1. Consider the following QuickSort algorithm (at a high level):

1: **function** QUICKSORT(A) ▷ A is an unsorted list of size $n > 1$
2: Pick "pivot" element $p \in A$ (e.g., let $p = A[0]$)
3: Partition A into $B = \{x \in A : \ x \leq p\}$ and $C = \{x \in A : \ x > p\}$ (in place)
4: Call QuickSort recursively on B and C (in place)
5: **end function**

Analyze the running time of this algorithm.

Ans: In order to find the running time we need to to write down a recurrence relation for the running time. The way the algorithm is defined in the problem it has no explicit base case. However, implicitly, the algorithm assumes its input has size to be bigger than 1. This means that we can safely re-write the algorithm to start by testing the input size. If the input size is ≤ 1 then the algorithm should do nothing. otherwise, it should carry out the partition and the recursive calls. Hence, we can consider a constant amount of work (testing if $n \leq 1$) to happen in the base case, so:

$$T(1) = \Theta(1)$$

For the general case of $n > 1$, the algorithm first partitions its input list A to get a new arrangement {elements $< p$, p, elements $> p$}, which takes linear time. Then, it makes recursive calls on each sublist B, C. So:

$$T(n) = T(|B|) + T(|C|) + \Theta(n) \quad \text{for n } > 1$$

The problem with this relation is that the size of B and C can change depending on the choice of p and particular realization of A.

To deal with this problem, we consider the worst-case running time of the QuickSort and try to characterize the worst-case worst-case behaviour of QuickSort. The worst-case happens when one of B or C is empty (i.e., p happened to be the minimum or maximum element in list A). In this case:

$$T(n) = T(n-1) + \Theta(n)$$

Is there a closed-form formula for $T(n)$ in this recursive relation? The recursive relation doesn't allow us to apply Master Theorem so do repeated substitution

$$\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= (T(n-2) + \Theta(n-1)) + \Theta(n) \\
&= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&= T(1) + \Theta(2) + \cdots + \Theta(n-1) + \Theta(n) && \text{(by aforementioned recursion relation)} \\
&= \Theta(1) + \Theta(2) + \cdots + \Theta(n) && \text{(by our analysis of base case)} \\
&= \Theta(1 + 2 + \cdots + n) && (\Theta(f(x)) + \Theta(g(x)) = \Theta(f(x) + g(x))) \\
&= \Theta\Big(\tfrac{n(n+1)}{2}\Big) && (\textstyle\sum_{i=1}^{n} x = \tfrac{n(n+1)}{2}) \\
&= \Theta(n^2)
\end{aligned}$$

But we have a sorting algorithms that run in worst-case time of $\Theta(n \log n)$ and here we have an algorithm that has a worst case of $\Theta(n^2)$ and this happens when the list is already sorted! So what is with the name "quick"?

The best-case runing time of "QuickSort" happens when B, C each have roughly half the elements. Then:

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

and the Master Theorem applies with $a = 2 = 2^1 = b^d$, so $T(n)$ is $\Theta(n \log n)$. We will see in CSC263 that on average (assuming every possible ordering of the input list is equally likely), the partition step generates sublists B, C that are "roughly" of equal size, and it's possible to show that in this case, $T(n)$ remains $\Theta(n \log n)$. Hence, on average the running time of "QuickSort" is $\Theta(n \log n)$.
□

2. Find a good (as efficient as possible) divide and conquer algorithm that given an unsorted list A of distinct numbers and a "rank" $k \in \{1, 2, \cdots, |A|\}$ returns the k^{th} smallest element of A.

Ans: If we do not consider the restriction that the algorithm should be a divide and conquer algorithm, we can simply sort A and return the element at position k. The running time of this algorithm is dominated by the sorting step which is $\Theta(n \log n)$. Therefore we expect our divide and conquer algorithm to have a comparable running time if not better.

Idea: We may not need to fully sort the list to find the k^{th} smallest element! We can use the partition process from QuickSort, but only recurse on one of the two sublists.

```
1: function FindK(A, k)
2:     Pick pivot element p ∈ A at random
3:     Partition A into B = {x ∈ A : x ≤ p}, C = {x ∈ A : x > p}
4:     if k = |B| + 1 then
5:         return p
6:     else if k < |B| + 1 then
7:         return FindK(B, k)
8:     else
9:         return FindK(C, k − |B| − 1)
10:    end if
11: end function
```

We can define the following recursive definition for $T(n)$:

$$T(n) = T(|B|) + T(|C|) + \Theta(n)$$

Similar to QuickSort, the runtime depends on the choice of pivot. In the worst-case B is empty and the algorithm recurses on C which gives us a $\Theta(n^2)$ running time. However, on average case we recurse on sub-lists of roughly equal size and hence the average running time of this random algorithm is $\Theta(n)$.

Remark 1. *We can use celever tricks to get worst-case performance of $\Theta(n)$ deterministically, but the constants are actually large enough that the randomized algorithm is still better in practice!*

□