Solutions to Homework 2. Due April 23

CS180: Algorithms and Complexity
Spring 2018

1. In class we gave an algorithm for finding the closest pair of points on the plane. The notion of distance we used was Euclidean distance: $d((x,y),(x',y')) = \sqrt{(x-x')^2+(y-y')^2}$.

   Show how you can adapt the algorithm to find the closest pair for a different notion of distance. Specifically, give an algorithm with run-time $O(n \log n)$ for the following problem. Given a set of points $P = \{p_1, \ldots, p_n\}$ in the plane, find a pair of points with the smallest possible L1-distance among the points where L1-distance between two points is defined by $d_1((x,y),(x',y')) = |x-x'| + |y-y'|$.

   For full-credit your algorithm should be correct and you should explain why your algorithm works. In particular, if your algorithm is similar to what we did in class, you need to prove the analogue of the main lemma (one that allowed us to look only 15 distances ahead for Euclidean distance) we did in class. [.75 points]

   Solution Nothing really changes: we can follow the exact same algorithm as before except that when looking through S_y we check points 19 positions ahead instead of 15 as we did in class. We just have to 'L1-version' of the main lemma saying that looking 19 positions ahead is enough for L1-distance as well. Concretely, let S be the set of points in the strip of points at distance at most δ from the separating line and let S_y be the set of points in the set S. Suppose we have two points $s, s' \in S$ with $d_1(s,s') < \delta$, then we have to show that s' is no more than 15 positions away from s in S_y. Suppose that s occurs before s'.

   Now, split the region in the strip into rectangles whose length is $\delta/2$ and height is $\delta/3$. Then, just as before, we cannot have two points of S in any rectangle as this would contradict δ being the minimum of the distances on the same side. Second, if s is in a rectangle, then s' cannot be more than 4 rows above s as then the L1-distance just along the y-axis would be at least $4 \cdot (\delta/3) > \delta$. Thus, by the same reasoning as in class, the number of points between s, s' is at most $5 * 4 - 1 = 19$.

   Remark: The bound of 20 is not the best possible (you can improve it further if you do more fine-grained case analysis), but it suffices for the purpose of getting a $O(n \log n)$ algorithm. In fact, as long as you show that some constant number of positions is enough, you get a run-time of $O(n \log n)$.

2. An array $A[0, 1, \ldots, n-1]$ is said to have a *majority element* if more than half of its elements are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as mp3 files, say; so

in particular, you cannot sort the elements.) However you can answer questions of the form: "is $A[i] = A[j]$" in constant time.

Give an algorithm to solve the problem. For full-credit, your algorithm should run in time $O(n \log n)$ and you need to bound the time-complexity of the algorithm. (You don't have to prove correctness.) [.75 points]

(Hint: Split the array A into two arrays A_L and A_R of half the size each. Does knowing the majority elements of A_L and A_R help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

Solution. We split the original array into two halves A_L and A_R . The only way for the original array A to have a majority is if one of the subarrays (A_L or A_R) has a majority element. This is true because if both subarrays did not have a majority element, then that means that all elements occurred at most n/4 times in the halves. Therefore it would be impossible for any one of those elements to have occurred more than n/2 times in the array.

Using divide and conquer we can divide the problem until we hit the base case (say 3 element sized array) and then when merging, we can check to see if any of the majority elements from each respective subproblems is still the majority in the merged array (by counting the number of times the element occurs in the array).

Here is the algorithm. Suppose the algorithm returns NULL if there is no majority element.

```
FindMajority(A)
(a) If the array is size of less than 4, check by brute-force if there is a majority element.
(b) Split the array into two halves $A_L$ and $A_R$. (Takes $O(n)$ time).
(c) Let $\ell$ = FindMajority($A_L$), and $r$ = FindMajority($A_R$).
(d) If $\ell$ is not NULL, check if $\ell$ is a majority element in $A$ by counting the number of times
    $\ell$ occurs in $A$. (This takes $O(n)$ time.) If yes, return $\ell$.
(e) If $r$ is not NULL, check if $r$ is a majority element in $A$ by counting the number of times
    $r$ occurs in $A$. (This takes $O(n)$ time.) If yes, return $r$.
(f) Return NULL.
```

The run-time $T(n)$ of the algorithm satisfies the recurrence $T(n) \leq 2T(n/2) + O(n)$; the solution to this recurrence is $T(n) = O(n \log n)$ (as we did for mergesort by Case 2 of the master theorem).

Remark: (1) There is also a way to solve the problem in $O(n)$ time; these solutions will also be accepted. (2) A common mistake is to overlook that even if one of the two recursive calls returns NULL, there could still be a majority element in the array. For instance, say $A = [2, 3, 1, 1, 2, 1, 1, 1]$, then $A_L = [2, 3, 1, 1]$ and $A_R = [2, 1, 1, 1]$. Here A_R has a majority element 1, but A_L has none. nevertheless, 1 is a majority element in A.

3. Run the BFS algorithm for the following graph with $s = 1$ and $t = 9$: $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{4, 5\}, \{5, 6\}, \{8, 9\}\}$.

Your work should the step-by-step evolution of the lists $L_0, L_1, ...$ and of the array $DISCOVERED$ as we did in class (cf. the transcript uploaded in the calendar). [.75 points]

Solution. Here is what the evolution looks like:

(a) $L[0] = [1]$. Discovered $= [T \mid F \mid F \mid F \mid F \mid F \mid F \mid F \mid F \mid]$

(b) $L[1] = [2, 3]$. Discovered $= [T \mid T \mid T \mid F \mid F \mid F \mid F \mid F \mid F \mid]$

(c) $L[2] = [4, 5, 7]$. Discovered $= [T \mid T \mid T \mid T \mid T \mid F \mid T \mid F \mid F \mid]$

(d) $L[3] = [6]$. Discovered $= [T \mid T \mid T \mid T \mid T \mid T \mid T \mid F \mid F \mid]$

(e) $L[4] = \emptyset$. Discovered $= [T \mid T \mid T \mid T \mid T \mid T \mid T \mid F \mid F \mid]$

4. Given a graph $G = (V, E)$ in adjacency list representation, give an algorithm that runs in time $O(|V| \cdot |E|)$ to check if G has a 'triangle', i.e., a triple of distinct vertices $\{u, v, w\}$ such that all three edges between them are present in G. [.75 points]

Solution. The required run-time complexity of $O(|V| \cdot |E|)$ hints at how we can approach the problem.

For each edge $\{u, v\}$ in the graph, we need to find at least one vertex w, such that there exist edges $\{u, w\}$ and $\{w, v\}$. We could do this by finding the intersections of the adjacency lists of u and v. In particular, we will have a subroutine $lookForCommonNeighbor(u, v)$ which checks if there is an element common to the adjacency lists of u, $A[u]$, and v, $A[v]$. We do this essentially by exploiting the fact the lists $A[u], A[v]$ have numbers from $\{1, \ldots, V\}$ (indicating the indices of the vertices). We create a "flag" array B initialized to all zeroes. We traverse the list $A[u]$ and 'flag' all elements of the list in B by setting the corresponding entries of B to 1. We then traverse the list $A[v]$ and check if any of these elements has already been flagged in our first pass through $A[u]$.

The resulting algorithm could be written as follows:

```
for each vertex $u \in V$ do
    for each neighbor $v$ of $u$ do
        if $lookForCommonNeighbor(u, v)$ then
            return True
        end if
    end for
end for

return False

function lookForCommonNeighbor(Vertex $u$, Vertex $v$)
    set $B \leftarrow$ array of $|V|$ zeros.            ▷ Neighbors of $u, v$ range from $1 : |V|$

    for each neighbor $w$ of $u$ do
        set $B[w] \leftarrow 1$
    end for

    for each neighbor $w$ of $v$ do
        if $B[w] == 1$ then                    ▷ Both vertexes are adjacent to $w$
            return True
        end if
    end for

    return False
end function
```

The algorithm to check for the intersection of the two adjacency lists uses extra memory, but runs in $O(|V|)$ time. The outer loop runs $|V|$ times, but for each vertex u, it runs exactly $degree(u)$ times (the length of the adjacency list of u). So the total number of times the inner function `lookForCommonNeighbor` is called is $\sum_u degree(u) = O(|E|)$. As each call takes time $O(|V|)$, the total run-time is $O(|V| \cdot |E|)$.