# DSC 40B - Discussion 04

**Problem 1.**

The python code for quickselect is given below.

```
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=0
    for end_barrier in range(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):
    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)
```

**a)** Suppose we use quickselect to select the minimum element of the array $A = [3, 2, 9, 0, 7, 5, 4, 8, 6, 1]$. Describe a sequence of partitions that results in a worst-case performance of quickselect.

> **Solution:** For the worst case performance, the minimum element should be chosen as the pivot element at the very end. Moreover, we also want the array to be partitioned such that if arr has n elements, then partition on which the recursive call is made has n-1 elements. This can be done by choosing the maximum element in the array as the pivot. Hence, the pivot elements should be chosen in the order 9,8,7,6,5,4,3,2,1,0.

**Problem 2.**

The python code for quicksort is given below.

```
def partition(arr, start, stop):
    left=[]
    right=[]
    pivot=arr[stop-1]
    print('Hello!')
    for ix in range(start,stop):
        if arr[ix]<pivot:
```

```
            left.append(arr[ix])
        else if arr[ix]>pivot:
            right.append(arr[ix])
    ix=start
    for x in left:
        arr[ix]=x
        ix+=1
    pivot_ix=ix
    arr[ix]=pivot
    ix+=1.
    for x in right:
        arr[ix]=x
        ix+=1
   return pivot_ix

def quicksort(arr,start,stop):
    if (stop-start) > 1:
        pivot_ix = partition(arr,start,stop)
        quicksort(arr,start, pivot_ix)
        quicksort(arr, pivot_ix+1, stop)
```

**a)** How many lines are printed when `quicksort([5,2,3,1,4],0,5)` is called? Your answer should be a number.

**Solution:** The line is printed once per each call to the partition function.

$$([5,2,3,1,4],0,5) \to ([2,3,1,4,5],0,3)([2,3,1,4,5],4,5) \textbf{ partition, 1 print}$$
$$([2,3,1,4,5],0,3) \to ([1,2,3,4,5],0,0)([1,2,3,4,5],1,3) \textbf{ partition, 1 print}$$
$$([1,2,3,4,5],0,0) \to \textit{Basecase}$$
$$([1,2,3,4,5],1,3) \to ([1,2,3,4,5],1,2)([1,2,3,4,5],3,3) \textbf{ partition, 1 print}$$
$$([1,2,3,4,5],1,2) \to \textit{Basecase}$$
$$([1,2,3,4,5],3,3) \to \textit{Basecase}$$
$$([1,2,3,4,5],4,5) \to \textit{Basecase}$$

The line is printed 3 times.

**b)** The code sorts an array in place in the ascending order. How would you modify the code so that the array is sorted in the descending order?

We need to modify only the partition function as follows:

**Solution:**

```
def partition(arr, start, stop):
    left=[]
    right=[]
    pivot=arr[stop-1]
    for ix in range(start,stop):
        if arr[ix]>pivot:
            left.append(arr[ix])
        else if arr[ix]<pivot:
```

```
            right.append(arr[ix])
    ix=start
    for x in left:
        arr[ix]=x
        ix+=1
    pivot_ix=ix
    arr[ix]=pivot
    ix+=1.
    for x in right:
        arr[ix]=x
        ix+=1
    return pivot_ix
```

**Problem 3.**

Given a sorted array, write a function which efficiently constructs a Balanced Binary Search Tree. (Hint: this can be done in $O(n)$ time!)

**a)** To help get started, here's a sample function header:

```
def sortedArrayToBST(arr):
      # your code here
```

**Solution:**

```
def sortedArrayToBST(arr):

    if not arr:
        return None

    # find median
    mid = (len(arr)) // 2

    # make the median element the root
    root = Node(arr[mid])

    # left subtree of root has all values < arr[mid]
    root.left = sortedArrayToBST(arr[:mid])

    # right subtree of root has all values > arr[mid]
    root.right = sortedArrayToBST(arr[mid+1:])

    return root
```