain need to form a left-hand turn / have positive orientation. So if $<p_i, S[t], S[t-1]>$ forms a right-hand turn, those cannot be the last three vertices of the upper chain. In particular, $S[t-1]$ lies above the line $\leftrightarrow_{p_i} S[t]$, so $S[t]$ is not a member of the upper chain for $P_i$. We can safely pop $S[t]$ from the stack.
- We repeat this process over and over: Check each new triple $<p_i, S[t], S[t-1]>$ for a right-hand turn. If it has one, pop $S[t]$. Stop when the turn is left-hand or the stack only contains $p_1$ (which, again, we never want to pop).
- If and when we see a left-hand turn $<p_i, S[t], S[t-1]>$, we push $p_i$ onto the stack. So now $S[t] = p_i$.
- Every turn is now left-hand, so we cannot remove any vertex of the upper chain without it appearing above the shortcut supporting line. I guess we’re done!
- Here’s a figure of the process. Here, $p_j$ is the predecessor to $p_i$ in the upper chain for $P_i$.
- Here’s some pseudocode from Mount:
Running Time

- Sorting takes $O(n \log n)$ time.
- Let $d_i$ be the number of pops when inserting $p_i$.
- We do one orientation test per pop and one more before inserting $p_i$, so the time adding $p_i$ is $O(d_i + 1)$.
- In total, we spend time proportional to $\sum_{i=1}^n (d_i + 1) = n + \sum_{i=1}^n d_i$ building the upper hull after sorting.
- But each node is popped at most once, so $\sum_{i=1}^n d_i \leq n$. The total time building the upper hull is $O(n)$ in addition to the time for sorting. So $O(n \log n)$ total.

Lower Bounds

- But can we find an algorithm with better worst-case performance? It turns out, no, we cannot, assuming our algorithm’s decisions are based on binary comparisons.
- You may be familiar with another setting where this is true. I won’t go into the proof here, but it’s well known that sorting a set of $n$ numbers takes $\Omega(n \log n)$ time in the worst-case if all we’re allowed to do with the numbers directly is compare them. This means every correct sorting algorithm has some input where it runs in time proportional to $n \log n$ or worse.
- We can prove computing the convex hull takes the same amount of time using a reduction: solving one problem (sorting) by calling a correct algorithm for another problem (convex hull in 2D).
- Suppose we want to sort $X = \{x_1, \ldots, x_n\}$ with each $x_n \in \mathbb{R}$.
- We can do this by building an instance of convex hull in 2D. Start with $P$ being empty. For each number $x_i$, add $p_i = (x_i, x_i^2)$ to $P$.
- When you do this, each point appears on the parabola $y = x^2$. So every point of $P$
belongs to the convex hull.

- Now suppose we run some algorithm for convex hull in f(n) time.
- We then find the leftmost point, and trace the hull in counterclockwise order. The x-coordinates of these points are exactly the numbers of X in sorted order.
- Everything we did except computing the hull itself takes O(n) time, so the whole sorting algorithm takes O(n + f(n)) time.
- f(n) = Omega(n log n) in the worst case; otherwise, we could sort in o(n log n) time!
- Now, you might complain that asking for all the edges in counterclockwise order is what made things slow. Mount gives a proof that simply counting the points on the convex hull requires Omega(n log n) time.
- It turns out what really hurts us here is that we create a problem instance where every point appears on the convex hull. Can we do better if significantly fewer points appear on the hull?

**Jarvis's March**

- Let's consider a different algorithm by Jarvis ['73].
- The idea behind the algorithm is we start with some point we know must be on the convex hull, and then we perform linear time searches for each next point along the hull.
- The bottommost point is on the hull, so we can start there.
- Now, suppose we've just added some point v_{i-1} to the hull (not necessarily the bottom one) and we want to know what's next.
- We want to minimize the angle between the directed lines <->v_{i-2} v_{i-1} and <->v_{i - 1} v_i, also called the *turning angle* of v_i with respect to v_{i-2} and v_{i-1}.
- So this process is well-defined for i = 2, we'll add an imaginary “sentinel point” v_0 := (-infty, 0) way off to the left.
- We repeat the process for larger and larger i until we find v_i = v_1.

![Diagram of Jarvis's March](image)

1. Given P, let v_0 = (-\infty, 0) and let v_1 be the point of P with the smallest y-coordinate
2. For i ← 2, 3, ...
   a. v_i ← the point of P \ \{v_{i-1}, v_{i-2}\} that minimizes the turning angle with respect to v_{i-2} and v_{i-1}
   b. If v_i == v_1, return (v_1, ..., v_{i-1})

Finding v_i can be done in O(n) time using orientation tests. It turns out orient(p, v_{i-1}, r) > 0 if and only if the turning angle for r is smaller than the turning angle for p. So we can
use those orientation tests to compare turning angles in the standard min algorithm.

- In the worst-case, we do our $O(n)$ time min turning angle process a total of $O(n)$ times for $O(n^2)$ time total.
- But, if $h$ is the number of points on the convex hull, it actually runs in only $O(nh)$ time!
- Why doesn’t this break the lower bound proof? In that proof $h = n$. This algorithm is actually worse than Graham’s scan in that case.
- But for very small $h$, this algorithm is better than Graham’s scan.
- This is an example of an output sensitive algorithm: the running time depends not just on the size of the input, but also the output.
- We’ll see more output sensitive algorithms throughout the semester, especially when discussing data structures that need to output a bunch of geometric objects.

Chan’s Algorithm

- Chan [’96] described an algorithm that’s as good as both Graham’s scan and Jarvis’s March. It’s actually a combination of both algorithms, but somehow runs in only $O(n \log h)$ time.
- Let’s build up this algorithm piece-by-piece to see how it works.
- First off, Graham’s scan suggests that sorting points is useful. Unfortunately, sorting $k$ points takes $\Theta(k \log k)$ time.
- However, if we break the point set into $n / k$ subsets of size $k$, we can sort them all individually in $O((n / k) \log k) = O(n \log k)$ time. For any constant $c$ and $k \leq h^c$, this running time for those sorts is $O(n \log h)$.
- In fact, we even have time to run Graham’s scan on these subsets.
- Of course, we don’t know $h$ in advance, so we do not know what size subsets we can afford to sort. Let’s suppose for now we have a guess $h^*$ on the true value for $h$.
- We’ll break up the input set $P$ into $k = \lceil n / h^* \rceil$ subsets of size at most $h^*$ each and run Graham’s scan on each subset. This will create a collection of $k$ mini-hulls that may overlap.

Computing all $k$ mini hull takes $O(k(h^* \log h^*)) = O(n \log h^*)$ time total.
- And now we make a couple observations.
- First, any point strictly interior to its subset’s mini-hull will lie strictly interior to $\text{conv}(P)$. In
other words, each vertex of conv(P) comes from a vertex of a mini-hull.

- Second, suppose we know a vertex \( p \) on conv(P). If the next vertex \( q \) of conv(P) in ccw order lies on a vertex of mini-hull \( Q \), then \( \langle pq \rangle \) is a support line for \( Q \).
- Finally, there are only two support lines between \( p \) and \( Q \), and if we’re given an array \( q_1, …, q_m \) of \( Q \)'s boundary vertices in ccw order, we can find both support lines in \( O(\log m) \) time. This utility function as Mount calls it is essentially a pair of binary searches.

- Taken together, these facts give us a faster way to run Jarvis's March! When it’s time to compute vertex \( v_i \) of conv(P), we only consider the \( \leq 2k \) support line vertices from the mini hulls.

- We spend \( O(k \log h^*) + O(k) = O(k \log h^*) \) time finding each vertex of conv(P). Finding all \( h \) vertices of conv(P) will take \( O(h (k \log h^*)) = O(n (h / h^*) \log h^*) \) time.

**Conditional Algorithm**

- Again, \( h^* \) is just a guess for \( h \). What happens if our guess is wrong?
- If we guess high so that \( h^* > h \), then both phases will take \( O(n \log h^*) \) time. That’s bad if we guess really really high, but if, say, \( h < h^* \leq h^2 \), then \( O(n \log h^*) = O(n \log h) \). We can overshoot by a polynomial larger than \( h \).
- If we guess too low, then it takes \( O(n \log h^*) = O(n \log h) \) time to compute the mini hulls, but completing the march may take much more time than we would like. Fortunately, a low guess is easy to detect before it becomes a problem. Just cut the algorithm off as soon as we add an \( h^* \)th vertex to conv(P). We spend at most \( O(n \log h^*) \) time if we cut the algorithm off in this way.
- Both observations motivate this procedure Mount calls ConditionalHull. Given a guess \( h^* \), it tries to compute conv(P), but it gives up if the guess is too low.
Chan’s Algorithm for the Conditional Hull Problem

\textbf{ConditionalHull}(P, h^*) :

1. Let \( k \leftarrow \lceil n/h^* \rceil \). Partition \( P \) (arbitrarily) into disjoint subsets \( P_1, \ldots, P_k \), each of size at most \( h^* \).
2. For \( j \leftarrow 1 \) to \( k \), compute \( H_j = \text{conv}(P_j) \) using Graham’s scan, storing each in an ordered array.
3. Let \( v_0 \leftarrow (-\infty, 0) \), and let \( v_1 \) be the bottommost point of \( P \).
4. For \( i \leftarrow 1, 2, \ldots, h^* \):
   a. For \( j \leftarrow 1 \) to \( k \), using the utility lemma, compute the tangents points \( q_j^- \) and \( q_j^+ \) for \( H_j \) with respect to \( v_{i-1} \).
   b. Set \( v_i \) to be the tangent point that minimizes the turning angle with respect to \( v_{i-2} \) and \( v_{i-1} \).
   c. If \( v_i = v_1 \) then return the pair (success, \( V = \{v_1, \ldots, v_{i-1}\})
5. If we get here, we know that \( h^* < h \), and we return (failure, \emptyset).

• As we saw, the procedure will take \( O(n \log h^*) \) time total.

\textbf{Guessing the Hull’s Size}

• Now we just need to find a guess \( h^* \) such that \( h < h^* \leq h^2 \) (or any other polynomial). But how?
  • We’ll start with a low guess like \( h^* = 3 \). Then we’ll slowly increase it until we get one that we would like.
  • Adding 1 to \( h^* \) after each guess is too slow. Even doubling after each guess isn’t great, because we would need to make \( \log h \) guesses.
  • But we can be off by a polynomial, so why don’t we square the guess each time, trying 2, 4, 16, 256, ..., \( 2^{2^i} \). Even growing this quickly, we’ll still have our desired \( h < h^* \leq h^2 \) the first time we guess high.
  • But will we spend too much time on the earlier guesses?
  • For the \( i \)th guess, we use \( h^* \cdot i := 2^{2^i} \). The time taken to run \text{ConditionalHull} is \( O(n \log h^* \cdot i) = O(n \log 2^{2^i}) = O(n 2^i) \).
  • We’ll use \( \log \) to denote log base 2. Summing the times for each guess gives us \( O(n) \cdot \sum_{i = 1}^{\log \log h} 2^i \) time total. That \( O(n) \) times a geometric series with a constant ratio. It’s well known that these series are proportional to their largest term (try to recall the Master Method if you took 6363), so the total time is \( O(n 2^{\log \log h}) = O(n \log h) \). Hurray!
  • Here’s a summary of the whole process:

\textbf{Hull}(P) :

1. \( h^* \leftarrow 2 \); status \leftarrow failure
2. While (status \neq failure):
   a. Let \( h^* \leftarrow \min((h^*)^2, n) \)
   b. (status, \( V \)) \leftarrow \text{ConditionalHull}(P, h^*)
3. Return \( V \)

• Can we do better? No [Kirkpatrick and Seidel ’86]. See Mount if you want the details.