Problem Set 7 Solutions

Problem 7-1. Finding a Close Pair

The solution to this problem requires only a simple modification of the close pair algorithm presented in lecture. Recall briefly the algorithm from lecture: we hash each point p into the rectangular cell c containing p. For each cell c, the algorithm thus obtains the set of all of points in c, which we denoted B_c. The construction of B_c for every non-empy cell c requires in total $O(n)$ time, and we then noted that if $|B_c| \geq 256/\pi$, for any cell c, then c must contain some pair of points which are "close". Otherwise, if $|B_c| < 256/\pi$ for every cell c, then one may simply compare, for every cell c, all pairs of points within c, to determine if any of these are close. Since there is at most a constant number of points in each cell, this takes constant time per cell, and linear time over all cells.

Our modification to the algorithm is the following. Suppose we run the above algorithm and notice that $|B_c| \geq 256/\pi$ for some cell c (i.e. there exists a close pair of points within c). We then consider an arbitrarily-chosen subset of exactly $\lceil 256/\pi \rceil$ points from B_c, and compare the distances between every pair of points within this subset. Since the number of points in the subset is constant, this entails performing only a constant number of comparisons, adding only $O(1)$ extra running time to the algorithm. Moreover, we know from the packing argument in class that among any subset of $\lceil 256/\pi \rceil$ points in the same cell, *some* pair must be close, so we are guaranteed to find a close pair using this technique if there exists a cell c such that $|B_c| \geq 256/\pi$. If there is no such cell, then we continue exactly as before, spending constant time per cell to determine if there exists a close pair of points in any cell.

Problem 7-2. TA Location Problem

Let's actually consider part (b) first. This problem decomposes by coordinates into two instances of the one-dimensional problem from part (a). That is, we want to select a point q which minimizes:

$$\sum_{i=1}^{n} |p_i^x - q^x| + |p_i^y - q^y| = \sum_{i=1}^{n} |p_i^x - q^x| + \sum_{i=1}^{n} |p_i^y - q^y|.$$

Note that, of the two summations above, the first only depends on x coordinates whereas the second depends only on y coordinates. Therefore, we can minimize the entire expression simply by picking q^x to minimize the first expression and by picking q^y to minimize the second expression. The subproblems of optimally selecting values of q^x and q^y are each the same as the problem from part (a). This is an important general observation: oftentimes one may decompose an optimization problem into a collection of smaller, simpler independent problems.

To solve part (a), we wish to select a point q minimizing

$$\sum_{i=1}^{n} |p_i - q|.$$

It turns out that the optimal value of q will be nothing more than the median value of the p_i's. Why is this? Well, let's suppose for the purposes of contradiction that the optimal value of q (which we denote by q^*) isn't the median of the p_i's. In this case, consider the set A of values p_i strictly less than q^*, and B of values p_i strictly greater than q^*. This allows us to write our objective function without using absolute values as follows:

$$\sum_{i=1}^{n} |p_i - q^*| = \sum_{p \in A} q^* - p + \sum_{p \in B} p - q^*.$$

Suppose now we change the value of q^* by some tiny amount ϵ. If $|A| > |B|$, we will choose ϵ to be negative; otherwise, if $|A| < |B|$ we will take ϵ to be positive. Note that $|A| \neq |B|$, since we're assuming that q^* is not the median of the p_i's. By adding ϵ to q^*, all terms in the summation over A will increase by ϵ whereas all terms in the summation over B will decrease by ϵ, for a net change of $|A|\epsilon - |B|\epsilon = (|A| - |B|)\epsilon$. By our choice of the sign of ϵ, this is always a negative quantity, which contradicts the optimality of q^*. That is, if q^* were not chosen as the median of the p_i's then it could be "improved" by moving it by some tiny amount ϵ in the appropriate direction. Note that if the number of points n is even, then the median value is not necessarily unique – however, the above argument works for any value between the pair of median elements.

Since we can compute the median of n numbers in $O(n)$ time, our algorithm for solving both parts (a) and (b) requires only $O(n)$ time.

Problem 7-3. Orthogonal Intersection Counting

Following the same structure as the solution to the previous problem, let's first consider a simpler one-dimensional problem. Suppose we're given a collection of n points on a one-dimensional number line which are stored in a balanced binary search tree, and we want to count the number of points which appear in a given interval $[a, b]$. This problem can be solved in $O(\log n)$ time through the use of an augmented balanced BST known as an order-statistic tree (studied in recitation, and also appearing in CLRS, section 14.1), which can compute in $O(\log n)$ time the rank of any given element stored within the tree. Supposing our points are stored in an order-statistic tree, we do the following: look up the point p_a which has the lowest x value greater than or equal to a (this requires only $O(\log n)$ time), and the point p_b wich has the highest x value from among all points less than or equal to b (also $O(\log n)$ time). The number of elements in the interval $[a, b]$ is then given by $\textsc{Rank}(p_b) - \textsc{Rank}(p_a) + 1$, or zero if $p_b < p_a$. We will now use this result to solve the intersection-counting problem.

As discussed in lecture, we will use a sweep-line approach for computing the number of intersection points among n vertical and horizontal line segments. We will sweep a vertical line from left to right across the plane, stopping at every endpoint of each horizontal segment, and also stopping at each vertical segment. We will maintain an order statistic tree of the y coorindate values of all of the horizontal segments which currently intersect the sweep line – we call such segments "active segments". Any time the sweep line stops at the left endpoint of a horizontal line segment S, we add S to the order statistic tree as it will now be active. Similarly, when the sweep line finally encounters the right endpoint of S, we will remove S from the tree. Since tree insertions and removals take $O(\log n)$ time each, and since we perform at most n of these due to horizontal segments, we are so far using only $O(n \log n)$ running time. When the sweep line encounters a vertical segment S, we simply need to count the number of active horizontal segments intersected by S. This is exactly the problem discussed above, since we must find all active segments having y coordinates between the y coordinates of the endpoints of S. Since we can solve for the number of intersection points with each individual vertical segment in $O(\log n)$ time using the method outlined above, and since there are at most n vertical segments, we spend only $O(n \log n)$ time processing vertical segments, for a total running time of $O(n \log n)$.

Note that we are allowed to make the simplifying assumption that all x and y coordinate values in the input are distinct. This simplifies the problem considerably since at each "interesting" x value where the sweep line stops, *exactly* one of the following occurs: (i) a single horizontal segment starts, (ii) a single horizontal segment ends, or (iii) there is a single vertical segment. Without our simplifying assumption, we'd need to handle many more special cases, for example coincident horizontal or vertical segments.