CSCI270 Week6

Jacob Ma

October 3, 2022

Problem 1: Closest Pair of Points (in 2-D)

Given points $p(1) = (x(1), y(1)), p(2) = (x(2), y(2)), \cdots, p(n) = (x(n), y(n))$. Find the closet pair of points.

We use Euclidean Distance:

$$d(p(i), p(j)) = \sqrt{(x(i) - x(j))^2 + (y(i) - y(j)^2)}$$

Brute Force (try all pairs takes $O(n^2)$).

For simplicity, assume all x coordinates and all y coordinates are different.

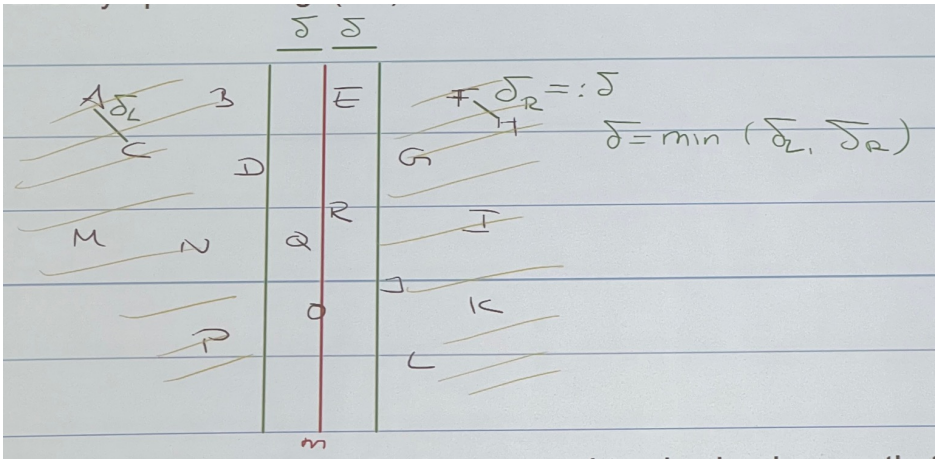
To do better, we try Divide & Conquer.

(1) If n small enough, then solve directly.

(2) Otherwise, partition the points into left and right half.

(3) Recursively find the closet pair on the left and closet pair on the right.

(4) Then do something to finish the result.



To reduce the number of pairs we need to check, observe that if one of the points is very far from the dividing line, then it cannot be part of the closet pair.

0.1 Closest pair of points

Proof. Overall correctness proof: induction on array size n.

Base case: $n = 1$ or 2, which is trivial

IS: By IH, the recursive calls on the left and the right return the closest pairs in those sets. So delta is the correct value. We can safely remove others because any pair that might be closer must be within at most delta of the mid line. Since the code finds the closest pair among points within delta of the mid line, it finds the closest pair overall: either the one from the recursive calls or the new pair if it is better. □

Proposition 0.1: Running Time Analysis

Assume we get all the sorting by x and y coordinates for free.
Find the median (dividing line) and partition points into left and right: $O(n)$ or $O(1)$ is we just pass indices.
Find the points in the middle strip: $O(n)$ with one scan through the array sorted by n coordinate.
Running the comparison function on the left: n iterations over points i and for each, we need to look at most 12 indices ahead by the square area argument - $O(n)$
Recurrence:

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

Solution:

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

Note: To avoid having to sort by x and/or y coordinate in each recursive call (which would cause the time to be $O(n \log n)$), we can pre-sort the entire array once by x coordinate and once by y coordinate. So when we know which subset we deal with, we go through the sorted array, and copy the desired subset in sorted order into another array that is used in the sub-call.

1 Dynamic Programming

One way to think about it: avoid duplicating work in backtracking solution if possible. Closely related to recursive thinking to break bigger problems into smaller pieces, but contrast to greedy.
Warm up: Fibonacci Numbers. Straight forward algorithm gives $T(n) = T(n-1)+T(n-2)+O(1)$. So $T(n) \geqslant f(n) =$ Golden-ratio^n. However, this calculated $f(n-2)$ twice (in both $f(n-1)$ and $f(n-2)$). We should instead store the result once we computed it. So we can even get rid of recursion.

1.1 Weighted Interval Selection

The problem: n intervals with start times $s(i)$ finish time $f(i)$, and weights $w(i)$. Goal: Select a set S of intervals in which no two intervals overlap, and so that the total weight of selected interval is maximized.
Application: weights could be importance/interest of events between which we need to choose.
Key insight: If we have intervals $1, 2, ..., n$ available, the optimal solution for those intervals does one of the following with respect to interval i :

(1) Not include i , and get the optimum solution for all intervals except i.

(2) Include i, and get the optimum solution for all intervals except those intersecting i, plus the weight that it gets from i itself.

Among these, optimum solution chooses the better one.

Note: once we commit to including i , or to not including i, we compute/use the optimum solution for the resulting sub-problem.