Exercises 16: closest pair of points

Questions

1. In the closest pair of points algorithm, one of the subproblems is to find the closest pair of points with one point in the left half and one point in the right half. If we use brute force for that problem, we get a recurrence $t(n) = 2\, t(n/2) + n^2 /4 + c$. Solve this recurrence by backsubstitution. (Note that this is a particular case of the Master method which we will see next lecture.)

2. The subtle part of the closest pair problem is how to efficiently deal with pairs for which one point is in the left half and the other point is in the right half. In the part of the algorithm dealing these pairs, we did not explicitly test that one point was in the left half and one point was in the right half, however. For each point p, we merely checked the next 7 points in order of increasing y value. Was this a mistake in the slides? Could it happen that one of the next 7 points might lie in the same half as p ? Would this be a problem?

3. Does the closest pair algorithm assume that the x coordinates (and y coordinates) of the points are distinct? Is there a problem with the O(n log n) performance if they are not distinct?

4. How would you solve the closest pair of points problem in 3D?

Answers

1. Assume n is a power of 2 for simplicity.

$t(n) = 2\, t(n/2) + n^2 /4 + c$

$= 2 \{ 2\, t(n/4) + (n/2)^2 / 4 + c\} + n^2 /4 + c$

$= 4\, t(n/4) + 2 (n/2)^2 / 4 + n^2 /4 + 2 c + c$

$= 4 \{2\, t(n/8) + (n/4)^2/4 + c\} + 2 (n/2)^2 / 4 + n^2 /4 + 2 c + c$

$= 8\, t(n/8) + n^2/16 + n^2/8 + n^2 /4 + 4 c + 2 c + c$

$= 8 \{2\, t(n/16) + (n/8)^2/4 + c\} + (n^2/4 + n^2/2 + n^2) /4 + (4 c + 2 c + c)$

$=$ etc

$= 2^k\, t(n / 2^k) + n^2 /4 * (\Sigma_{i = 0 \text{ to } k-1}\, 2^{-i}) + (\Sigma_{i = 0 \text{ to } k-1}\, 2^i) * c$

$<= n\, t(1) + n^2/2 + c\, 2^{\log n}$, since $\Sigma_{i = 0 \text{ to } k-1}\, 2^{-i}) < 2$ for any finite k

$= n^2/2 + (c + 1) n$, when $t(1) = 1$.

Notice that the n^2 which appears in the first line ends up being the dominant term. That is, the smaller subproblems still have their own "n^2" term, but the sum of these is determined by Σ_{i = 0 to k-1} 2^{-i} which is bounded above by a constant so ends up not mattering in the O() analysis.

2. No, it wasn't a mistake. It could happen that some of the 7 points are in the same half as p. But in this case, the distance from such a point to p cannot be less than δ because δ was defined to be the smallest distance between points that are both in the left half or both in the right half. Thus, although the algorithm considers these points, it never uses them to find a lower δ. (Note that the algorithm could have instead detected directly that the points lie on the same side, rather than checking their distance.)

3. Take the case that two points have the same x coordinate. The only conceivable issue I can think of is that it might be impossible to find a vertical line such that half the points are to the left of the line and half are to the right, namely if the point at position n/2 in the X ordering has the same value as the point at position n/2+1, assuming indices are from 1 to n. (The extreme case is that all the points have the same x value.) But its easy to solve this problem. Just take the vertical line to be through those two points of equal x value. So the set X_L will be defined as those points whose x values are less than or equal to the x value of the vertical line and X_R will be those points whose x values are greater than or equal to it, and for those points whose x values are equal to the x value of the vertical line, we decide on membership in X_L or X_R based on whether the index is less than or equal to n/2 or not.

   As long as we are dividing the points into (roughly) equal parts, the recurrence given in class holds.

   What about problems with two points have equal y value? First, if two points have the same x and y value, then the distance would be 0. Let's not worry about this as a problem. The algorithm will simply find (one of) these two points, if they exist. Second, If two points with the same y value have different x values, then there is still no issue. The same "at most 7 next points" condition still holds.

4. The same idea can be used as for the 2D problem. Sort the points by x, y, and z, giving three arrays X, Y, Z. Partition the points in X into two sets X_L and X_R based on their sorted x values. Recursively find the closest pair in each of X_L and X_R, and let δ be the closest distance of pairs either both in X_L or both in X_R. To deal with the pairs of points where one is in X_L and one is in X_R, we need a condition similar to what he had for the closest distance in 2D problem. In the 3D problem, we don't have a middle strip of width 2δ but rather we have a middle slab of width 2δ where the slab extends in directions y and z. But the same idea can be used as before. For each point in the slab, there can be at most some bounded number of points in its neighborhood, since those points in the slab that are in X_L (or X_R) must be spaced at least δ

apart. Its not obvious what the bound is, but let’s say it is B points. Then finding the closest pair with one point in X_L and the other in X_R can be done in O(n) time.
Notes:

- You need access to the sorted Y and Z array to do it the O(n) since you need to identify the B points that are nearest in Y and Z.
- The above algorithm gives the same recurrence for time complexity, and hence the 3D problem can be solved in O(n log n) time too. Wow!
- I am not specifying details of how these B points are defined. Hopefully you agree with me that these details are not the main point here.
- The extension to the k-dimensional problem is possible and again leads to the same recurrence (although the constant B will grow with k – not clear how). BTW, in class I was asked about this k dimensional problem. The answer I gave was wrong.