

---

## DSC 40B - Homework 04

Due: Monday, October 24

---

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.

### Problem 0. (1 Point Extra Credit on Midterm)

We'd like to know what you think about DSC 40B and the DSC program overall. We've posted a mid-quarter survey here:

<https://forms.gle/gEeLri8ds4hiHfZx7>

The survey is totally anonymous (unless you include your email, but that's optional).

**If 80% of the class fills out the survey before Midterm 01, we'll give everyone 1 point of extra credit on the midterm exam.** In addition, you'll get the warm fuzzy feeling of having made DSC a better place.

### 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  $k$ th 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(...)
```

**Solution:** Since the code only checks the size of the left root, whenever we traverse towards the right root, we can subtract the  $k$ th order value by the order of the left nodes value. This works as all the values on the left of the node already covers the the orders from  $(1, k-x)$  therefore the new  $k$ th value to look for is  $k_{\text{new}} = k_{\text{old}} - \text{order}$ . Using this we can get the solution to the algorithmn as,

#### Code

```
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(node.right, k - order)
else:
    return order_statistic(node.left, k)

```

## 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:

**Def** create\_bst(arr):

    Initialize a BST

**Def** recursive\_algorithm

**if** length of array is 0:

            return none

**if** length of array is 1:

            Insert arr[0] into BST

        mid = midpoint of array rounded down

        Insert arr[mid] into BST

        return recursive\_algorithm for the left and right side of the midpoint

    return BST

## 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 its  $k$ th 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  $k$ th 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  $k$ th 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  $k$ th closest point to  $q$  in `arr`;
- the  $k$ th 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  $\leq \text{len}(\text{arr})$ .

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:

```
import math
import random

def partition(arr, start, stop, pivot_ix):
    """Partition arr[start:stop] around pivot."""
    left = []
    pivot_count = 0
    right = []
    pivot = arr[pivot_ix]
    for ix in range(start, stop):
        if arr[ix] < pivot:
            left.append(arr[ix])
        elif arr[ix] == pivot:
            pivot_count += 1
        else:
            right.append(arr[ix])
    ix = start
    for x in left:
        arr[ix] = x
        ix += 1
    for i in range(pivot_count):
        arr[ix] = pivot
        ix += 1
    for x in right:
        arr[ix] = x
        ix += 1
    return start + len(left)

def quickselect(arr, k, start, stop):
    """Finds kth order statistic in numbers[start:stop)"""
    if k > len(arr):
        return math.inf

    pivot_ix = random.randrange(start, stop)
    pivot_ix = 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)
```

```

def knn_distance(arr, q, k):
    less = []
    more = []

    for ele in arr:
        if ele <= q:
            less.append(ele)
        else:
            more.append(ele)

    from_less = quickselect(less, len(less) - k + 1, 0, len(less))
    from_more = quickselect(more, k, 0, len(more))

    out_diff = 0
    out_val = 0
    if abs(from_less - q) < abs(from_more - q):
        out_diff = abs(q - from_less)
        out_val = from_less
    else:
        out_diff = abs(q - from_more)
        out_val = from_more

    return (out_diff, out_val)

```