CS 6301.002.20S Lecture 2—January 16, 2020

Main topics are #convex_hulls.

Prereq Forms

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

Finishing Graham Scan

- 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|.
- In this case, 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.
- Well focus on finding the upper hull, the set of points between the rightmost and leftmost when going in counterclockwise order.
- To do this, we'l use *incremental construction*.
- Let p_1, …, p_n be the points in left-to-right order.
- We'll add points to our collection one-by-one, maintaining the upper hull of the points added so far. In other words, after adding point p_i to our collection, we should have an upper hull for points P_i = {p_1, …, p_i}.
- There's a few ways we could implement this, but as suggested by Mount's notes, we'll store the upper hull computed so far in a stack S where S[top] refers to the rightmost point in the upper hull, S[top - 1] refers to the next point to the left and so on.
- There are two points guaranteed to be in the upper hull for P_i. These are p_1 and p_i.
- So, p_i *always* gets added as the rightmost point of the the upper hull. But what other points get to stay?

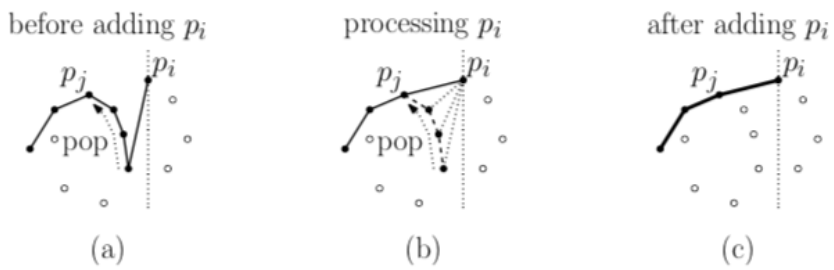


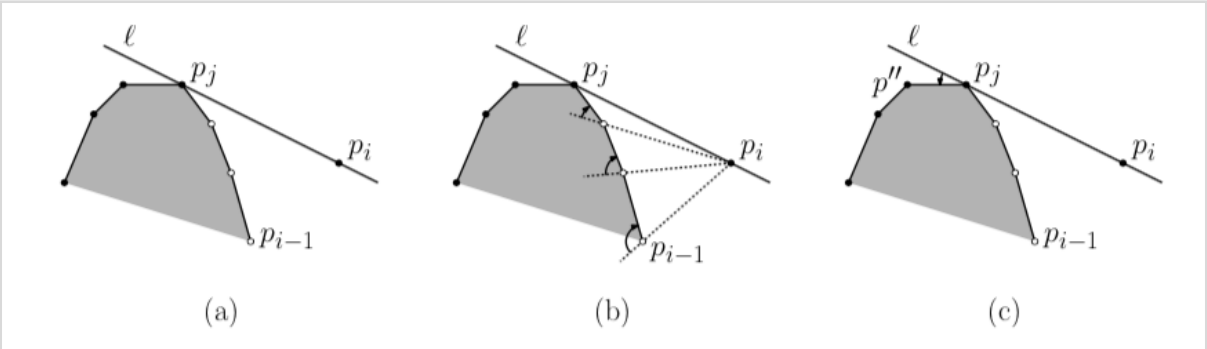
Fig. 14: Graham's scan.

- Consider the ordered triple, <p_i, S[top], S[top - 1]>. I'll prove the following: If we do a right-hand turn following these points, then S[top] has to go! The upper hull goes over

S[top], so we should pop it from the stack. Then we repeat the test, popping, popping, popping, until finally we have only p_1 remaining in the stack or <p_i, S[top], S[top - 1]> forms a left-hand turn.

- Then we can add p_i to the upper hull and we're done with that iteration.
- 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

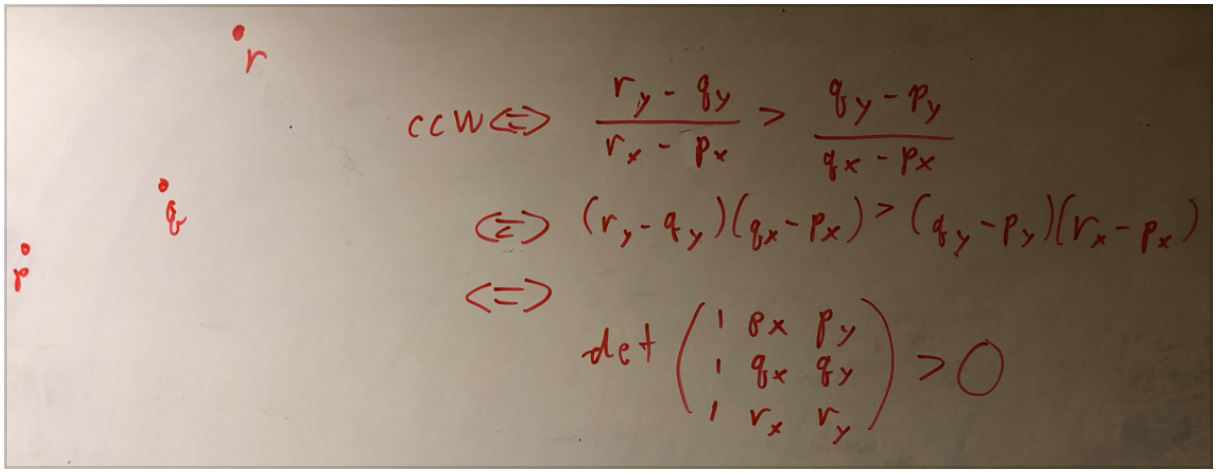
- So we should prove that this actually works. We'll prove the following claim using induction on i:
- Claim: After inserting p_i, the vertices of S from top to bottom form the upper hull of P_i = {p_1, …, p_i} going right-to-left.
- Proof:
  - The claim is clearly true before the for loop, because the upper hull of p_1 and p_2 is the line segment p_1 p_2.
  - Now, consider when we add p_i to our collection to create P_i. p_i must be in the upper hull of P_i since it's furthest point 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 means p_j is on the upper hull of P_{i-1}; otherwise there would be points above p_j. In particular, the induction hypothesis guarantees p_j is in S when we enter the ith iteration of the for loop.
  - Now for each p_k where j < k < i, we have that 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.

Testing Turns and Final Details

- All that remains for designing the algorithm is to find a fast way of determining if a triple makes a right hand turn or left hand turn. It turns out we don't need trig for this!
- Let say we have three points <p, q, r> going from left to right and we want to know if <p, q, r> is a left-hand turn.
- We can determine if they do a left-hand turn by comparing slopes. Let o_x and o_y be the x and y coordinates of point o.



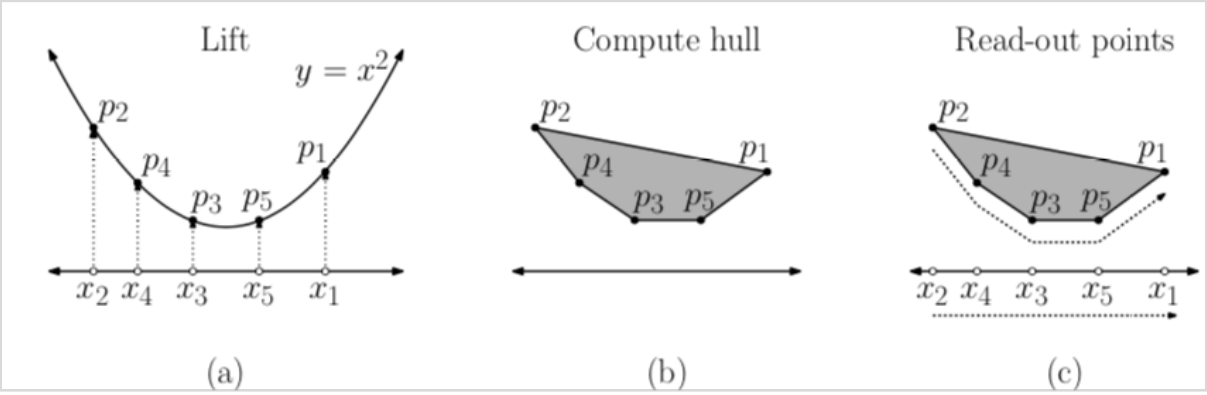
- I'm not going to go through all the other cases, but it turns out this one test using the determinant works no matter how p, q, and r are arranged.
  - If the determine is greater than 0, then left-hand turn.
  - If less than 0, then right-hand turn.
  - If equal to 0, then they're on a line (and not in general position).
- We've designed an algorithm for convex hull! Now for the 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 ≤ n. The total time building the upper hull is O(n) in addition to the time for sorting or 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, … 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):
  - ell ← leftmost point of P
  - p ← ell
  - add p to CH
  - repeat
    - r ← max point ≠ p where q_1 < q_2 iff <p, q_1, q_2> 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 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 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.
- We'll see it on Tuesday.