An 'nem schönen blauen Sonntag, liegt ein toter Mann am Strand
Und ein Mensch geht um die Ecke, den man Mackie Messer nennt.
— Bertholt Brecht, "Die Moritat von Mackie Messer", *Die Dreigroschenoper* (1928)

First buffalo gal go around the outside, 'round the outside, 'round the outside.
Two buffalo gals go around the outside, 'round the outside, 'round the outside.
Three buffalo gals go around the outside.
Four buffalo gals go around the outside, 'round the outside, 'round the outside,
Four buffalo gals go around the outside, and dosey-do your partners.
— Malcolm McLaren, "Buffalo Gals", *Duck Rock* (1983)

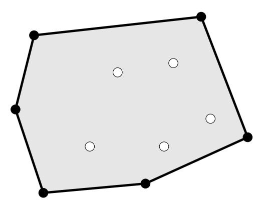
1 Convex Hulls

1.1 Definitions

Suppose we are given a set P of n points in the plane, and 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 and q inside the polygon, the entire line segment $\overline{pq}$ lies 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.

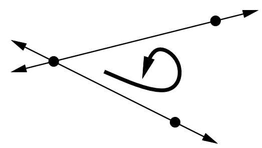
Just to make things concrete, we will represent the points in P by their Cartesian coordinates, in two arrays $X[1..n]$ and $Y[1..n]$. We will represent the convex hull as a circular linked list of vertices in counterclockwise order. If the ith point is a vertex of the convex hull, $next[i]$ is index 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, and our 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 collinear triples correctly, or at least consistently. Laster in the semester, we'll see a systematic method to do this.

1.2 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. Distinguishing between these two possibilities requires us to do a little algebra. This is a standard feature of most geometric algorithms. All the basic 'geometric' primitives evaluate *algebraic* functions of the input coordinates; once these primitives are in place, the rest of the 'geometric' algorithm is purely combinatorial.



Three points in 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 collinear (remember, we're assuming that never happens), then the two expressions are equal.

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 think about this orientation test in terms of three-dimensional vectors. Recall that the cross product $u \times v$ of two vectors in $\mathbb{R}^3$ is defined according to the right-hand rule; the vectors u, v, and $u \times v$ are oriented counterclockwise around the origin. Thus, three vectors u, v, and w are oriented counterclockwise if and only if triple product $(u \times v) \cdot w$ is positive. If we put our original points are in the plane $z = 1$, their orientation is defined by the sign of the triple product $((a, b, 1) \times (c, d, 1)) \cdot (e, f, 1)$, which can in turn be written as a 3×3 determinant:

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

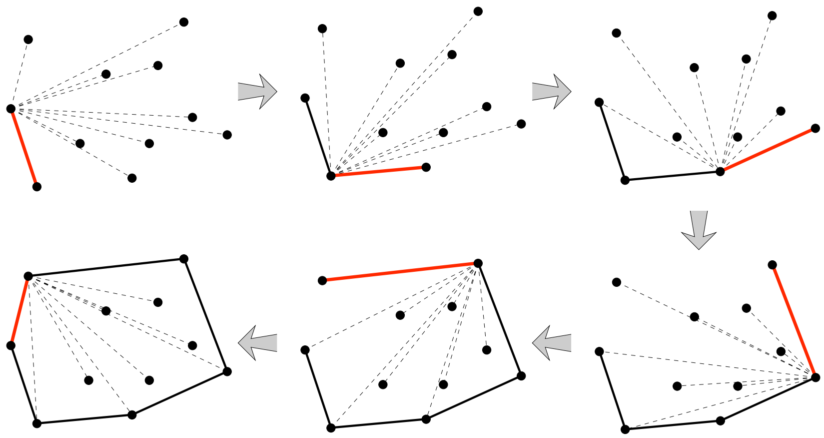
All three boxed expressions are algebraically identical.

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.

1.3 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 (more formally, the point whose x-coordinate is smallest); this point must be a convex hull vertex. We can clearly find this point in $O(n)$ time.



The execution of Jarvis's March.

The algorithm then performs a series of *pivoting* steps to find each successive convex hull vertex, starting with ℓ and continuing until it reaches ℓ again. The vertex immediately following a point p is the point that appears 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 counterclockwise order. We can find each successive vertex in linear time by performing a series of $O(n)$ counterclockwise 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
```

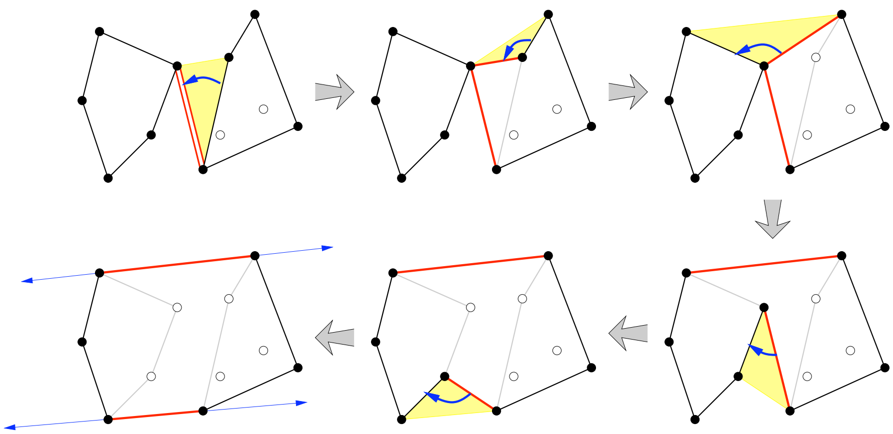
Since the algorithm spends $O(n)$ time for each convex hull vertex, the worst-case running time is $O(n^2)$. However, this naïve analysis hides the fact that if the convex hull has very few vertices, Jarvis's march is extremely fast. A better way to write the running time is $O(nh)$, where h is the number 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.

1.4 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.

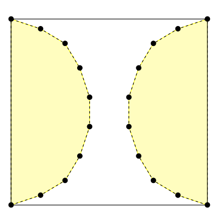


Merging the left and right subhulls.

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 counterclockwise, 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.

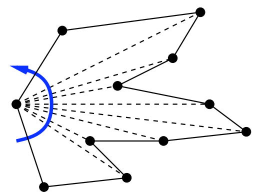


A set of points that shouldn't be divided and conquered.

1.5 Graham's Algorithm (Das Dreigroschenalgorithmus)

Our next 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.

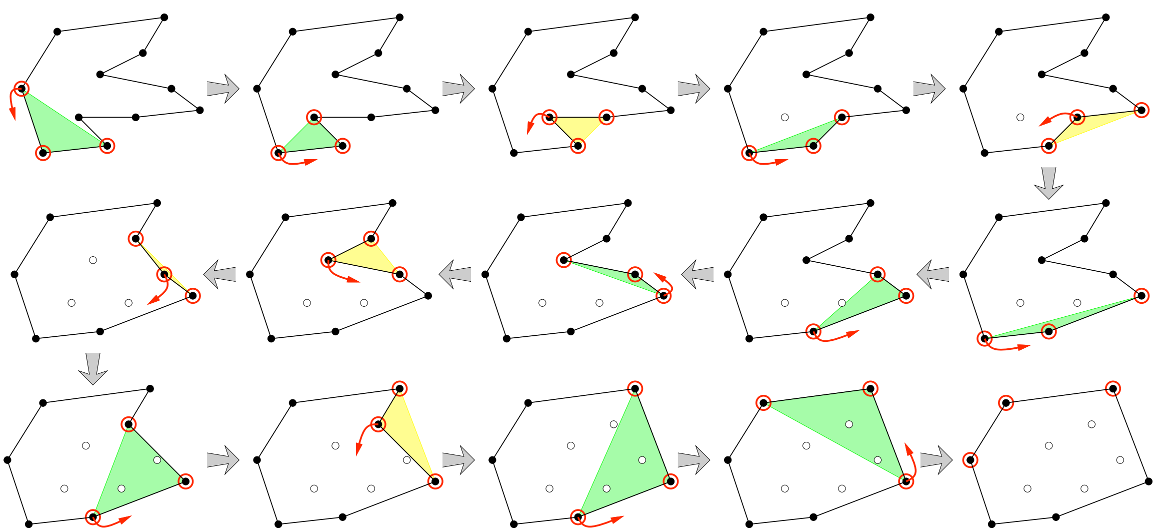
We start Graham's scan by finding the leftmost point ℓ, just as in Jarvis's march. Then we sort the points in counterclockwise order around ℓ. We can do this in $O(n \log n)$ time with any comparison-based sorting algorithm (quicksort, mergesort, heapsort, etc.). To compare two points p and q, we check 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.

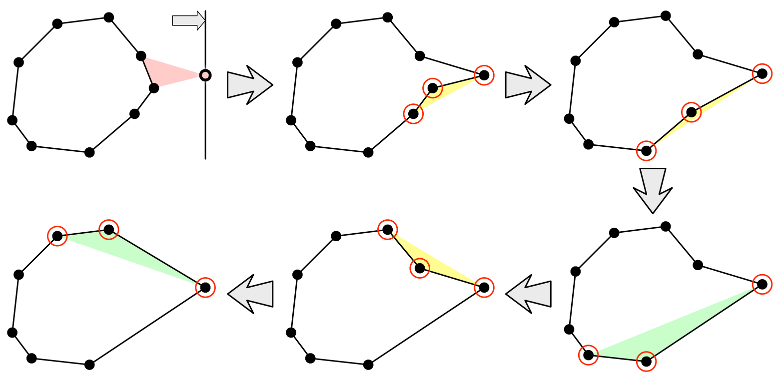
Whenever a penny moves forward, it moves onto a vertex that hasn't seen a penny before (except the last time), so the first rule is applied $n-2$ times. Whenever a penny moves backwards, a vertex is removed from the polygon, so the second rule is applied exactly $n-h$ times, where h is as usual the number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

1.6 Incremental Insertion (Sweeping)

Another convex hull algorithm is based on a design principle shared by many geometric algorithms: the *sweep line*. Imagine continuously sweeping a vertical line from left to right over the points. At all times, our algorithm maintains the convex hull of the points that have already been swept over; as the sweep line encounters each new point, the convex hull is updated to include that new point.

Of course, we cannot *actually* move the line continuously from left to right on a digital computer. Instead, any sweep-line algorithm considers only the discrete set of positions of the line, called *events*, where the invariant we want to maintain changes. In our case, these are just the x-coordinates of the input points. We want to consider these *events* in increasing order, so our algorithm begins by sorting the points by their x-coordinates.

To insert a new point, we first connect it to the rightmost point in the convex hull, as well as one of that point's neighbors, to create a simple, but not necessarily convex, polygon. We then repeatedly remove concave corners from this polygon, just as in Graham's scan. The way we construct the polygon guarantees that any concave vertices are adjacent to the newly added point, which implies that we can find and remove each concave vertex in $O(1)$ time.



Inserting a new rightmost point into a convex hull.

```
SweepHull(X[1..n], Y[1..n]):
  Sort X and permute Y to match

  next[1] ← 2; prev[2] ← 1
  next[2] ← 1; prev[1] ← 2

  for i ← 3 to n
      if Y[i] > Y[i−1]
          next[i] ← i−1; prev[i] ← prev[i−1]
      else
          prev[i] ← i−1; next[i] ← next[i−1]

      next[prev[i]] ← i; prev[next[i]] ← i

      while CCW(i, prev[i], prev[prev[i]])
          next[prev[prev[i]]] ← i
          prev[i] ← prev[prev[i]]

      while CCW(next[next[i]], next[i], i)
          prev[next[next[i]]] ← i
          next[i] ← next[next[i]]
```

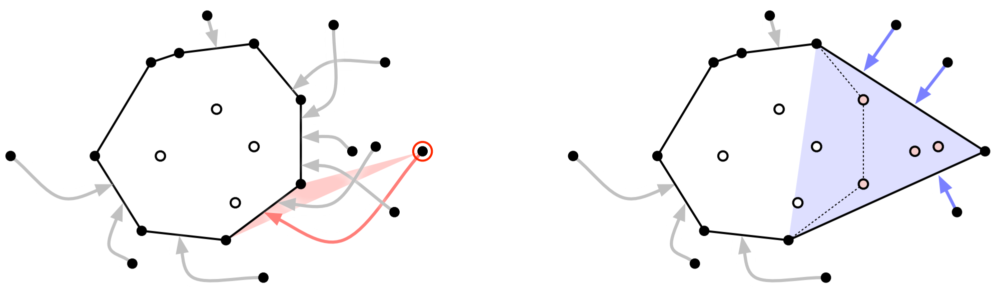
A single iteration of the main loop takes $\Theta(n)$ time in the worst case, because we may delete up to $i-3$ points. However, each point is deleted from the evolving hull at most once, so in fact the entire sweep requires only $O(n)$ time after the points are sorted. Thus, the overall running time of this algorithm, like Graham's scan, is $O(n \log n)$.

1.7 Randomized Incremental Insertion

Each point maintains a pointer to an arbitrary visible edge (null if inside). $\Theta(n^2)$ changes in the worst case, but easy backwards analysis shows $O(n \log n)$ expected changes if points are inserted in random order.

Each point p maintains a pointer $e(p)$ to some *visible* edge of the evolving convex hull; we call this the *conflict edge* for p. If p is already inside the evolving convex hull, then $e(p)$ is a null pointer. These pointers are bidirectional; thus, every edge e has a *conflict list* of all points for which e is the conflict edge.

We insert each new point p by splicing it into the endpoints of edge $e(p)$ and then removing concave vertices, exactly as in our earlier sweep-line algorithm. Whenever we remove an edge of the polygon—either $e(p)$ or an edge adjacent to a concave vertex—we mark each point on its conflict list. After the polygon has been completely repaired, we assign a new conflict edges to each marked point q as follows. If q lies inside the triangle formed by p and its two neighbors on the polygon, then q is an interior point, so we can set $e(q)$ to NULL. Otherwise, at least one of the edges adjacent to p is visible to q; assign $e(q)$ to one such edge (chosen arbitrarily if q can see both). Each change to a conflict pointer $e(q)$ requires constant time.

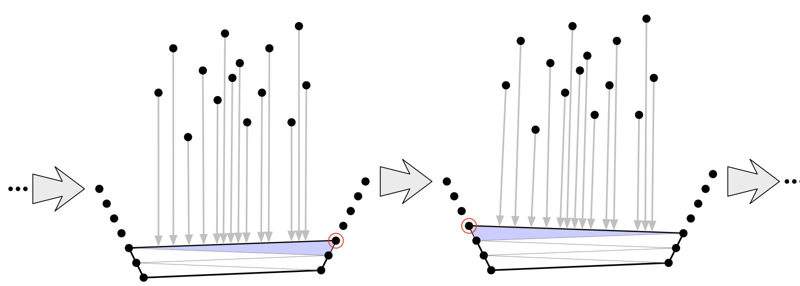


Inserting a point into a convex hull and updating conflict pointers for uninserted points.

The running time of each phase is dominated by (1) the number of concave corners removed and (2) the number of changes to conflict pointers; everything else clearly takes $O(n)$ time. Each point can be removed from the polygon at most once, so the number of corner removals is clearly $O(n)$. Thus, to complete the analysis of our algorithm, we only need to determine the total number of conflict pointer changes.

$$T(n) = O(n) + O(\text{number of conflict pointer changes})$$

Unfortunately, in the worst case, a constant fraction of the pointers can change in a constant fraction of the insertions, leading to $\Omega(n^2)$ changes overall. Suppose the input point set consists of two columns, each containing $n/4$ points, along with a cloud of $n/2$ points between and above both columns. If the points are inserted in order from bottom to top, *every* point in the cloud changes its conflict pointer at each of the first $n/2$ insertions.



Inserting points in the wrong order can take $\Omega(n^2)$ time.

However, if we insert the points *in random order*, the *expected* total number of conflict changes is only $O(n \log n)$. To establish this claim, all we need to do is bound the probability that a particular point's conflict edge changes at a particular iteration of the insertion algorithm. For any pair of indices i and j, let $e_i(p_j)$ denote the conflict edge for point $p_j \in P$ after the first i random points have been inserted. If point p_j has already been inserted, or is already inside the evolving hull, then $e_i(p_j) = \textsc{Null}$. We now easily observe that

$$\mathrm{E}[\text{number of conflict pointer changes}] \le \sum_{i=1}^{n} \sum_{j=1}^{n} \Pr[e_i(p_j) = e_{i-1}(p_j)].$$

But how do we compute the probability that a given conflict pointer changes at a given iteration? The easiest method is called *backwards analysis*—imagine the algorithm running backward in time. In this backward view, the algorithm *deletes* the input points one at a time in random order, maintaining the convex hull of the points that have not yet been deleted, along with conflict pointers for the points that *have* been deleted. A given deletion changes the conflict pointer $e(q)$ only under the following circumstances:

- If q was a vertex of the polygon before the deletion, then q was the deleted point.
- If q was inside the polygon before the deletion, then it was outside afterward.
- If q was outside the polygon before the deletion, then one of the endpoints of $e(q)$ was the deleted point.

The first and second cases happen at most once per point, and thus at most n times altogether. To bound the probability of the third case, we observe that our backwards algorithm (ɯɥʇᴉɹoƃlɐ?) deletes a *random* point at each iteration. Specifically, in the ith iteration (counting backward), the probability that any particular point is deleted is *exactly* $1/i$. **Thus, the probability that an exterior point's conflict pointer changes is *precisely* $2/i$.** We conclude:

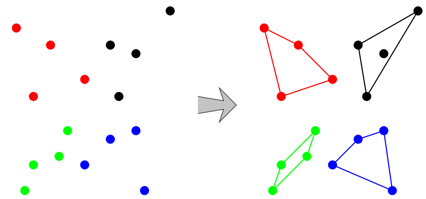
$$\begin{aligned}
\mathrm{E}[\text{number of conflict pointer changes}] &\le \sum_{i=1}^{n} \sum_{j=1}^{n} \Pr[e_i(p_j) = e_{i-1}(p_j)] \\
&\le n + \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{2}{i} \\
&= n + 2nH_n = \boxed{O(n \log n)}
\end{aligned}$$

1.8 Chan's Algorithm (Shattering)

Our next algorithm 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.

[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!

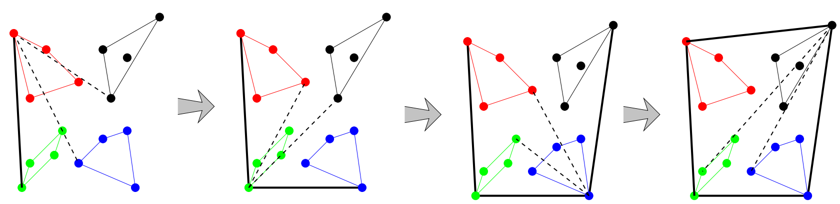


Shattering the points and computing subhulls in $O(n \log h)$ time.

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. (Details are left as an exercise.) 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 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. Thus, in the ith iteration, we have $h^* = 3^{2^i}$. 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

$$\sum_{i=1}^{k} O(n \log 3^{2^i}) = O(n) \cdot \sum_{i=1}^{k} 2^i$$

for some integer k. 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.

1.9 Prune and Search (Filtering)

We can also achieve $O(n \log h)$ running time using a variant of the earlier divide-and-conquer algorithm, called *QuickHull*. This algorithm avoids the $O(n^2)$ worst-case running time by choosing an approximately balanced partition of the points; at the same time, whenever possible, the algorithm prunes away a large subset of the interior points before recursing.

The basic QuickHull algorithm starts by identifying the leftmost and rightmost points, which we will call a and z. We then find the convex hulls of the points on either side of the line $\overleftrightarrow{az}$, using the following recursive subroutine.

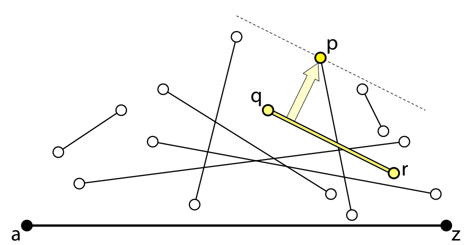
<u>QuickHull(P, a, z):</u>
  ⟨⟨*Precondition: Every point in P lies to the left of $\overrightarrow{az}$*⟩⟩
  if $P = \varnothing$
      $next[z] \leftarrow a$; $pred[a] \leftarrow z$
  else
      $p \leftarrow$ point in P furthest to the left of $\overrightarrow{az}$
      $L \leftarrow$ all points in P to the left of $\overrightarrow{ap}$
      $R \leftarrow$ all points in P to the left of $\overrightarrow{pz}$
      QuickHull(L, a, p)
      QuickHull(R, p, z)

Except for the recursive calls, QuickHull clearly runs in $O(n)$ time. The point r is guaranteed to be a convex hull vertex. Thus, each call to QuickHull discovers either a vertex or an edge of the convex hull, so the overall running time is $O(nh)$, just like Jarvis's march. More importantly, any points inside the triangle $\triangle apz$ cannot be convex hull vertices; the QuickHull algorithm simply discards these points. As a result of this filtering, this algorithm is blindingly fast in practice.

Unfortunately, the $O(nh)$ running time analysis is tight in the worst case; because the splitting is performed deterministically, an adversarial input can force unbalanced splits, just like quicksort. To avoid these problems, we make a few minor changes to the algorithm, using the so-called *prune and search* paradigm.[2]

To simplify the presentation slightly, I will assume that the line $\overleftrightarrow{az}$ is horizontal, and that a lies to the left of z. This assumption can be enforced by a change of coordinates, but in fact, if everything is implemented using the correct algebraic primitives, no such transformation is necessary.

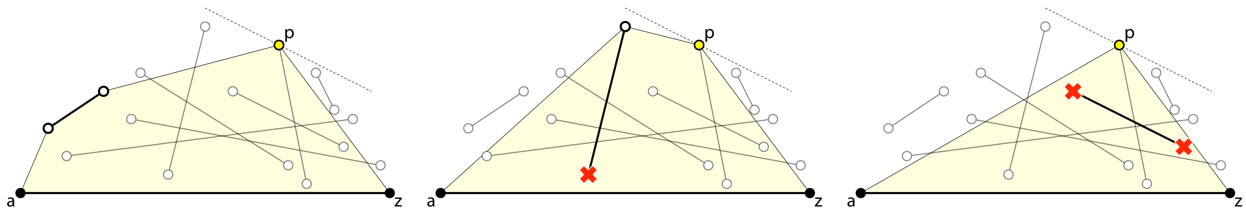
We start by arbitrarily pairing up the points in P. Let q, r be the pair whose connecting line has *median slope* among all $n/2$ pairs; we can find this pair in $O(n)$ time, even without sorting slopes. We now choose the pivot point p to be the point in P *furthest above the line* $\overleftrightarrow{qr}$, rather than the point furthest above the baseline $\overleftrightarrow{az}$.



Segment qr has median slope among all $n/2$ pairs, and p is the point furthest above $\overleftrightarrow{qr}$.

Next we *prune* the set P as follows: For each pair s, t of points in P, we discard any point in the subset $\{a, z, p, s, t\}$ that is inside the convex hull of those five points. This pruning step automatically discards any points inside the triangle $\triangle apz$, just like the earlier algorithm, but other points may be removed as well. The remainder of the algorithm is unchanged.

[2]However, unlike quicksort, we can't avoid unbalanced splits just by randomly permuting the points. The pivot point is chosen according to its position *in the plane*, not its position in the input array. The prune-and-search algorithm could be simplified by a solution to the following open problem: ***Given a set of n points in the plane, find a random vertex of the convex hull in $O(n)$ expected time.*** Specifically, each of the h hull vertices should be returned with probability (roughly) $1/h$.



Pruning pairs of points. Left: Both points survive. Middle: One point survives. Right: Both points discarded.

```
PrunedQuickHull(P, a, z):
  ⟨⟨Precondition: az→ points directly to the right⟩⟩
  ⟨⟨Precondition: Every point in P is above az→⟩⟩
  if |P| < 5
      use brute force
  else
      Arbitrarily pair up the points in P
      q, r ← pair with median slope among all pairs
      p ← point in P furthest above ↔qr
      for each pair s, t
          Discard any interior point from {a, z, p, s, t}
      L ← all points in P above ↔ap
      R ← all points in P above ↔pz
      PrunedQuickHull(L, a, p)
      PrunedQuickHull(R, p, z)
```

Now I claim that in the first recursive call, ***the subset L contains at most $3n/4$ points***. Consider a pair s, t from our arbitrary pairing of P. If both of these points are in L, they must satisfy two conditions: (1) they both lie to the left of $\overrightarrow{ap}$, and (2) the slope of st is larger than the slope of qr. By definition, less than half the pairs have slope larger than the median slope. Thus, in at least $n/4$ pairs, at least one point is *not* in L. Symmetrically, the subset R also contains at most $3n/4$ points.

This simple observation about the sizes of L and R implies that the running time of the modified algorithm is only $O(n \log h)$. The running time is described by the two-variable recurrence

$$T(n,h) = O(n) + T(n_1,h_1) + T(n_2,h-h_1),$$

where $n_1 + n_2 \le n$ and $\max\{n_1,n_2\} \le 3n/4$. Let's assume that the $O(n)$ term is at most αn for some constant α. The following inductive proof implies that $T(n,h) \le \beta n \ln h$ for some constant β.

$$\begin{aligned}
T(n,h) &\le \alpha n + T(n_1,h_1) + T(n_2,h-h_1) \\
&\le \alpha n + \beta n_1 \ln h_1 + \beta n_2 \ln(h-h_1) && \text{[inductive hypothesis]} \\
&\le \alpha n + \beta n_1 \ln h_1 + \beta (n-n_1) \ln(h-h_1) && [n_1 + n_2 \le n] \\
&\le \alpha n + \beta \frac{3n}{4} \ln h_1 + \beta \frac{n}{4} \ln(h-h_1) && [n_1 \le 3n/4] \\
&\le \alpha n + \beta \frac{n}{4} (3 \ln h_1 + \ln(h-h_1)) && \text{[algebra]} \\
&\le \alpha n + \beta \frac{n}{4} (\max_x (3 \ln x + \ln(h-x))) && \text{[algebra]}
\end{aligned}$$

The function $3 \ln x + \ln(h-x)$ is maximized when its derivative is zero:

$$\frac{d}{dx}(3 \ln x + \ln(h-x)) = \frac{3}{x} - \frac{1}{h-x} = 0 \implies x = \frac{3h}{4}$$

Thus,

$$\begin{aligned} T(n,h) &\le \alpha n + \beta \frac{n}{4}\left(3\ln\frac{3h}{4} + \ln\frac{h}{4}\right) \\ &\le \alpha n + \beta n \ln h + \beta n \left(\frac{3}{4}\ln\frac{3}{4} + \frac{1}{4}\ln\frac{1}{4}\right) \\ &\le \beta n \ln h + (\alpha - 0.562336\beta)n. \end{aligned}$$

If we assume $\beta \ge 2\alpha$, the induction proof is complete.

Although we can deterministically compute the median slope in $O(n)$ time, the algorithm is rather complicated, and the constant factor hidden in the $O(\,)$ notation is quite large. A randomized median-selection algorithm is both simpler and more efficient in practice. But if we're using a randomized algorithm, why not go all out? In fact, to get an *expected* running time of $O(n \log h)$, it suffices to choose the points q and r uniformly at random from P.

Exercises

*Starred problems are more challenging.

1. Many convex hull algorithms closely resemble well-known algorithms for sorting: for example, Jarvis's march resembles selection sort, and QuickHull resembles quicksort. Describe a convex hull algorithm that closely resembles mergesort: Arbitrarily partition the set of points into two subsets of equal size, recursively compute the convex hull of each subset, and then merge the two sub-hulls into a single convex hull. How do we merge two subhulls in $O(n)$ time?

2. Let P be a convex polygon with n vertices, and let p be a point outside P. Recall that the *right tangent line* between p and P is a line $\overleftrightarrow{pq}$, where q is a vertex of P, and for every other vertex r of P, the points p, q, r are oriented clockwise.

   (a) Describe an algorithm to find the right tangent line between p and P in $O(\log n)$ time. You will need an auxiliary data structure in addition to the standard doubly-linked list of vertices of P. This data structure should have size $O(n)$ and should be constructible in $O(n)$ time.

   (b) What does your algorithm do if p happens to lie inside P? Modify your algorithm to handle this case correctly, without increasing its running time.

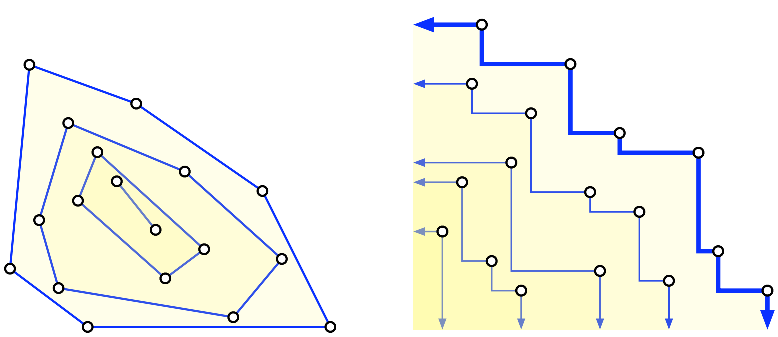
   (c) Now suppose we add a new point to P and update its convex hull. Describe how to update your auxiliary data structures from parts (a) and (b) in $O(\log n)$ time.

   (d) Describe an incremental convex hull algorithm, based on parts (a)–(c), that runs in $O(n \log n)$ time.

3. The *convex layers* of a point set X are defined by repeatedly computing the convex hull of X and removing its vertices from X, until X is empty.

   (a) Describe an algorithm to compute the convex layers of a given set of n points in the plane in $O(n^2)$ time.

   *(b) Describe an algorithm to compute the convex layers of a given set of n points in the plane in $O(n \log n)$ time.



Left: The convex layers of a set of points; see problem 3.
Right: The staircase (thick line) and staircase layers (all lines) of a set of points; see problems 4 and 5.

4. Let X be a set of points in the plane. A point p in X is *Pareto-optimal* if no other point in X is both above and to the right of p. The Pareto-optimal points can be connected by horizontal and vertical lines into the *staircase* of X, with a Pareto-optimal point at the top right corner of every step. See the figure above.

   (a) QUICKSTEP: Describe a divide-and-conquer algorithm to compute the staircase of a given set of n points in the plane in $O(n \log n)$ time.

   (b) SCANSTEP: Describe an algorithm to compute the staircase of a given set of n points in the plane, sorted in left to right order, in $O(n)$ time.

   (c) NEXTSTEP: Describe an algorithm to compute the staircase of a given set of n points in the plane in $O(nh)$ time, where h is the number of Pareto-optimal points.

   (d) SHATTERSTEP: Describe an algorithm to compute the staircase of a given set of n points in the plane in $O(n \log h)$ time, where h is the number of Pareto-optimal points.

   In all these problems, you may assume that no two points have the same x- or y-coordinates.

5. The *staircase layers* of a point set are defined by repeatedly computing the staircase and removing the Pareto-optimal points from the set, until the set becomes empty.

   (a) Describe and analyze an algorithm to compute the staircase layers of a given set of n points in $O(n^2)$ time.

   *(b) Describe and analyze an algorithm to compute the staircase layers of a given set of n points in $O(n \log n)$ time.

   (c) An *increasing subsequence* of a point set X is a sequence of points in X such that each point is above and to the right of its predecessor in the sequence. Describe and analyze an algorithm to compute the *longest* increasing subsequence of a given set of n points in the plane in $O(n \log n)$ time. *[Hint: There is a one-line solution that uses part (b). But why is is correct?]*

6. Consider the two-variable recurrence that describes the running time of QUICKHULL:

$$T(n,h) = T(m,g) + T(n-m,h-g) + O(n).$$

   (a) Suppose we can guarantee, at every level of recursion, that $\alpha n \le m \le (1-\alpha)n$ for some constant $0 < \alpha < 1$. Prove that $T(n,h) = O(n \log h)$.

   (b) Suppose we can guarantee, at every level of recursion, that $\alpha h \le g \le (1-\alpha)h$ for some constant $0 < \alpha < 1$. Prove that $T(n,h) = O(n \log h)$.

   (c) Suppose m is chosen uniformly at random from the range $1 \le m \le n-1$, independently at each level of recursion. Prove that $\mathrm{E}[T(n,h)] = O(n \log h)$.

   (d) Suppose g is chosen uniformly at random from the range $1 \le g \le h-1$, independently at each level of recursion. Prove that $\mathrm{E}[T(n,h)] = O(n \log h)$.

   (e) Now suppose we modify the recurrence slightly:

$$T(n,h) = T(m,g) + T(n-m,h-g) + O(\min\{m,n-m\})$$

   Prove that $T(n,h) = O(n \log h)$, regardless of the values of m and g.