CS 6301.008.18S Lecture—January 11, 2017

Main topics are #convex_hulls, #lower_bounds, and #output_sensitivity.

Prereq Forms

- Please fill out prerequisites forms and turn them in before you leave class today. Thanks!
- The course has one official prerequisite: CS 5343—Algorithm Analysis and Data Structures.

Graham Scan Proof

- On Tuesday, we were considering the following problem: Given a set of points P subseteq R^2 in the plane, compute their convex hull. Here, n := |P|.
- The convex hull is the smallest convex polygon containing the points. Its vertices all come from P.
- And one way to represent it is to make a list of its vertices in counterclockwise order.
- The leftmost and rightmost points of P belong to the convex hull.
- In class we focused on finding the upper hull, the set of points between the rightmost and leftmost when going in counterclockwise order.
- We looked at the following algorithm, which ran in O(n log n) time.
- UpperHull(P):
  - Sort points by x-coordinate increasing to form <p_1, …, p_n>
  - push p_1 and p_2 into S
  - for i ← 3 to n
    - while (|S| ≥ 2 and <p_i, S[top], S[top - 1]> make a right-hand turn
      - pop from S
    - push p_i into S

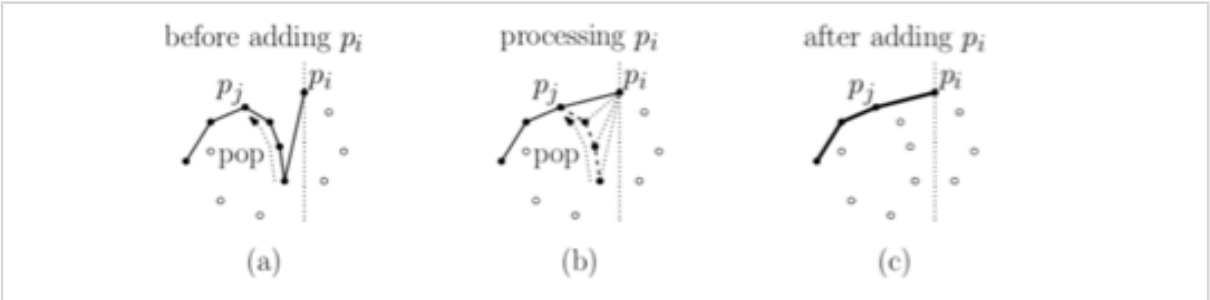
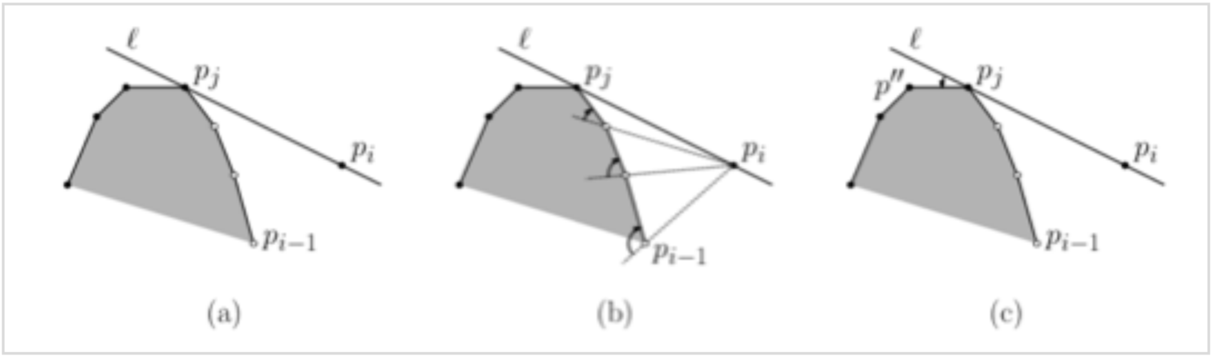


Fig. 14: Graham's scan.

- Let's start today by arguing that the algorithm is correct. Then, we'll discuss whether we can find a faster algorithm.
- In short, we need to argue that it is correct to do those pops, and it is correct to stop when we reach the first left-hand turn.
- Claim: After inserting p_i, the vertices of S from top to bottom form the upper hull of P_i =

{p_1, …, p_i}.

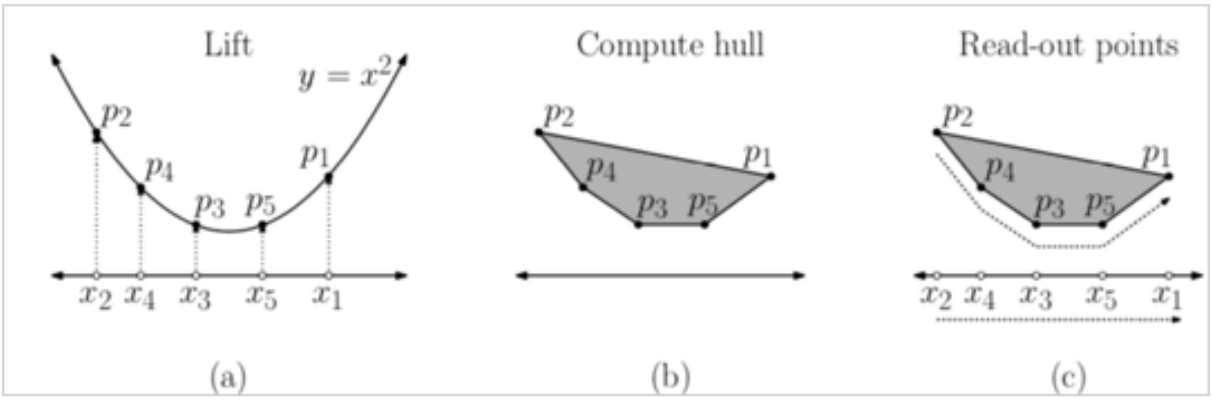
- Proof:
    - Claim clearly true after inserting p_2 since the upper hull uses both p_1 and p_2.
    - So, p_i must be in the upper hull of P_i since it's furthest to the right.
    - Let p_j be the next point to the left on the upper hull of P_i.



    - No point z of P_i lies *above* the line through p_j and p_i; segments zp_i and z_pj contain points above the upper hull.
    - But that mean p_j is on the upper hull of P_{i-1}; otherwise there would be points above it. In particular, p_j is in S when we enter the for loop the ith time by induction.
    - Now for each p_k where j < k < i, we have the turn from p_i, p_k, and p_k's predecessor on the hull of P_{j-1} forms a right-hand turn.
    - Each p_k lies strictly below the line, so it is correct to pop it off the stack. And the turn at p_j is left-hand, so it doesn't get popped. Good.
    - Finally, we push p_i into S giving us the final correct hull.

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. 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, … x_n} with each x_n in 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.
- However, 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 (Gift-Wrapping)

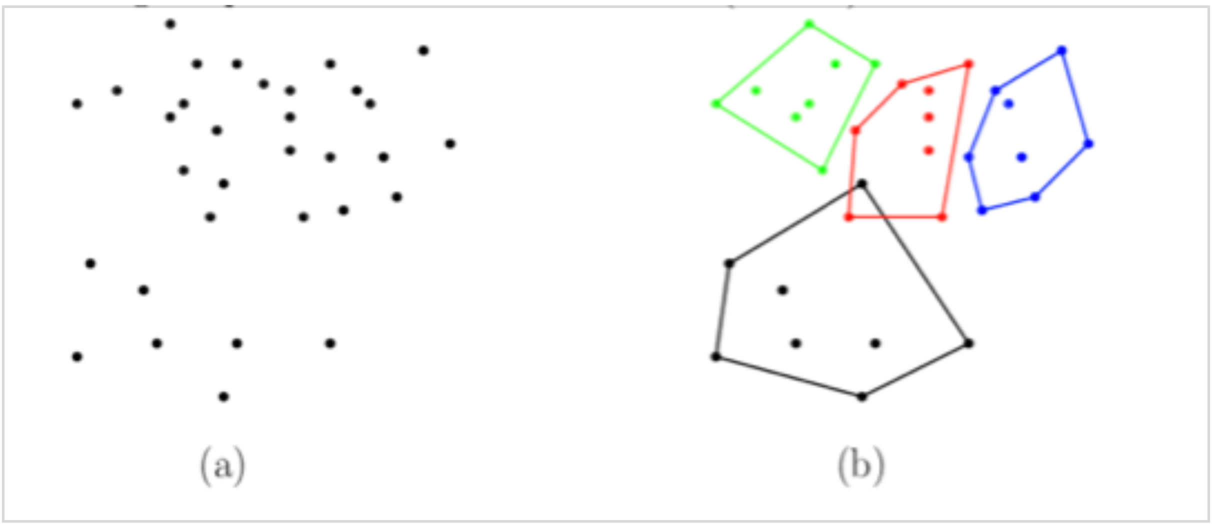
- Let's consider a different algorithm by Jarvis ('73).
- The idea behind the algorithm is we start with some point we know must be on the convex hull, and then we perform linear time searches for each next point along the hull.
- We already discussed how the leftmost point is on the hull, so we can start there.
- Now, suppose we've just added *some* point p to the hull (not necessarily the leftmost one) and we want to know what's next.
- From the perspective of the point we're at, the other points can be ordered *cyclicly* from left to right, with the "leftmost" point being the previous one on the convex hull and the "rightmost" point being the one we want.
- One nice observation is if you take two other points q and r, r is "further to the right" relative to p if and only if <p, q, r> is a right-hand turn. So we have a way to compare pairs of points q and r. The next point we want is a max using these comparisons.
- Here's code for the algorithm.
- JarvisMarch(P = {p_1, …, p_n}):

```
ell ← leftmost point of P
p ← ell
add p to CH
i ← 2
repeat
    r ← max point ≠ p where q < r iff <p, q, r> is a right-hand turn
    if p = ell, break
    else, add p to CH
```

- In the *worst-case* this algorithm runs in O(n^2) time.
- But, if h is the number of points on the convex hull, it only runs in 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 better than 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 O(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) k log k) = O(n log k) time. For any constant c and k ≤ h^c, this running time is O(n log h).
- In fact, we can even have time to run Graham's scan on these subsets.
- So here's what we'll do: Suppose we know some k where h ≤ k ≤ h^c.
- We'll discuss how to do this guess later.
- We'll break up the input set P and run Graham's scan on each. This will create a collection of n / k mini-hulls.



- These mini-hulls are useful for a couple reasons.
- First, any points from a subset that aren't vertices of its mini-hull aren't vertices of the convex hull of P either. We can safely ignore them.
- Second, the mini-hulls have a lot of useful structure since we know the counterclockwise ordering of their vertices.
- Given a vertex p of the convex hull of P, you can figure out the clockwise-most vertex of any one mini-hull in only O(log k) time using a binary search.
    - In short: break a mini-hull at some edge so you get a counterclockwise ordering of vertices from one edge endpoint to the other.
    - If you look at the vertices in this order, they first go counterclockwise, relative to p, then clockwise, and then start going counterclockwise again.
    - So you need to do a couple comparisons per iteration, but you can figure out which half of the list contains the max.
- So now we have a faster way to run Jarvis's March! For each vertex p along the hull, you run the binary search on each of the n / k mini-hulls in O(log k) time. You'll find the next vertex in O((n / k) log k) time total.
- If h ≤ k ≤ h^c, then you do h ≤ k iterations in O((n / k) k log k) = O(n log k) = O(n log h) time total.
- Here's code summarizing what we did. To keep things fast, we'll only compute convex hulls with at most k vertices.
- RestrictedHull(P = {p_1, …, p_n}, k):
    - s ← ceil{n / k}
    - Partition P into disjoint subsets P_1, P_2, …, P_s each of size at most k
    - for j ← 1 to s
        - compute CH(P_j) using Graham's scan
    - ell ← leftmost point of P
    - p ← ell
    - add p to CH

```
    i ← 2
    repeat
        for j ← 1 to s
            r_j ← clockwise most point relative to p on CH(P_j)
        r ← clockwise most point of {r_1, …, r_s}
        if r = ell, break
        else, add r to CH
        if i > k, return fail (k is too small)
```

- This whole thing take $O((n / k) k \log k) = O(n \log k)$ time total, but to actually output a convex hull in $O(n \log h)$ time, we need to guess k so that $h \le k \le h^c$.
- To do so, we'll try progressively bigger values of k.
- We can afford to overshoot h by a whole constant power, so we'll try repeatedly squaring k until the algorithm successfully returns a convex hull.

```
FastHull(P):
    k ← 2. CH ← fail
    while CH = fail
        k ← min{k^2, n}
        CH ← RestrictedHull(P, k)
    return CH
```

- We're running RestrictedHull several times. Is the algorithm fast enough?
- Consider the ith guess, $k = 2^{2^i}$. Running RestrictedHull(P, 2^{2^i}) takes $O(n \log (2^{2^i})) = O(n 2^i)$ time.
- The algorithm succeeds as soon as $i = \lceil \lg \lg h \rceil$ where $\lg = \log_2 h$.
- So the total running time (ignoring the ceiling) is proportional to $\sum_{i=1}^{\lg \lg h} n 2^i = n \sum_{i = 1}^{\lg \lg h} 2^i$.
- If a geometric series has a constant ratio, then its sum is proportional to its largest term. The total running time is $O(n 2^{\lg \lg h}) = O(n \log h)$. In other words, the time taken to do that last guess is greater than the time taken to do all previous guesses.
- Can we do better? Kirkpatrick and Seidel ('86) say no. I will not attempt to share their proof with you.