Solutions for Homework Assignment #4

Answer to Question 1.

Let $1 \le t \le b \le n$ and $1 \le \ell \le r \le n$. We denote by $A[t..b, \ell..r]$ the subarray of A consisting of rows t through b, and rows ℓ through r; the subarray is empty if $t > b$ or $\ell > r$ (Variable t stands for "top", b for "bottom", ℓ for "left", and r for "right".)

The divide-and-conquer algorithm 2DSearch(A, t, b, ℓ, r, x) shown in pseudocode below returns **true** if x is in $A[t..b, \ell..r]$ and returns **false** otherwise. Thus, to determine whether x is in A, we simply return the value returned by the call 2DSearch$(A, 1, n, 1, n, x)$.

```
    2DSearch(A, t, b, ℓ, r, x)
1   if t > b or ℓ > r then return false
2   elsif t = b and ℓ = r then return A[t, ℓ] = x
3   else
4       h := (t + b) div 2
5       v := (ℓ + r) div 2
6       if x > A[h, v] then
7           return 2DSearch(A, t, h, v+1, r, x) or 2DSearch(A, h+1, b, ℓ, v, x) or 2DSearch(A, h+1, b, v+1, r, x)
8       else
9           return 2DSearch(A, t, h, ℓ, v, x) or 2DSearch(A, t, h, v + 1, r, x) or 2DSearch(A, h + 1, b, ℓ, v, x)
```

We now explain how 2DSearch(A, t, b, ℓ, r, x) works, and simultaneously justify its correctness. If the subarray is empty then x is not in the array; and if $A[t..b, \ell..r]$ consists of a single element, i.e., if $t = b$ and $\ell = r$, then $A[t..b, \ell..r]$ contains x if and only if x is that element — i.e., if and only if $A[t, \ell] = x$. This is precisely what the algorithm does in these cases (lines 1-2). Otherwise, the algorithm computes the "horizontal mid-point" h between t and b, and the "vertical mid-point" v between ℓ and r (lines 3 and 4). It then compares x to the "middle" element of $A[t..b, \ell..r]$, i.e., $A[h, v]$ (line 6). Depending on the outcome of this comparison, the algorithm rules out one of the four quadrants of $A[t..b, \ell..r]$, and searches the remaining three quadrants (lines 7 and 9). More precisely, there are two cases:

Case 1. $x > A[h, v]$. Then clearly x cannot be in the top-left quadrant of $A[t..b, \ell..r]$: since the array is sorted by both row and column, all the elements in that quadrant are $\le A[h, v]$, and therefore $< x$. Thus, x is in $A[t..b, \ell..r]$ if and only if it is in one of the remaining three quadrants: top-right, bottom-left, or bottom-right — i.e., $A[t..h, v + 1..r]$, $A[h + 1..b, \ell..v]$, or $A[h + 1..b, v + 1..r]$, respectively. Assuming, inductively, that the algorithm works correctly for smaller arrays, each of the three calls to 2DSearch returns **true** if and only if x is in the corresponding quadrant of $A[t..b, \ell..r]$; by the preceding analysis, x is in $A[t..b, \ell..r]$ if and only if it is in one of these three quadrants. So, the algorithm returns the correct value in this case.

Case 2. $x \le A[h, v]$. If $x < A[h, v]$, by an argument similar to Case 1, x cannot be in the bottom-right quadrant, and so it is in $A[t..b, \ell..r]$ if and only if it is in the top-left, top-right, or bottom-right quadrant. If $x = A[h, v]$, then x is in the top-left quadrant. Either way, x is in $A[t..b, \ell..r]$ if and only if it is in the top-left, top-right, or bottom-right quadrant. As in Case 1, assuming inductively that the algorithm works correctly in arrays with fewer elements, it returns the correct value for $A[t..b, \ell..r]$ in this case as well.

We now analyze the running time of 2DSearch. Let n be the number of rows and columns of A. 2DSearch recursively searches three subarrays of size at most $\lceil n/2 \rceil$. Thus, the following recurrence describes the running time of 2DSearch:

$$T(n) = \begin{cases} 3T(\lceil n/2 \rceil) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 0 \text{ or } n = 1 \end{cases}$$

In terms of the parameters a, b and d of the general divide-and-conquer recurrence, in this case we have $a = 3$, $b = 2$ and $d = 0$. So we are in the case where $a > b^d$, and the solution of the recurrence satisfies $T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.5849\ldots})$.

Since $n \log n \in O(n^{1.5849\ldots})$ but $n^{1.5849\ldots} \notin O(n \log n)$, it follows that the simple algorithm, which does binary search on each row ignoring the fact that columns are ordered, is asymptotically faster than 2DSEARCH, which attempts to take advantage of both row- and column-ordering. Sometimes, it's better to be only half-clever!

Even better algorithm: Did you find one? If you tried, failed, and you need a hint: Start at the top right ($A[1..n]$) or bottom left ($A[n..1]$). You can find what you are looking for (or determine that it is not in the array) after at most n comparisons.

Answer to Question 2.

First we prove the proposition given in the hint. Let $a = \mathbf{ads}(S)$. It suffices to show that $a_m \leq a$. Suppose, for contradiction that $a_m > a$. Since $a_m = \min(a_0, a_1)$ we have that $a_0 > a$ and $a_1 > a$.

By definition, we have $a_0 = \big(p - \min(S)\big)/(|S_0| - 1)$ and $a_1 = \big(\max(S) - p\big)/(|S_1| - 1)$. Therefore $a_0(|S_0| - 1) = p - \min(S)$ and $a_1(|S_1| - 1) = \max(S) - p$. By adding these equalities, we get

$$a_0(|S_0| - 1) + a_1(|S_1| - 1) = \max(S) - \min(S). \qquad (*)$$

We have

$$\begin{aligned}
a &= \frac{\max(S) - \min(S)}{|S| - 1} && \text{[by definition of } a\text{]} \\
&= \frac{a_0(|S_0| - 1) + a_1(|S_1| - 1)}{|S| - 1} && \text{[by (*)]} \\
&> \frac{a(|S_0| - 1) + a(|S_1| - 1)}{|S| - 1} && \text{[since } a_0 > a \text{ and } a_1 > a\text{]} \\
&= \frac{a(|S_0| + |S_1| - 2)}{|S| - 1} \\
&= \frac{a(\cancel{|S| - 1})}{\cancel{|S| - 1}} && \text{[since } |S_0| + |S_1| = |S| + 1\text{]} \\
&= a
\end{aligned}$$

which is a contradiction. Therefore, $a_m \leq a$, as wanted.

Given this, we now have the following divide-and-conquer algorithm to find a close pair in S: First we use the (deterministic) linear-time order statistics algorithm to find the median p of S. Then, in linear time, we construct the sets S_0 and S_1 consisting of the elements of S that are at most p and of the elements of S that are at least p, respectively. As we construct each of these sets, we also keep track of the minimum and maximum number in S, and of the total number of elements in S. This way we can compute, in constant time, $\mathbf{ads}(S_0)$ and $\mathbf{ads}(S_1)$. If $\mathbf{ads}(S_0) \leq \mathbf{ads}(S_1)$, the above result implies that it suffices to find any close pair in S_0; otherwise, it suffices to find any close pair in S_1. Thus, we apply the algorithm recursively to S_0 or S_1, respectively. The algorithm is shown in pseudocode below.

```
FINDCLOSE(S)
if |S| ≤ 3 then return pair (among the at most three possibilities) with minimum distance
else
    p := D-SELECT(S, ⌈|S|/2⌉)
    S_0 := {x ∈ S : x ≤ p}
    S_1 := {x ∈ S : x ≥ p}
    if ads(S_0) ≤ ads(S_1) then return FINDCLOSE(S_0)
    else return FINDCLOSE(S_1)
```

The correctness of the algorithm follows immediately from the proposition in the hint, which we proved above. Its running time is described by the recurrence

$$T(n) = \begin{cases} T(n/2) + cn, & \text{if } n > 3 \\ 1, & \text{if } 2 \leq n \leq 3 \end{cases}$$

In terms of the parameters of the Master Theorem, we have $a = 1$, $b = 2$, and $d = 1$, so that $a < b^d$. Therefore, the first case of the Master Theorem applies here, and $T(n) = \Theta(n^d) = \Theta(n)$.