Finding Convex Hulls - part I

1 Introduction

The convex hull of n points is the smallest convex polygon which contains all n points. Imagine trying to wrap a cord around the points. The shape of the cord is actually the convex hull.

There are many ways of finding the convex hull. The first approach is to determine the points which are interior to the hull and throw them out. This is done by taking subsets of three points and throwing out all the points interior to the triangle described by the three points. More formally:

```
naiveHull(S):
    for all triples xyz ∈ S:
        for all w ∈ S:
            if w ⊆ △xyz:
                throw out w
```

Doing this for all possible subsets would remove all points not on the hull leaving only in the hull. Finding all triples takes $O(n^3)$. Searching all points to see if they are interior to the triple takes $O(n)$ per triple. The total time for the algorithm is thus $O(n^4)$. Computing the order of the vertices on the hull can be done in $O(h \log(h))$ time where h is the number of hull vertices: compute the centroid c of any three points; sort the vertices by slope around c; since c is in the interior of the hull, this sorted order corresponds to the order on the hull.

So convex hull must be at most $O(n^4)$, but what is the lower bound for the time complexity of any convex hull algorithm? To calculate this, we compare convex hull to sorting.

2 Lower Bound for Convex Hull

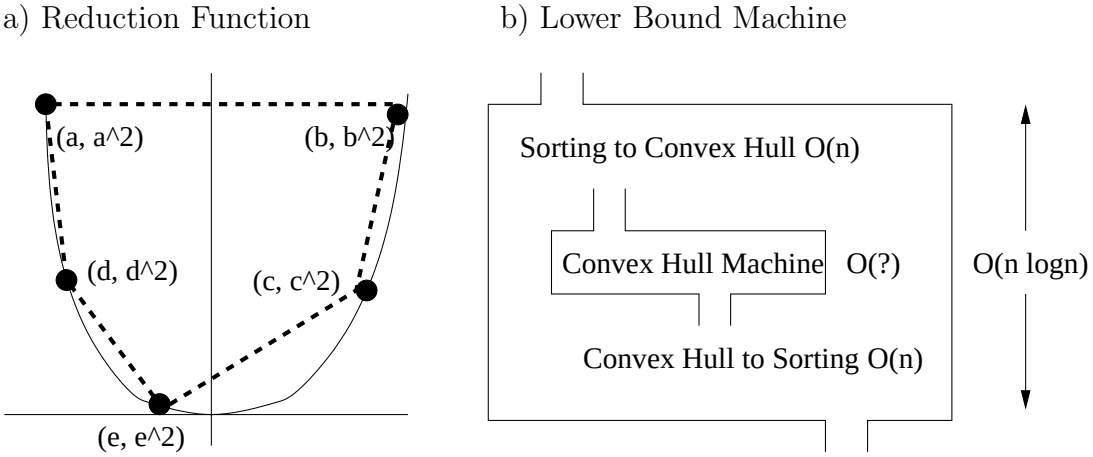


Figure 1: Time complexity and Reduction function for Convex Hull

Assume that we are given $\{x_1, ..., x_n\}$ numbers to sort. We map each x_i to the point (x_i, x_i^2) in linear time and use the convex hull machine to compute the convex hull of the new set of points.

These points lie on the convex parabola $y = x^2$, so the convex hull contains the input points sorted by their x-coordinate [fig 1a]. We can clearly remap the convex hull back to the sorted set of original points using $f(x_i, y_i) = x_i$ in linear time.

The lower bound for convex hull must therefore be $\Omega(n \log(n))$. If it was better than this, we could perform sorting in less than $O(n \log(n))$ and would have a contradiction [fig 1b].

3 Jarvis March / Giftwrapping Algorithm

We have a large gap between our upper bound of $O(n^4)$ and our lower bound of $\Omega(n \log(n))$. This next algorithm, called *giftwrapping* or the *Jarvis March*, begins to close the gap. Jarvis march starts at the point with minimum x value and successively adds the point with maximum slope relative to the current point.

Note that we calculate slope as $\frac{dy}{dx}$. It is positive in the first and third quandrants and negative in the second and fourth quadrants.

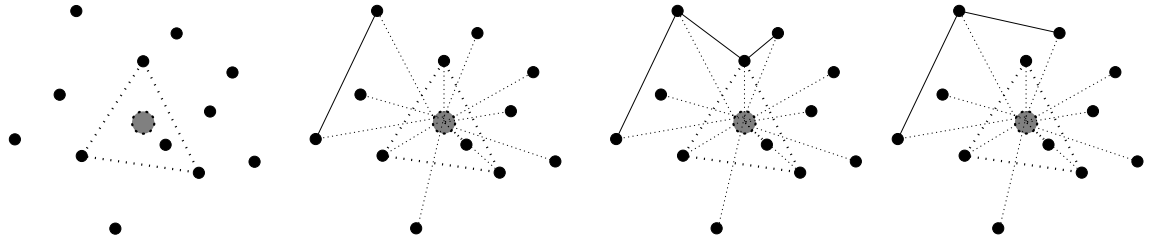
Let $min_x(S)$ and $max_x(S)$ give the points in S with min and max x values respectively. Let $maxSlopeRight(S, p)$ return the point in S with max slope relative to p with x value greater than or equal to p. Let $maxSlopeLeft(S, p)$ similarly return the point in S with max slope relative to p with x value less than or equal to p.

```
jarvisMarch(S):
    p ← min_x(S):
    while p ≠ max_x(S):
        append(p, hull)
        p ← maxSlopeRight(S, p)
    while p ≠ min_x(S):
        append(p, hull)
        p ← maxSlopeLeft(S, p)
```

This requires $O(n^2)$ time in the worst case. However, if h is the number of points actually on the hull, the algorithm requires $O(n \cdot h)$ time. For many distributions of points in the plane, this will be much better than $O(n^2)$.

4 Graham's Scan

Figure 2: Illustration of Graham's Scan



a) Centroid b) Sort and first edge c) First two steps d) Pop and Push

Graham's scan gives an optimal ($O(n\log(n))$) algorithm for finding the convex hull. The algorithm for a set S and associated time complexity is:

1. Take 3 points from S and compute their centroid c. $O(1)$ [fig 2a]
2. Sort each point in S by it's slope relative to c. $O(n\log(n))$
3. Find the leftmost point in S, called l. $O(n)$
4. Find the first Jarvis March edge at l, called j. $O(n)$ [fig 2b]
5. Push l and j onto the stack T. $O(1)$
6. Perform the following [fig 2c,2d]:

```
p ← j
while p ≠ l:
    p ← next point in sorted order
    while (top 2 points in T and p) do not form a right hand turn:
        pop the top element off of T
    push p onto T
```

Note that each point can only be pushed on the stack once and popped off the stack once. This implies that the time complexity for this loop is $O(n)$ for all the points.

The overall time complexity is $O(n\log(n))$. More interestingly, the time complexity is $O(n)$ if the points are presorted.