Computational Geometry - Divide and Conquer Closest Pair

In computational geometry, two well-known problems are to find the closest pair of points and the convex hull of a set of points.

The closest-pair problem, in 2D space, is to find the closest pair of points given a set of n points. Given a list P of n points, $P_1=(x_1,y_1)$, … $P_n=(x_n,y_n)$ we simply do the following:

```
BruteForceClosest(P)
        min ← ∞
        for i = 1 to n-1
                for j = i+1 to n do
                        d ← distance(Pi,Pj)    // Use sqrt(distances squared)
                        if d < min then
                                min ← d
                                minPoints = (Pi,Pj)
```

The basic operation is computing the Euclidean distance between all pairs of points and requires $O(n^2)$ runtime. We could arrive at this value more formally by noting:

$$T(n) = \sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} C\right) = C\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1\right) = C\sum_{i=1}^{n-1}(n-i) = C(n-1)(n-i) = \theta(n^2)$$

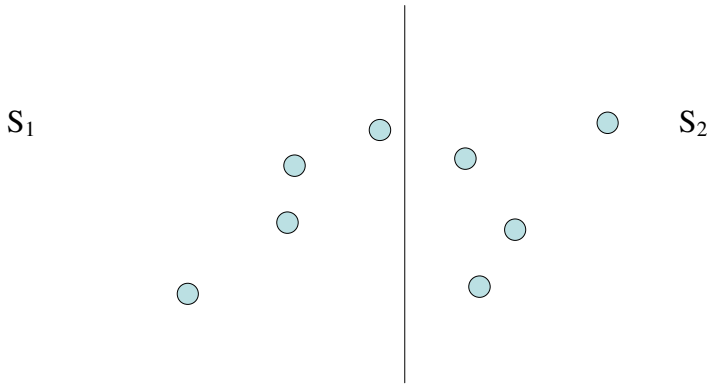
This requires computing the square root of the sum of squares of the difference between the coordinates in the point. For a large number of points, computing the square root is a very expensive operation and can take a long time to run.

In fact, we don't even need to compute the square root – we can simply ignore the square root and compare the values $(x_i - x_j)^2 + (y_i - y_j)^2$ themselves, since this value is strictly increasing compared to the square root of the value. This results in the same runtime, but would significantly increase the execution speed.

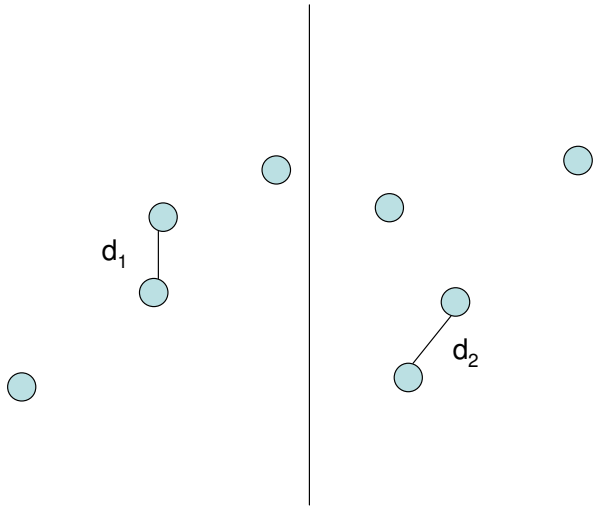
However, we can do better!

Let $P_1 = (x_1, y_1)$, … $P_n = (x_n, y_n)$ be a set S of n points in the plane, where n, for simplicity is a power of two. We can sort these points in ascending order of their x coordinate using a O(nlgn) sorting algorithm such as mergesort.

Next we divide the points into two subsets S_1 and S_2 where each have n/2 points by drawing a line through the median of all points. We can find the median in constant time once the points are sorted. This is shown below:

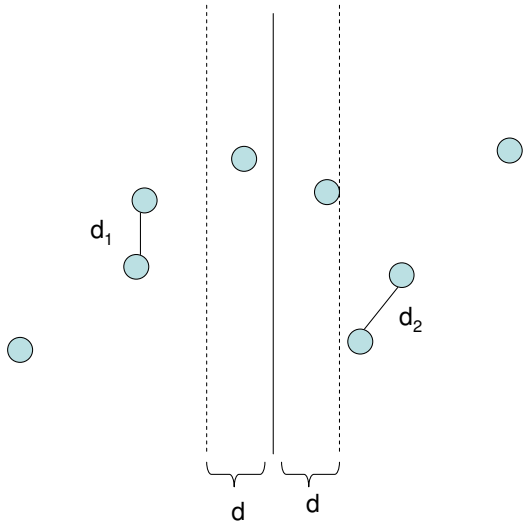


Next, we recursively find the closest pairs for the left subset S_1 and for the right subset S_2. If the set consists of just two points, then there is only one solution (that connects those two points). Let d_1 and d_2 be the smallest distances between pairs of points in S_1 and S_2 as shown below:

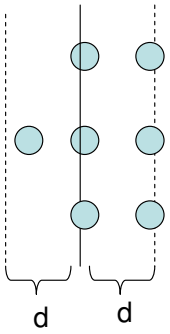


Let d = the minimum of (d_1 and d_2). This is not necessarily our answer, because the shortest distance between points might connect a pair of points on opposite sides of the line. Consequently, we must compare d to the pairs that cross the line.

We can limit our attention to points in the symmetrical vertical strip of width 2d since the distance between any other pair of points is greater than d:



For every point between the left dashed line and the symmetrical line in the middle we must inspect the distance to points on the right side of the line to the right dashed line. However, we only need to look at those points that are within a distance of d from the current point. The key here is that there can be no more than six such points because any pair of points in the right half is at least d apart from each other. The worst case is shown below:



If we maintain an additional list of the points sorted by y coordinate then we can limit the points we examine to +/-d in both the x and y direction for these border points.

To perform the step of comparing distances between points on both sides of the line requires O(n) runtime. For each of up to n points we have a constant number (up to 6) of other points to examine. This process is similar to the "merge" time required for MergeSort. We then compare the smallest of these distances to d, and choose the smallest distance as the solution to the problem.

Our total runtime is then:

$T(n) = 2T(n/2) + O(n)$ ' Time to split in half plus line comparison

We can solve this using a variety of methods as O(nlgn). This is the best we can do as an efficiency class, since it has been proven that this problem is Ω(nlgn).

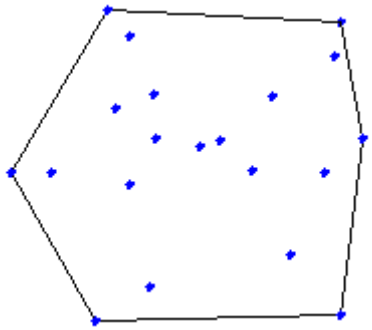
Convex Hull Problem

In this problem, we want to compute the convex hull of a set of points. What does this mean?

- Formally: It is the smallest convex set containing the points. A convex set is one in which if we connect any two points in the set, the line segment connecting these points must also be in the set.
- Informally: It is a rubber band wrapped around the "outside" points.

Here is a picture from:
http://www.cs.princeton.edu/~ah/alg_anim/version1/ConvexHull.html

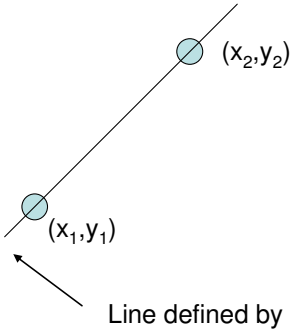
It is an applet so you can play with it to see what a convex hull is if you like.



Theorem: The convex hull of any set S of n>2 points (not all collinear) is a convex polygon with the vertices at some of the points of S.

How could you write a brute-force algorithm to find the convex hull?

In addition to the theorem, also note that a line segment connecting two points P_1 and P_2 is a part of the convex hull's boundary if and only if all the other points in the set lie on the same side of the line drawn through these points. With a little geometry:



For all points above the line, $ax + by > c$, while for all points below the line, $ax + by < c$. Using these formulas, we can determine if two points are on the boundary to the convex hull.
High level pseudocode for the algorithm then becomes:

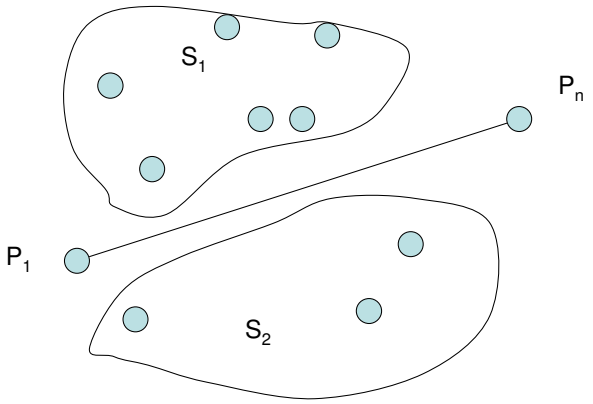
```
for each point Pi
        for each point Pj where Pj ≠ Pi
                Compute the line segment for Pi and Pj
                for every other point Pk where Pk ≠ Pi and Pk ≠ Pj
                        If each Pk is on one side of the line segment, label Pi and Pj
                        in the convex hull
```

What is the runtime for this algorithm?

Let's look at an expected O(nlgn) algorithm called QuickHull that is somewhat similar to Quicksort.

Once again, we have a set of points P located in a 2D plane. First, sort the points in increasing order of their x coordinate. This can be done in O(nlgn) time.

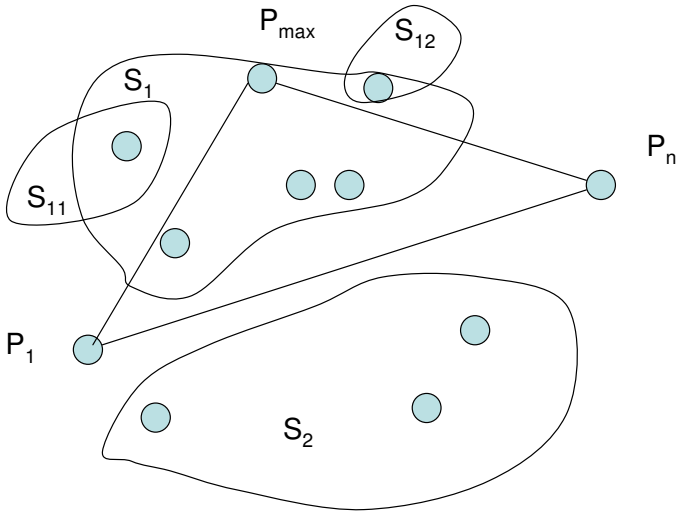
It should be obvious that the leftmost point P_1 and the rightmost point P_n must belong to the set's convex hull. Let $\mathbf{P_1P_n}$ be the line drawn from P_1 to P_n. This line separates P into two sets, S_1 to the left of the line, and S_2 to the right of the line. (left is counter clockwise when connecting the points). S_1 constitutes the boundary of the upper convex hull and S_2 the boundary of the lower convex hull:



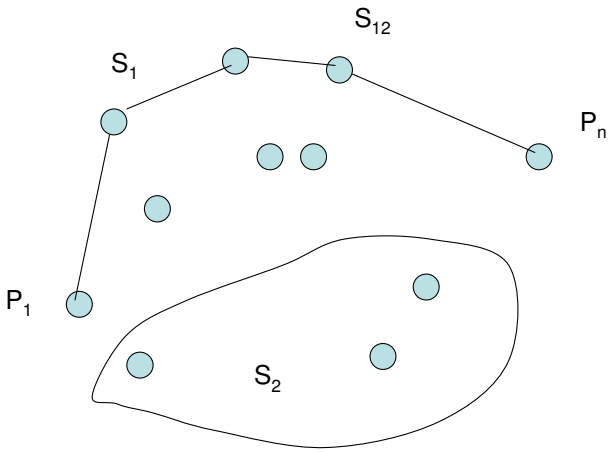
If we're lucky, the line exactly separates the points in half, so half are in S_1 and half are in S_2. In the worst case, all other points are in S_1 or S_2. If there is some randomness the on general we can expect to cut the problem close to in half as we repeat the process.

Next we'll find the convex hull for S_1. We can repeat the same exact process to solve S_2. To find the convex hull for S_1, we find the point in S_1 that is farthest from the line segment $\mathbf{P_1P_n}$. Let's say this point is P_{max}. If we examine the line segment from P_1 to P_{max} then we can recursively repeat the algorithm for all points to the left of this line (The set S_{11} in the diagram below). Similarly, if we examine the line segment from P_n to P_{max} then we can recursively repeat the algorithm for all points to the right of this line (The set S_{12} in the diagram below).

All points inside the triangle can be discarded, since they are in the triangle and can't be part of the convex hull.



If we try to find the points in the convex hull for a set with only one point, then that point must be in the set. At this point we have determined the upper convex hull:



If we repeat the process on the lower convex hull we will complete the problem and find all of the points for the entire convex hull.

For an animated applet that illustrates this algorithm, see:
http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html

What is the runtime? Let's say that we always split the set of points in half each time we draw the line segment from P_1 to P_n. This means we split the problem up into two subproblems of size n/2. Finding the most distant point from the line segment takes O(n) time. This leads to our familiar recurrence relation of:

$T(n) = 2T(n/2) + O(n)$ which is O(nlgn).