CS 218 : Design and Analysis of Algorithms

Lecture 9: Finding a Pair of Closest Points

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering Indian Institute of Technology, Bombay

1 Finding Closest Points

1.1 Problem Statement

Our next problem is from computational geometry. **Input**: A set of n points specified by their x and y co-ordinates. **Output**: The closest pair of points.

The distance between two points is the usual: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

A naive algorithm takes $O(n^2)$ time. We, as usual, like to do better. Why do we even suspect that this is true?

Design Tip for Geometric Problems. First work with the same problem in smaller dimensions. Here then the problem boils down to finding the closest pair of points on a line. Say the x-axis. Most students at this point say sort and check distances between adjacent points. This does work. Time is $O(n \log n)$. So, there is hope.

The obvious problem in 2D is that there is no notion of sorting. Also, in 1D for each point the number of distances that one needs to compute for this point is at most two. In 2D this can clearly be $O(n)$. Draw a figure to convince yourself of this.

The algorithm we described above was construted in an ad-hoc manner. It will be useful to construct it using the principles that we have discussed so far. There are two types of induction that have typically worked for us. Remove one point, recurse on the rest and construct the solution with this point put in. Or divide the input into two (roughly) equal parts, recurse on each and put Humty-Dumpty together. We are still working in 1D. So, we can remove one element, and recursively find the closest pair among the rest. Now we notice that if we soup up the induction and return the sorted order we can place the new point ($O(\log n)$ time) and then finish in $O(1)$ time. This of course leads to insertion sort-a familiar problem with familiar solutions. One is to use a balanced binary search tree and the other is a merge sort type recursion. Now that we have these two ways of doing the recursion, can we extend this to 2D? What are the problems we face? The key problem remains the same: no notion of order amongst the points.

When we started the course we mentioned three principles: induction, order and store old values. The first we are doing. The last we are. This leaves us with the second. We have not really explored orders in which we consider the input. So far, it has either not mattered or has happened naturally but in this case we need to look at it more carefully. By order, we mean, we do not arbitrarily split the input but use some order to split. What are natural orders that one can try here? Two possible orders are say distance of the points from the origin or order by one co-ordinate.

With some foresight we explore the latter. We then have two choices. Recurse on the first $n-1$ points (soted on the x-coordinate) and then patch the result of putting in the last point. Or split the points into two parts: the first containing the ones with the smallest $n/2$ x-coordinates. Both can be extended to give us an algorithm. We begin with the second.

Warning: We will make an initial guess for the algorithm and keep modifying it till we achieve what we want. Our goal is an algorithm that runs in time $O(nlogn)$ time.

In the procedure we describe, we will only say how to find the distance of the closest pair. The actual pair of points can be easily found by doing some book-keeping along with the algorithm we design and we leave this as an exercise.

Split the points into two equal parts based on their x co-ordinates. Those to the left of the median and those to the right. Note that we can do this in $O(n)$ time. It is all right if we do this in $O(n \log n)$ time too, at this stage. We will fix this later.

Recursively find the closest pair in the two halves.

We now need to put the solutions together. Within points in each group we know the answer. But we still have to somehow find the closest pair among points where the two points are in different groups. If we compare every pair again, we are back to $\Omega(n^2)$ comparisons. We first organise the information we have at this point and see if we can extract any mileage from it. This is to do with problem dependent properties.

Let the distance of the closest pair, among points in the left half be d_L an among points in the right half d_R. Let $d = \min(d_L, d_R)$.

Recall that we need to find the closest distance between pairs of points such that one of them is among the left half and the other among the right half. Can we say something more about where these points will lie? If we draw a vertical line l through the median of the x-coordinates how far from l will these points lie?

It is easy to see that we only need to consider points whose x-coordinate differs from the median x-coordinate by at most d.

We still do not seem to have made much progress. In the worst case all points may lie in this band of width $2d$ around the line l and we may have to compute $\Omega(n^2)$ distances.

To beat this $\Omega(n^2)$ bound we need more ideas. The next one is crucial. *Observe how these are discovered one by one.* Take any point P in the left band. How do we find the point which is closest to P on the right? Where do we look, for points which are at distance at most d from P?

It sufffices to consider points to the right of l whose y-coordinate differs from that of P by at most d. Question: Why not look at a circle of radius r around P?

The reason why the square is more attractive than the circle is that if we now have to focus on only the y-co-ordinates. We have a problem in one-dimension which is easier to solve. In other words, it is more difficult to find points in the circle as opposed to the square though the circle captures all the information we know.

Exercise. Clearly state the problem we have to solve now.

The crucial intuition that drives the algorithm is that there cannot be too many points in a square of side d, such that the distance between any two of them is at least d. This will limit the number of candidate points and hence distances we will need to check for P. Picture this. You have a square of side length d. If you start putting too many points you just cannot avoid putting points close to each other. We will make a precise statement. Part of algorithm design and analysis is to come up with such statements and proofs. The intuition comes first, but is equally important to translate this intuition into a formal proof. **Claim.** If a square of side length d contains 5 points then at least two of them are at distance at most $d/\sqrt{2}$ from each other.

Try proving this before reading ahead. The hint is to use the pigeon hole principle. To use the pigeon hole principle, you need to judiciously select your pigeons and holes. The points are your pigeons. For holes you divide the square into four. How? If you place 5 points in this big square, at least two of them will fall in one of the smaller squares. Finish this proof.

So for a point P on the left hand side there are at most 8 points that we need to consider. If we are able to identify these 8 points quickly, then we need to compute at most $O(n)$ distances. By now the overall structure of the algorithm should be clear in your minds. Here is a way to

do this in $O(logn)$ time per point on the right. Which means a total of $O(nlogn)$ time. This is a first attempt. We will subsequently do it faster. Sort the points to the right of l which lie in the "band", on their y-coordinates.

Now use binary search to place points from the left hand side, P, and for each point in the left compute distances with at most 5 points on either side. Pick the minimum among all the distances computed.

Exercise: Write down the entire algorithm. What is the total time taken by this algorithm?

$T(n)$ satisfies, $T(n) \leq 2T(n/2) + O(nlogn)$ giving a bound of $O(nlog^2n)$. Make sure you can solve this recurrence. How do we reduce the time in the last step from $O(nlogn)$ to $O(n)$? This should now look familiar. We encountered a very similar problem while dealing with inversion pairs. Suppose L and R are the points in the left and right of the line l, both sorted on their y coordinates. We can extract those that lie in the band in $O(n)$ time. We can now find all relevant distances in $O(n)$ time as follows. Define 3 pointers p, q and r. We assume that the points on the left band L and the points in the right band R are sorted by their y-coordinates.

The pointer p points to a point P in L, q and r will point to the bottom-most (and top-most) points in R whose y-coordinates differ from P by at most d. Once we are done with P, we update p by 1 to get the next point say P'. We then increment q as long as the y-coordinate is not to within d of P'. Similarly increment r.

Show that the time taken is $O(n)$. Apart from distance computation, the only other step is pointer movement and since each pointer only moves forward this is bounded by $O(n)$. As we have done before, we can sort on y-coordinates as part of the recursion. This means that during the recursion we only merge the arrays sorted on y-coordinates. Which brings down the time to $O(n)$ during the recursion.

Can you write the complete algorithm?