You're expected to work on the problems before coming to the lab. Discussion session is not meant to be a lecture. TAs will guide the discussion and correct your solutions if needed. We may not release solutions for all problems. If you're better prepared for discussion, you will learn more. TAs will record names of the students who actively engage in discussion and report them to the instructor. The instructor will factor in participation in final grade.

1. (Basic) We're given a *sorted* array $A[1 \cdots n]$. We want to find the kth smallest element of A. How much time do you need?
   Sol. $O(1)$. Return $A[k]$.

2. (Basic) We're given an array $A[1 \cdots n]$, which is not necessarily sorted. We want to find the kth smallest element of A. How much time do you need?
   Sol. We learned $O(n)$ time algorithms for Selection.

3. (Basic) We're given a *sorted* array $A[1 \cdots n]$ and a target number x. We want to find i such that $A[i] = x$ if such i exists. How much time do you need?
   Sol. $O(\log n)$. We can use binary search.

4. (Basic) We're given an array $A[1 \cdots n]$, which is not necessarily sorted, and a target number x. We want to find i such that $A[i] = x$ if such i exists. How much time do you need?
   Sol. $O(n)$ time. We literally need to read all elements in the worst case.

5. (Basic) Show that $T(n) = T(n/5) + O(n)$ implies $T(n) = O(n)$.
   Sol. We can solve it by repeatedly expanding the formula (see the lecture slides) or Master Theorem.

6. (Intermediate) Show how quicksort can be made to run in $O(n \log n)$ time in the worst case; assume that all elements are distinct.
   Sol. Run the deterministic linear time selection algorithm to find the ideal pivot (median) for partitioning. Then, the recurrence for the RT will be $T(n) = 2(n/2) + \Theta(n)$.

7. (Advanced) Recall the following deterministic select algorithm:

   (a) Divide the n elements into groups of size 5. So $n/5$ groups.
   (b) Find the median of each group.
   (c) Find the median x of the $n/5$ medians by a recursive call to Select.
   (d) Call Partition with x as the pivot.
   (e) Make a recursive call to Select either on the smaller elements or larger elements (depending on the situation); if the pivot is the answer we are done.

   Then, the recurrence for the running time was $T(n) = T(\frac{1}{5}n) + T(\frac{7}{10}n) + O(n)$ (Can you explain why?)

   Now suppose we change the algorithm as follows.

   (a) Divide the n elements into groups of size 7. So $n/7$ groups.
   (b) Find the median of each group.
   (c) Find the median x of the $n/7$ medians by a recursive call to Select.
   (d) Call Partition with x as the pivot.

(e) Make a recursive call to Select either on the smaller elements or larger elements (depending on the situation); if the pivot is the answer we are done.

Now what is the recurrence you obtain? Explain.
Sol. $T(n) = T(\frac{1}{7}n) + T(\frac{5}{7}n) + O(n)$. The second step takes $T(n/7)$. Do you see that the order of x is between $\frac{2}{7}n$ and $\frac{5}{7}n$? Then, do you see where $T(\frac{5}{7}n)$ comes from? Solving the recurrence, we will still get $T(n) = O(n)$.

Again, we change the algorithm as follows.

(a) Divide the n elements into groups of size 3. So $n/3$ groups.

(b) Find the median of each group.

(c) Find the median x of the $n/3$ medians by a recursive call to Select.

(d) Call Partition with x as the pivot.

(e) Make a recursive call to Select either on the smaller elements or larger elements (depending on the situation); if the pivot is the answer we are done.

Now what is the recurrence you obtain? Explain.
Sol. $T(n) = T(\frac{1}{3}n) + T(\frac{2}{3}n) + O(n)$. The second step takes $T(n/3)$. Do you see that the order of x is between $\frac{1}{3}n$ and $\frac{2}{3}n$? Then, do you see where $T(\frac{2}{3}n)$ comes from? Unfortunately, solving the recurrence, we only get $T(n) = O(n \log n)$.

8. (Very Advanced) Let $X[1 \cdots n]$ and $Y[1 \cdots n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\log^2 n)$ time algorithm to find the median of all $2n$ elements in arrays X and Y.
Sol. For simplicity, assume that no two elements have the same value. High-level idea: We would like to find x and y such that $A[x-1] < B[y] < A[x]$ such that $x + y = n$ (I'm ignoring boundary cases) — the other case $B[y-1] < A[x] < B[y]$ can be done similarly. So, for each x we find y such that $A[x-1] < B[y] < A[x]$ — this take $O(\log n)$ time for binary search on B. We also need to binary search in A to find x that leads to $x + y = n$.