DSC 40B - Homework 04

Due: Wednesday, February 8

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 p.m.

Note! Because the midterm is on Thursday, we would like to release the solutions to this homework immediately after the due date on Wednesday at midnight. Since this is before the late due date, we **cannot accept slip days for this homework**.

Problem 1.

Suppose a binary search tree has been augmented so that each node contains an additional attribute called `size` which contains the number of nodes in the subtree rooted at that node. Complete the following code so that it computes the value of the kth smallest key in the tree, where $k = 1$ is the minimum.

```
def order_statistic(node, k):
    if node.left is None:
        left_size = 0
    else:
        left_size = node.left.size

    order = left_size + 1

    if order == k:
        return node.key
    elif order < k:
        return order_statistic(...)
    else:
        return order_statistic(...)
```

Problem 2.

Describe a strategy that, given a sorted array with n elements, constructs a balanced binary search tree in $\Theta(n)$ time.

This is not a programming problem, so there is no autograder. But you should provide pseudocode in your written answer – that is, code that doesn't necessarily run on a computer, but which makes your strategy precise.

Hint: what's the best element to use as the root?

Programming Problem 1.

Suppose you are trying to remove outliers from a data set consisting of points in $\mathbb{R}^d$. One of the simplest approaches is to remove points that are in "sparse" regions – that is, points that don't have many other points close by. To do this, we might calculate the distance from a point to it's kth closest neighbor. If this distance is above some threshold, we consider the point an outlier.

More generally, the task of finding the distance from a query point to its kth closest "neighbor" is a common one in data science and machine learning. Here, we'll consider the 1-dimensional version of the problem of finding kth neighbor distance. In a file named `knn_distance.py`, write a function named `knn_distance(arr, q, k)` that returns a pair of two things:

- the distance between `q` and the kth closest point to q in `arr`;
- the kth closest point to `q` in `arr`

The query point `q` does not need to be in `arr`. For simplicity, `arr` will be a Python list of numbers, and `q` will be a number. `k` should start counting at one, so that `knn_distance(arr, q, 1)` returns the distance between `q` and the point in `arr` closest to `q`. Your approach should have an expected time of $\Theta(n)$, where n is the size of the input list. Your function may modify `arr`. In cases of a tie, the point you return is arbitrary (though the distance is not). Your code can assume that k will be $\leq$ `len((arr)`.

Example:

```
>>> knn_distance([3, 10, 52, 15], 19, 1)
(4, 15)
>>> knn_distance([3, 10, 52, 15,], 19, 2)
(9, 10)
>>> knn_distance([3, 10, 52, 15], 19, 3)
(16, 3)
```

As this is a programming problem, submit your code to the Gradescope autograder.