Solution of HW2

---

1. Let $A = 2^{\frac{2n}{3}} A_1 + 2^{\frac{n}{3}} A_2 + A_3$ and $B = 2^{\frac{2n}{3}} B_1 + 2^{\frac{n}{3}} B_2 + B_3$. Then we have:

$$A \times B = \left(2^{\frac{2n}{3}} A_1 + 2^{\frac{n}{3}} A_2 + A_3\right)\left(2^{\frac{2n}{3}} B_1 + 2^{\frac{n}{3}} B_2 + B_3\right),$$

which can be expanded as:

$$A_1B_1 2^{\frac{4n}{3}} + (A_1B_2 + A_2B_1)\, 2^n + A_2B_2 2^{\frac{2n}{3}} + (A_2B_3 + A_3B_2)\, 2^{\frac{n}{3}} + (A_1B_3 + A_3B_1)\, 2^{\frac{n}{3}} + A_3B_3.$$

Similar to the example introduced in Lecture 5, we have the following observations:

$$\begin{aligned} A_1B_2 + A_2B_1 &= (A_1 + A_2)(B_1 + B_2) - (A_1B_1 + A_2B_2) \\ A_2B_3 + A_3B_2 &= (A_2 + A_3)(B_2 + B_3) - (A_2B_2 + A_3B_3) \\ A_1B_3 + A_3B_1 &= (A_1 + A_3)(B_1 + B_3) - (A_1B_1 + A_3B_3). \end{aligned}$$

Therefore, we only need 6 products: $(A_1 + A_2)(B_1 + B_2), (A_2 + A_3)(B_2 + B_3), (A_1 + A_3)(B_1 + B_3), A_1B_1, A_2B_2, A_3B_3$, which yield:

$$T(n) = 6T\left(\frac{n}{3}\right) + O(n).$$

Thus, we obtain a running time of $O(n^{\log_3 6}) = O(n^{1.63})$.

2. A brute-force algorithm simply looks at all the $\binom{n}{2} = \Theta(n^2)$ pairs of points. We shall use a more efficient divide-and-conquer algorithm as follows.

Denote the given set of n points by Q. For each recursive invocation, it accepts a subset $P \subset Q$ and arrays X and Y, each of which contains all the points of P. The points in array X are sorted to be monotonically increasing w.r.t. their x-coordinates. Similarly, array Y is monotonically increasing w.r.t. y-coordinate.

The divide-and-conquer algorithm is as follows:

Divide: Find a vertical line l that bisects the point set P into left set P_L and right set P_R, such that $|P_L| = \lceil |P|/2 \rceil, |P_R| = \lfloor |P|/2 \rfloor$. Divide array X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide array Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.

Conquer: With P_L and P_R, now make two recursive calls: one to find the nearest pair of points in P_L and the other to find the nearest pair of points in P_R. The inputs to the first call are P_L, X_L and Y_L; the second call receives the inputs P_R, X_R, and Y_R. Let the nearest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.

Combine: The nearest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair with one point in P_L and the other in P_R. Now we check if the latter case is true. If such pair exists, then its both points must be within δ units of line l. As Figure (a) shows, they must reside in the 2δ-wide vertical strip centered at line l. To check such a pair, we do the following:

(a) Create an array Y', which is the array Y with all points not in the 2δ-wide vertical strip removed. The array Y' is sorted by y-coordinate.

(b) For each point p in the array Y', try to find points in Y' that are within δ units of p. As we shall prove, only the 7 points in Y' that follow p need to be considered. Compute the distance from p to each of these 7 points. Finally, return the nearest-pair distance δ' found in Y'.

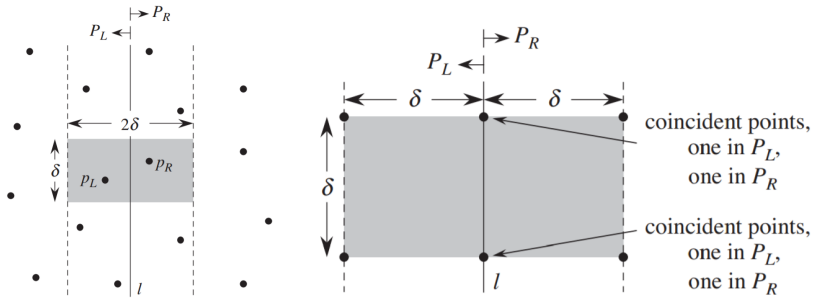
(c) If $\delta' < \delta$, then the vertical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ found by the recursive calls.

Since we need to sort X, Y and Y', the running time would be $T(n) = 2T(n/2) + O(n \log n)$, yielding $T(n) = O(n \log^2 n)$.

Improvement ("Presort" the arrays):

We can improve the above algorithm by "presorting" the input arrays once and maintaining the sorted property without any sorting in each recursive call:

We sort arrays X and Y once before running the algorithm. Then having partitioned P into P_L and P_R, we can form the sorted arrays Y_L and Y_R in linear time, which can be viewed as the opposite of the MERGE procedure from merge sort. Specifically, we simply examine the points in array Y in order. If a point $Y[i]$ is in P_L, we append it to the end of array Y_L; otherwise, we append it to the end of array Y_R. A similar procedure works for forming arrays X_L, X_R, and Y'. In this way, the generated subarrays Y_L, Y_R, X_L, X_R and Y' are also sorted. The total cost of this improved algorithm is $T'(n) = T(n) + O(n \log n)$, where $T(n) = 2T(n/2) + O(n)$ (which gives $T(n) = O(n \log n)$). So the total complexity is $O(n \log n)$.



Proof of the 7 points trick:

Suppose the nearest pair of points is $p_L \in P_L$ and $p_R \in P_R$. Then the distance δ' is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away, which is similar for p_R. Moreover, p_L and p_R are within δ units of each other vertically. Thus, as Figure (b) shows, p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at line l.

We next show that at most 8 points of P can reside within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square forming the left half of this rectangle. Since all points within P_L are at least δ units apart, at most 4 points can reside within this square; Figure (b) shows how. Similarly, at most 4 points in P_R can reside within the $\delta \times \delta$

square forming the right half of the rectangle. Thus, at most 8 points of P can reside within the $\delta \times 2\delta$ rectangle.

Now let us assume without loss of generality that p_L precedes p_R in array Y'. Then, even if p_L occurs as early as possible in Y' and p_R occurs as late as possible, p_R is in one of the 7 positions following p_L, due to the above 8 points conclusion. □