1 Computational Geometry

There are many important problems in which the relationships we wish to analyze have geometric structure. For example, computational geometry plays an important role in

- Computer-aided design
- Computer vision
- Computer animation
- Molecular modeling
- Geographic information systems, to name just a few.

Finding the Closest Pair of Points

Given a set of points Q, we wish to find the pair of points in Q whose (Euclidean) distance from each other is shortest.

The naive solution is to proceed by brute force, probing all $\binom{n}{2}$ pairs of points and taking $O(n^2)$ time. We will now exhibit a divide-and-conquer algorithm which runs in $(n \log n)$ time.

The algorithm begins by pre-sorting the points in Q according to their x- and y-coordinates:

```
Algorithm: CLOSEST-PAIR($Q$)
1 $X \leftarrow$ the points of $Q$, sorted by $x$-coordinate
2 $Y \leftarrow$ the points of $Q$, sorted by $y$-coordinate
3 return CLOSEST-PAIR-HELPER($X, Y$)
```

Most of the work is done by the helper function CLOSEST-PAIR-HELPER, which makes recursive calls to itself:

```
Algorithm: CLOSEST-PAIR-HELPER($X$,$Y$)
1  if $|X| \le 3$ then
2      Solve the problem by brute force and return
3  $x^* \leftarrow$ the median $x$-coordinate of $X$
4  Let $X_L \subseteq X$ consist of those points with $x$-coordinate $\le x^*$
5  Let $X_R \subseteq X$ consist of those points with $x$-coordinate $> x^*$
6  Let $Y_L \subseteq Y$ consist of those points which are in $X_L$
7  Let $Y_R \subseteq Y$ consist of those points which are in $X_R$
8  ▷ Find the closest two points in the left half
9  $\langle p_L, q_L \rangle \leftarrow$ CLOSEST-PAIR-HELPER($X_L$,$Y_L$)
10 ▷ Find the closest two points in the right half
11 $\langle p_R, q_R \rangle \leftarrow$ CLOSEST-PAIR-HELPER($X_R$,$Y_R$)
12 $\delta_L \leftarrow$ distance from $p_L$ to $q_L$
13 $\delta_R \leftarrow$ distance from $p_R$ to $q_R$
14 $\delta \leftarrow \min\{\delta_L, \delta_R\}$
15 $Y' \leftarrow$ those points in $Y$ whose $x$-coordinate is within $\delta$ of $x^*$
16 ▷ Recall that $Y'$ is already sorted by $y$-coordinate
17 $\epsilon \leftarrow \infty$
18 for $i \leftarrow 1$ to $|Y'| - 1$ do
19     $p \leftarrow Y'[i]$
20     for $q$ in $Y'[i+1, \ldots, \min\{i+7, |Y'|\}]$ do
21         if $\epsilon >$ distance from $p$ to $q$ then
22             $\epsilon \leftarrow$ distance from $p$ to $q$
23             $p^* \leftarrow p$
24             $q^* \leftarrow q$
25 if $\epsilon < \delta$ then
26     return $\langle p^*, q^* \rangle$
27 else if $\delta_R < \delta_L$ then
28     return $\langle p_R, q_R \rangle$
29 else
30     return $\langle p_L, q_L \rangle$
```

Correctness

CLOSEST-PAIR-HELPER begins by recursively calling itself to find the closest pairs of points on the left and right halves. Thus, lines 15-24 are ostensibly an attempt to check whether there exists a pair of points $\langle p^*, q^* \rangle$, with one point on the left half and one point on the right half, whose distance is less than that of any two points lying on the same half. What remains to be proved is that lines 15-24 do actually achieve this objective.

By the time we reach line 15, the variable δ stores the shortest distance between any two points that lie on the same half. It is easy to see that there will be no problems if δ truly is the shortest possible distance. The case we need to worry about is that in which the closest pair of points - call it $\langle p, q \rangle$ - has distance less than δ. In such a case, the x-coordinates of p and q would both have to be within δ of x^*; so it suffices to consider only points within a vertical strip V of width 2δ

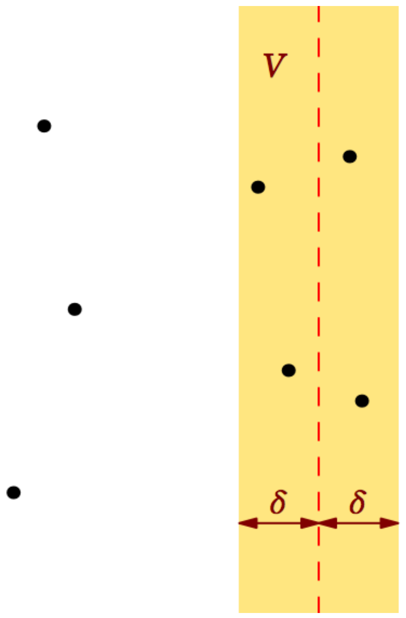


Figure 1: It suffices to consider a vertical strip V of width 2δ centered at $x = x^*$.

centered at $x = x^*$ (see Figure 1). These are precisely the points stored in the array Y' on line 15.

Say $p = Y'[i]$ and $q = Y'[j]$, and assume without loss of generality that $i < j$. In light of lines 20-24 we see that, in order to complete the proof, we need only show that $j - i \leq 7$.

Say $p = (p_x, p_y)$. Let S_L be the square (including boundaries) of side length δ whose right side lies along the vertical line $x = x^*$ and whose bottom side lies along the horizontal line $y = p_y$. Let S_R be the square (excluding the left boundary) of side length δ whose left side lies along the vertical line $x = x^*$ and whose bottom side lies along the horizontal line $y = p_y$. It is evident that q must lie within either S_L or S_R (see Figure 2). Moreover, any two points in the region S_L are separated by a distance of at least δ; the same is true for any two points in the region S_R. Thus, by a geometric argument, S_L and S_R each contain at most four points of Y'. In total, then, $S_L \cup S_R$ contains at most eight points of Y'. Two of these at-most-eight points are p and q. Moreover, since Y' consists of all points in V sorted by y-coordinate, it follows that the at-most-eight points of $S_L \cup S_R$ occur consecutively in Y'. Thus, $j - i \leq 7$, and the algorithm is correct.

Running Time

Within the procedure CLOSEST−PAIR−HELPER, everything except the recursive calls runs in linear time. Thus the running time of CLOSEST−PAIR−HELPER satisfies the recurrence

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$$

which has solution

$$T(n) = O(n \log n)$$

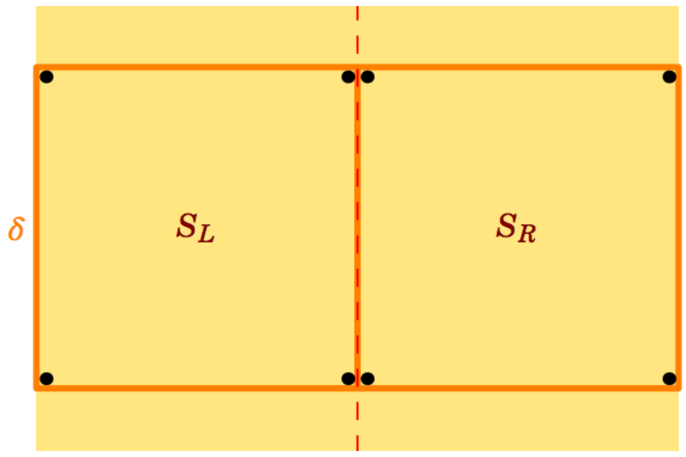


Figure 2: Each of S_L and S_R can hold at most 4 points.

Note that, if we had decided to sort within each recursive call to CLOSEST-PAIR-HELPER, the $O(n)$ term in the recurrence would have instead been an $O(n \log n)$ term and the solution would have been

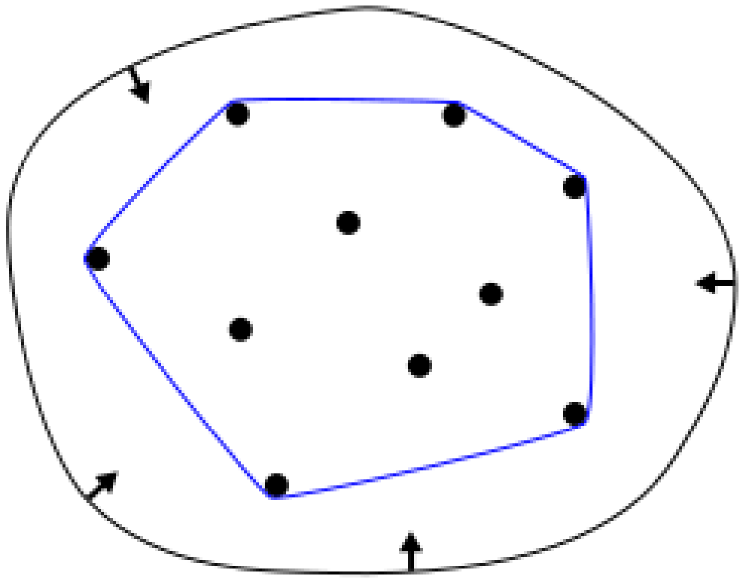
$$T(n) = O(n(\log n)^2)$$

This is the reason for creating a helper procedure to handle the recursive calls: it is important that the lists X and Y be pre-sorted so that recursive calls need only linear-time operations. Note also that, if instead of lines 20-24 we had simply checked the distance between each pair of points in Y', the $O(n)$ term in the recurrence would have instead been an $O(n^2)$ term, and the solution would have been

$$T(n) = O(n^2)$$

Convex Hull

Given a set of points Q, we may want to find the convex hull, which is a subset of points that form the smallest convex polygon where every point in Q is either on the boundary of the polygon or in the interior of the polygon. We can imagine fitting an elastic band around all of the points. When the band tightens, the points that it rests on form the convex hull.



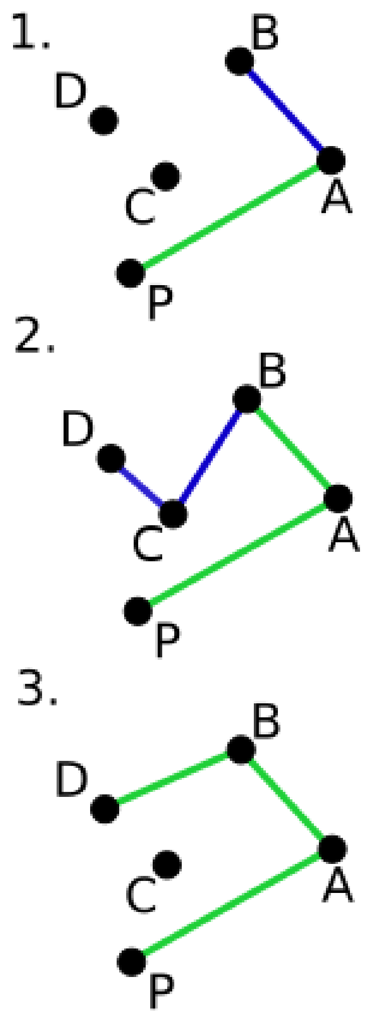
There are several algorithms to solve the convex hull problem with varying runtimes.

Graham's Scan

The Graham's scan algorithm begins by choosing a point that is definitely on the convex hull and then iteratively adding points to the convex hull.

1. Let H be the list of points on the convex hull, initialized to be empty
2. Choose p_0 to be the point with the lowest y-coordinate. Add p_0 to H since p_0 is definitely in the convex hull.
3. Let $(p_1, p_2, ..., p_n)$ be the remaining points sorted by their polar angles relative to p_0 from smallest to largest. Iterating through the points in sorted order should sweep around p_0 in counterclockwise order
4. For each point p_i:
   (a) If adding p_i to our convex hull results in making a "left turn", add p_i to H
   (b) If adding p_i to our convex hull results in making a "right turn", remove elements from H until adding p_i makes a left turn, then add p_i to H.

Below is an example of left and right turns. At step 1, our convex hull H is (P, A, B). Adding C resulted in making a left turn at B if we're going from A to C, so we add C to H, which is now (P, A, B, C). Adding D in step 2 resulted in making a right turn at C if we're going from B to D. If we just added D now, we would not have a convex shape. To fix this, we remove elements from H until adding D results in making a left turn. Simply removing C does the trick and we add D after C is removed from H. This results in the convex hull solution of $H = (P, A, B, D)$.



Note that there may be cases where we have to remove multiple points from H in order to fix the convexity. Once we iterate through every point, H will form the convex hull.

The bottleneck of the algorithm is sorting the points by polar angles. This operation requires $O(n \log n)$ time. Since every point is added to H exactly once and every point is removed from H at most once, iterating through the points and forming H after the sorting takes $O(n)$ time. Thus, the whole algorithm takes $O(n \log n)$ time.

Jarvis' march

The Jarvis' march algorithm conceptually is very similar to Graham's scan. Again, we first choose p_0 in the same fashion as before, by scanning through the points to find the point with minimal y-coordinate. This time, we find the next point in the convex hull by iterating through all n other points and discovering the next point with the smallest relative polar angle to the last point added to the convex hull. The difference here is that each point we add is definitely on the convex hull, as opposed to Graham's scan where each point we add may need to be removed later. The tradeoff is that after each addition to the convex hull, we need to iterate through every other point to find the next point on the convex hull, which takes $O(n)$ time. The algorithm terminates once we try to add p_0 to H again.

The runtime of this algorithm is output-sensitive, meaning that the runtime depends on how many points are on the convex hull solution. For each point on the convex hull, we need to spend $O(n)$ time to iterate through all of the other points. Thus, if there are $O(h)$ points on the convex hull, the runtime will be $O(nh)$. If $h = o(\lg n)$, i.e. the convex hull is formed by a small portion of the total points, then Jarvis' march outperforms Graham's scan.