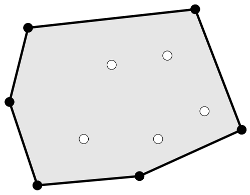
14 Convex Hulls (March 20)

14.1 Definitions

We are given a set P of n points in the plane. We want to compute something called the *convex hull* of P. Intuitively, the convex hull is what you get by driving a nail into the plane at each point and then wrapping a piece of string around the nails. More formally, the convex hull is the smallest convex polygon containing the points:

- **polygon:** A region of the plane bounded by a cycle of line segments, called *edges*, joined end-to-end in a cycle. Points where two successive edges meet are called *vertices*.
- **convex:** For any two points p, q inside the polygon, the line segment $\overline{pq}$ is completely inside the polygon.
- **smallest:** Any convex proper subset of the convex hull excludes at least one point in P. This implies that every vertex of the convex hull is a point in P.

We can also define the convex hull as the *largest* convex polygon whose vertices are all points in P, or the *unique* convex polygon that contains P and whose vertices are all points in P. Notice that P might have *interior* points that are not vertices of the convex hull.



A set of points and its convex hull.
Convex hull vertices are black; interior points are white.

Just to make things concrete, we will represent the points in P by their Cartesian coordinates, in two arrays $X[1..n]$ and $Y[1..n]$. We will represent the convex hull as a circular linked list of vertices in counterclockwise order. if the ith point is a vertex of the convex hull, $next[i]$ is index of the next vertex counterclockwise and $pred[i]$ is the index of the next vertex clockwise; otherwise, $next[i] = pred[i] = 0$. It doesn't matter which vertex we choose as the 'head' of the list. The decision to list vertices counterclockwise instead of clockwise is arbitrary.

To simplify the presentation of the convex hull algorithms, I will assume that the points are in *general position*, meaning (in this context) that *no three points lie on a common line*. This is just like assuming that no two elements are equal when we talk about sorting algorithms. If we wanted to really implement these algorithms, we would have to handle colinear triples correctly, or at least consistently. This is fairly easy, but definitely not trivial.

14.2 Simple Cases

Computing the convex hull of a single point is trivial; we just return that point. Computing the convex hull of two points is also trivial.

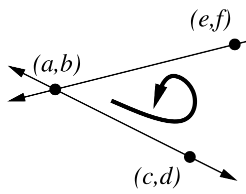
For three points, we have two different possibilities — either the points are listed in the array in clockwise order or counterclockwise order. Suppose our three points are (a, b), (c, d), and (e, f), given in that order, and for the moment, let's also suppose that the first point is furthest to the left, so $a < c$ and $a < f$. Then the three points are in counterclockwise order if and only if the line $\overleftrightarrow{(a,b)(c,d)}$ is less than the slope of the line $\overleftrightarrow{(a,b)(e,f)}$:

$$\text{counterclockwise} \iff \frac{d-b}{c-a} < \frac{f-b}{e-a}$$

Since both denominators are positive, we can rewrite this inequality as follows:

$$\boxed{\text{counterclockwise} \iff (f-b)(c-a) > (d-b)(e-a)}$$

This final inequality is correct even if (a,b) is not the leftmost point. If the inequality is reversed, then the points are in clockwise order. If the three points are colinear (remember, we're assuming that never happens), then the two expressions are equal.



Three points in counterclockwise order.

Another way of thinking about this counterclockwise test is that we're computing the *cross-product* of the two vectors $(c,d)-(a,b)$ and $(e,f)-(a,b)$, which is defined as a 2×2 determinant:

$$\boxed{\text{counterclockwise} \iff \begin{vmatrix} c-a & d-b \\ e-a & f-b \end{vmatrix} > 0}$$

We can also write it as a 3×3 determinant:

$$\boxed{\text{counterclockwise} \iff \begin{vmatrix} 1 & a & b \\ 1 & c & d \\ 1 & e & f \end{vmatrix} > 0}$$

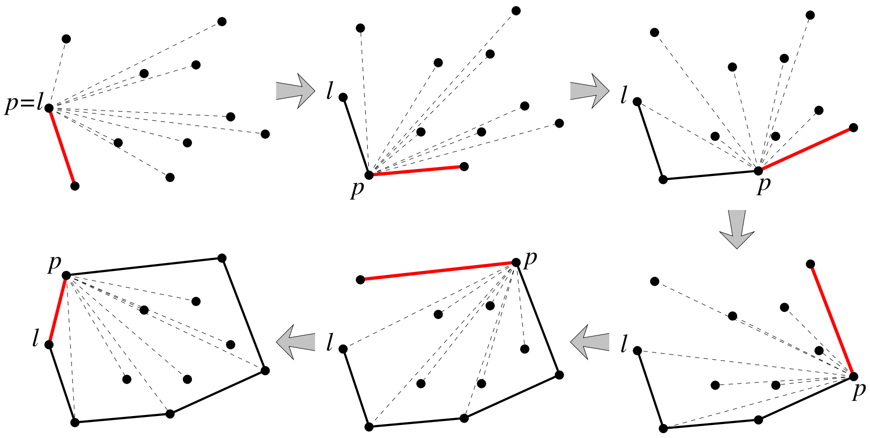
All three boxed expressions are algebraically identical.

This counterclockwise test plays *exactly* the same role in convex hull algorithms as comparisons play in sorting algorithms. Computing the convex hull of of three points is analogous to sorting two numbers: either they're in the correct order or in the opposite order.

14.3 Jarvis's Algorithm (Wrapping)

Perhaps the simplest algorithm for computing convex hulls simply simulates the process of wrapping a piece of string around the points. This algorithm is usually called *Jarvis's march*, but it is also referred to as the *gift-wrapping* algorithm.

Jarvis's march starts by computing the leftmost point ℓ, *i.e.*, the point whose x-coordinate is smallest, since we know that the left most point must be a convex hull vertex. Finding ℓ clearly takes linear time.



The execution of Jarvis's March.

Then the algorithm does a series of *pivoting* steps to find each successive convex hull vertex, starting with ℓ and continuing until we reach ℓ again. The vertex immediately following a point p is the point that appears to be furthest to the right to someone standing at p and looking at the other points. In other words, if q is the vertex following p, and r is any other input point, then the triple p, q, r is in counter-clockwise order. We can find each successive vertex in linear time by performing a series of $O(n)$ counter-clockwise tests.

```
JarvisMarch(X[1..n], Y[1..n]):
  ℓ ← 1
  for i ← 2 to n
      if X[i] < X[ℓ]
          ℓ ← i

  p ← ℓ
  repeat
      q ← p + 1        ⟨⟨Make sure p ≠ q⟩⟩
      for i ← 2 to n
          if CCW(p, i, q)
              q ← i
      next[p] ← q;  prev[q] ← p
      p ← q
  until p = ℓ
```

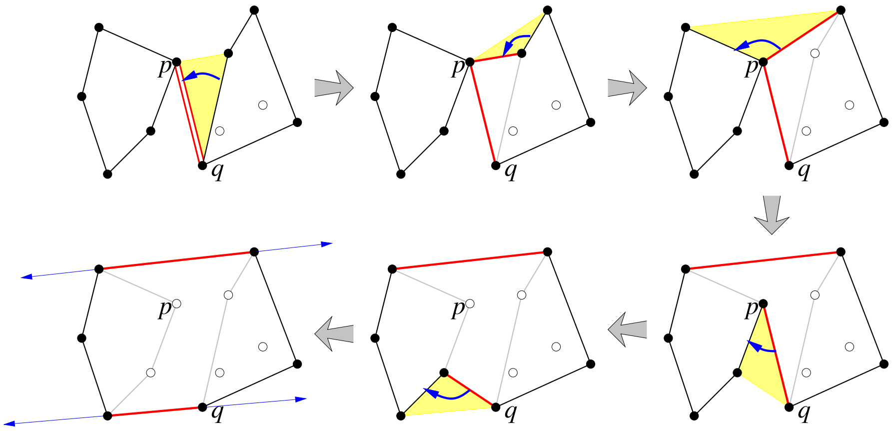
Since the algorithm spends $O(n)$ time for each convex hull vertex, the worst-case running time is $O(n^2)$. However, this naïve analysis hides the fact that if the convex hull has very few vertices, Jarvis's march is extremely fast. A better way to write the running time is $O(nh)$, where h is the number of convex hull vertices. In the worst case, $h = n$, and we get our old $O(n^2)$ time bound, but in the best case $h = 3$, and the algorithm only needs $O(n)$ time. Computational geometers call this an *output-sensitive* algorithm; the smaller the output, the faster the algorithm.

14.4 Divide and Conquer (Splitting)

The behavior of Jarvis's marsh is very much like selection sort: repeatedly find the item that goes in the next slot. In fact, most convex hull algorithms resemble some sorting algorithm.

For example, the following convex hull algorithm resembles quicksort. We start by choosing a *pivot* point p. Partitions the input points into two sets L and R, containing the points to the left of p, including p itself, and the points to the right of p, by comparing x-coordinates. Recursively compute the convex hulls of L and R. Finally, merge the two convex hulls into the final output.

The merge step requires a little explanation. We start by connecting the two hulls with a line segment between the rightmost point of the hull of L with the leftmost point of the hull of R. Call these points p and q, respectively. (Yes, it's the same p.) Actually, let's add *two* copies of the segment $\overline{pq}$ and call them *bridges*. Since p and q can 'see' each other, this creates a sort of dumbbell-shaped polygon, which is convex except possibly at the endpoints off the bridges.

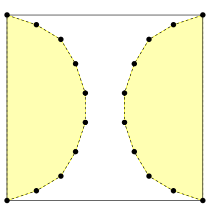


Merging the left and right subhulls.

We now expand this dumbbell into the correct convex hull as follows. As long as there is a clockwise turn at either endpoint of either bridge, we remove that point from the circular sequence of vertices and connect its two neighbors. As soon as the turns at both endpoints of both bridges are counter-clockwise, we can stop. At that point, the bridges lie on the *upper* and *lower common tangent* lines of the two subhulls. These are the two lines that touch both subhulls, such that both subhulls lie below the upper common tangent line and above the lower common tangent line.

Merging the two subhulls takes $O(n)$ time in the worst case. Thus, the running time is given by the recurrence $T(n) = O(n) + T(k) + T(n-k)$, just like quicksort, where k the number of points in R. Just like quicksort, if we use a naïve deterministic algorithm to choose the pivot point p, the worst-case running time of this algorithm is $O(n^2)$. If we choose the pivot point randomly, the expected running time is $O(n \log n)$.

There are inputs where this algorithm is clearly wasteful (at least, clearly to *us*). If we're really unlucky, we'll spend a long time putting together the subhulls, only to throw almost everything away during the merge step. Thus, this divide-and-conquer algorithm is *not* output sensitive.

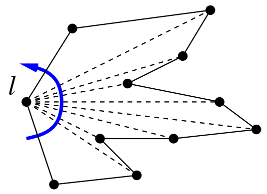


A set of points that shouldn't be divided and conquered.

14.5 Graham's Algorithm (Scanning)

Our third convex hull algorithm, called *Graham's scan*, first explicitly sorts the points in $O(n \log n)$ and then applies a linear-time scanning algorithm to finish building the hull.

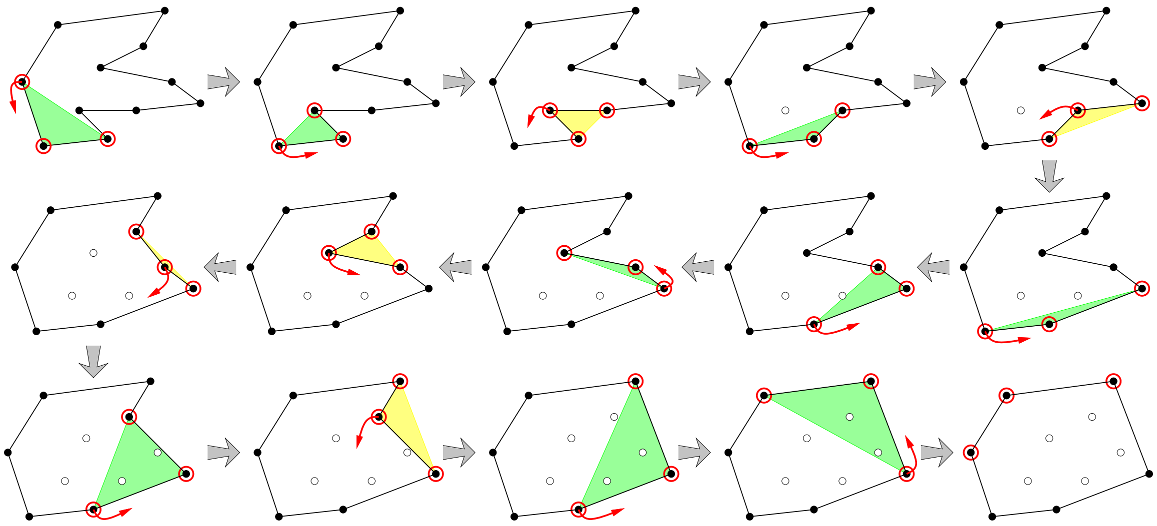
We start Graham's scan by finding the leftmost point ℓ, just as in Jarvis's march. Then we sort the points in counterclockwise order around ℓ. We can do this in $O(n \log n)$ time with any comparison-based sorting algorithm (quicksort, mergesort, heapsort, etc.). To compare two points p and q, we check whether the triple ℓ, p, q is oriented clockwise or counterclockwise. Once the points are sorted, we connected them in counterclockwise order, starting and ending at ℓ. The result is a *simple* polygon with n vertices.



A simple polygon formed in the sorting phase of Graham's scan.

To change this polygon into the convex hull, we apply the following 'three-penny algorithm'. We have three pennies, which will sit on three consecutive vertices p, q, r of the polygon; initially, these are ℓ and the two vertices after ℓ. We now apply the following two rules over and over until a penny is moved forward onto ℓ:

- If p, q, r are in counterclockwise order, move the back penny forward to the successor of r.
- If p, q, r are in clockwise order, remove q from the polygon, add the edge pr, and move the middle penny backward to the predecessor of p.



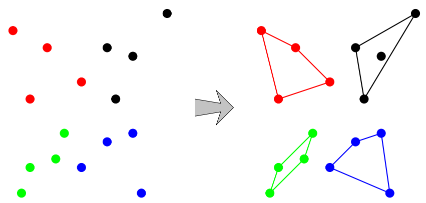
The 'three-penny' scanning phase of Graham's scan.

Whenever a penny moves forward, it moves onto a vertex that hasn't seen a penny before (except the last time), so the first rule is applied $n-2$ times. Whenever a penny moves backwards, a vertex is removed from the polygon, so the second rule is applied exactly $n-h$ times, where h is as usual the number of convex hull vertices. Since each counterclockwise test takes constant time, the scanning phase takes $O(n)$ time altogether.

14.6 Chan's Algorithm (Shattering)

The last algorithm I'll describe is an output-sensitive algorithm that is never slower than either Jarvis's march or Graham's scan. The running time of this algorithm, which was discovered by Timothy Chan in 1993, is $O(n \log h)$. Chan's algorithm is a combination of divide-and-conquer and gift-wrapping.

First suppose a 'little birdie' tells us the value of h; we'll worry about how to implement the little birdie in a moment. Chan's algorithm starts by *shattering* the input points into n/h arbitrary[1] subsets, each of size h, and computing the convex hull of each subset using (say) Graham's scan. This much of the algorithm requires $O((n/h) \cdot h \log h) = O(n \log h)$ time.

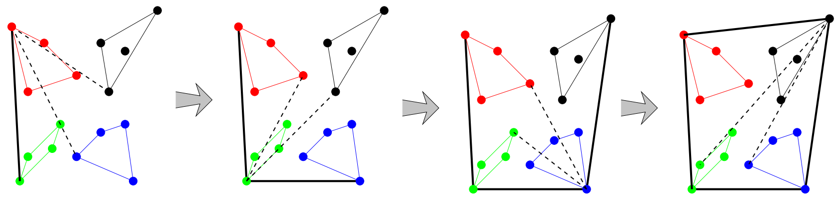


Shattering the points and computing subhulls in $O(n \log h)$ time.

Once we have the n/h subhulls, we follow the general outline of Jarvis's march, 'wrapping a string around' the n/h subhulls. Starting with $p = \ell$, where ℓ is the leftmost input point, we successively find the convex hull vertex the follows p and counterclockwise order until we return back to ℓ again.

The vertex that follows p is the point that appears to be furthest to the right to someone standing at p. This means that the successor of p must lie on a *right tangent line* between p and one of the subhulls—a line from p through a vertex of the subhull, such that the subhull lies completely on the right side of the line from p's point of view. We can find the right tangent line between p and any subhull in $O(\log h)$ time using a variant of binary search. (This is a practice problem in the homework!) Since there are n/h subhulls, finding the successor of p takes $O((n/h) \log h)$ time altogether.

Since there are h convex hull edges, and we find each edge in $O((n/h) \log h)$ time, the overall running time of the algorithm is $O(n \log h)$.



Wrapping the subhulls in $O(n \log h)$ time.

Unfortunately, this algorithm only takes $O(n \log h)$ time if a little birdie has told us the value of h in advance. So how do we implement the 'little birdie'? Chan's trick is to *guess* the correct value of h; let's denote the guess by h^*. Then we shatter the points into n/h^* subsets of size h^*, compute their subhulls, and then find the first h^* edges of the global hull. If $h < h^*$, this algorithm

[1]In the figures, in order to keep things as clear as possible, I've chosen these subsets so that their convex hulls are disjoint. This is not true in general!

computes the complete convex hull in $O(n \log h^*)$ time. Otherwise, the hull doesn't wrap all the way back around to ℓ, so we know our guess h^* is too small.

Chan's algorithm starts with the optimistic guess $h^* = 3$. If we finish an iteration of the algorithm and find that h^* is too small, we *square* h^* and try again. In the final iteration, $h^* < h^2$, so the last iteration takes $O(n \log h^*) = O(n \log h^2) = O(n \log h)$ time. The total running time of Chan's algorithm is given by the sum

$$O(n \log 3 + n \log 3^2 + n \log 3^4 + \cdots + n \log 3^{2^k}),$$

for some integer k. We can rewrite this as a geometric series:

$$O(n \log 3 + 2n \log 3 + 4n \log 3 + \cdots + 2^k n \log 3).$$

Since any geometric series adds up to a constant times its largest term, the total running time is a constant times the time taken by the last iteration, which is $O(n \log h)$. So Chan's algorithm runs in $O(n \log h)$ time overall, even without the little birdie.