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 subtree rooted at `node`, 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(...)
```

> **Solution:** If `order > k`, we should check the subtree of the left child. We are looking for the kth smallest thing among the nodes in the left subtree, so our recursive call is `order_statistic(node.left, k)`.
>
> On the other hand, if `order < k`, we should check the subtree of the right child, but we have to "update" the value of k. We do not want the kth smallest thing in the right subtree; rather, we want the `k - order` smallest thing (or, equivalently, `k - left_size - 1`).
>
> To see this, consider a simple example. Suppose we want the $k = 7$ smallest key in the tree and that `node.left.size` is 4. This makes the key of the current node the fifth smallest in the tree. All keys in the right subtree are larger, so we want the second smallest among them.
>
> The recursive call in this case is therefore `order_statistic(node.right, k - order)`.

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?

Solution: The general strategy is to pick the median as the root, then recurse – the root of the left subtree should be a median of the bottom half of elements, and the root of the right subtree should be a median of the upper half of elements. The code below implements this idea:

```python
class Node:

    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


def build_balanced_bst(sorted_arr, start, stop):
    if stop - start <= 0:
        return None

    mid_ix = (start + stop) // 2
    median = sorted_arr[mid_ix]
    root = Node(median)

    root.left = build_balanced_bst(sorted_arr, start, mid_ix)
    root.right = build_balanced_bst(sorted_arr, mid_ix + 1, stop)

    return 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.

Solution: The idea is to compute the distance from `q` to all of the points in `arr` in $\Theta(n)$ time, then use quickselect to find the kth order statistic in $\Theta(n)$ expected time. The tricky part is recovering the kth point from the kth distance. You could loop back through the distances searching for the index at which the kth distance occurs, then use this to index into `arr`; essentially, a linear search. This would still be linear time, but there's an approach that you might consider a little cleaner: instead of doing quickselect on the distances alone, run quickselect on tuples of distance/point pairs. This solution is shown below:

```python
import random


def knn_distance(arr, q, k):
    """Compute the kth nearest point and the distance to it."""

    # compute the distance between x and q
    def dist(x):
        return abs(x - q)

    # there's a small trick here that allows us to use `quickselect` from
    # lecture unmodified; instead of passing in a list of numbers, we pass in a
    # list of (distance, point) tuples. when Python compares two tuples, it
    # compares their first elements; if there is a tie, it then compares their
    # second elements, and so on. here, quickselect will find the pair with the
    # kth smallest first element, which is exactly what we need to return.
    distance_point_pairs = [(dist(x), x) for x in arr]
    return quickselect(distance_point_pairs, k, 0, len(arr))


# everything below is code for quickselect that comes unmodified from lecture

def in_place_partition(arr, start, stop, pivot_ix):
    def swap(ix_1, ix_2):
        arr[ix_1], arr[ix_2] = arr[ix_2], arr[ix_1]

    pivot = arr[pivot_ix]
    swap(pivot_ix, stop-1)
    middle_barrier = start
    for end_barrier in range(start, stop - 1):
        if arr[end_barrier] < pivot:
            swap(middle_barrier, end_barrier)
            middle_barrier += 1
        # else: do nothing
    swap(middle_barrier, stop-1)
    return middle_barrier


def quickselect(arr, k, start, stop):
    """Find kth order statistics in arr[start, stop]"""
    pivot_ix = random.randrange(start, stop)
```

```
    pivot_ix = in_place_partition(arr, start, stop, pivot_ix)
    pivot_order = pivot_ix + 1
    if pivot_order == k:
        return arr[pivot_ix]
    elif pivot_order < k:
        return quickselect(arr, k, pivot_ix+1, stop)
    else:
        return quickselect(arr, k, start, pivot_ix)
```