DSC 40B - Homework 04

Due: Tuesday, February 2

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 on Tuesday at 11:59 p.m.

Problem 0.

We'd like to hear how things are going. What are we doing that is working? What could we be doing better? If you fill out the Google Form below corresponding to your section, we'll give you two points of extra credit on this homework.

- Dr. Eldridge's section: https://forms.gle/m3YGfWYUaqT2DUc67
- Dr. Wang's section: https://forms.gle/gw57anSKEoQKp2aW6

The form is completely anonymous, so we'll use the honor system to assign credit for this problem. To get your extra credit, simply write in your answer to this question that you filled out the form.

Problem 1. *(Eligible for Redemption)*

What is the *best* case time complexity of `quickselect`? Describe why and provide the input that yields this best case. For simplicity, you may assume that Quickselect always chooses the last element of the input array as the pivot (that is, the pivot is *not* chosen randomly).

Solution: The best case time complexity of quickselect is $\Theta(n)$, where n is the size of the input array.

In the best case, the element chosen as the first pivot is exactly the order statistic being queried so that no recursive calls are made. However, we still must perform the partition operation to count the number of elements to the left and right of the pivot, and this takes linear time.

Alternatively, another case which also leads to $\Theta(n)$ run time (and so is also a "best case") is when the chosen pivot is not the order statistic we are looking for, but it is the median (or close to it). Then there is still a recursive call on an array of size $n/2$, plus $\Theta(n)$ work for the partition. If the recursive calls also choose the median as their pivot, this gives a recurrence of $T(n) = T(n/2) + n$. Solving this gives a time complexity of $\Theta(n)$.

Problem 2. *(Eligible for Redemption)*

Determine the worst case time complexities of the following functions under the assumption that the binary search tree being operated is balanced tree of n nodes.

For this problem, assume that we represent a binary search tree node as an object with `left`, `right`, `key`, and `parent` attributes. If `left`, `right`, or `parent` are `None`, it means that the node does not have a left/right child or a parent, respectively.

a) Assume that the input, `root`, is the root of the binary search tree (and is not `None`).

```
def foo(root):
    node = root
    while node is not None:
        parent = node.parent
        node = node.left
    return parent.key
```

Solution: This will take $\Theta(\log n)$ in the worst case, which is when it must traverse the entire height of the tree.

b) Assume that the input, `root`, is the root of the binary search tree (and is not `None`).

```
def bar(root):
    if root is not None:
        bar(root.left)
        print(root.key)
        bar(root.right)
```

Solution: This will take $\Theta(n)$ in the worst case (and in the best case), as it visits each node in the tree exactly once.

Problem 3. *(Eligible for Redemption)*

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(...)
```

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 4.

Consider this version of quicksort given below. It is essentially the same as that given in lecture, except that 1) it always uses the last element of the array as the pivot, and 2) it has a `print` statement inserted at a crucial place.

```
def quicksort(arr, start, stop):
    """Sort arr[start:stop] in-place."""
    if stop - start > 1:
        pivot_ix = partition(arr, start, stop, stop-1)
        quicksort(arr, start, pivot_ix)
        quicksort(arr, pivot_ix+1, stop)


def 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:
            print('hello')
            swap(middle_barrier, end_barrier)
            middle_barrier += 1
        # else:
            # do nothing
    swap(middle_barrier, stop-1)
    return middle_barrier
```

Suppose `arr` is an array of length n with entries `[1, 2, 3, ..., n]`, where n is some large integer. If `quicksort(arr, 0, n)` is run, exactly how many times will `"hello"` be printed to the screen? Your answer should be an expression involving n, and should not involve $\sum$ or $\ldots$. Show your work.

Solution: Suppose we make a call to `quicksort` on a sorted array of size k. The pivot is set to the last (and largest) element in the array, therefore the condition of the `if`-statement in `partition` always evaluates to true, printing `"hello"` $k-1$ times.

When the array is sorted, the root call to `quicksort` spawns two recursive calls; one on an array of size $n-1$ and the other on an array of size zero. The first recursive call spawns two calls; one on an array of size $n-2$ and the other on an array of size zero. And so forth. Only the recursive calls to arrays of size > 1 result in a call to `partition`. Therefore `partition` is called on arrays of size $n-1$, $n-2$, $n-3$, …, 3, 2, and the total number of printed `"hello"`s is:

$$(n-2)+(n-3)+\ldots+3+2+1=\frac{(n-1)n}{2}.$$