Paul Laliberte'
CSCI-3104, Algorithms
Written Assignment-4
Recitation TA: Ning Gao

P1:

If we are looking solely at the contents within the function then we have one bug:

1. **Bug (line 8):** i, j = 0, 0. **Fix:** i = -1. We can exclude j completely since we are using a for loop.

If we are looking at inputs and returns of the function, so that we can use our partition with quick-sort, then we have two additional bugs:

1. **Bug (line 4):** def wrongPartition(a) **Fix:** def wrongPartition(a, left, right). We can also exclude len(n) now and use right.
2. **Bug (line 18):** NO return of pivot. **Fix:** return i+1

Three arrays which will produce the wrong output if the partition is not fixed are:

1. Input: [10,5,1] Output: [10, 1, 5]
2. Input: [70,40,80,30,90,40] Output: [70,40,30,40,90,80]
3. Input: [54,26,22,17,77] Output: INDEXING ERROR

Fixed code with successful output (use of Prof. Sriram's partition check):

```
#---------Partition----------#

def partition(a, l, r):
    # Implement Lomuto partition algorithm
    pivot=a[r-1]
    i = l-1
    for j in range(l,r-1):
        if (a[j] <= pivot):
            i = i+1
            swap(a,i,j)

    swap(a,i+1,r-1)
    return i+1

#--------------------------#

def test(a):
    print('Input',a, 'Pivot:',a[len(a) -1])
    q = partition(a, 0, len(a))
    isPartitioned(a, q)                          #Prof. Sriram's function
    print('Output',a)
    print("\n")


a = [10,5,1]
b = [70,40,80,30,90,40]          #duplicates on the left
c = [54,26,22,17,77]

test(a)
test(b)
test(c)
```

(a) code

```
In [71]: %run myQuickSort.py
Input [10, 5, 1] Pivot: 1
The pivot is: 1
-> Partition is correct (trumpets please)
Output [1, 5, 10]


Input [70, 40, 80, 30, 90, 40] Pivot: 40
The pivot is: 40
-> Partition is correct (trumpets please)
Output [40, 30, 40, 70, 90, 80]


Input [54, 26, 22, 17, 77] Pivot: 77
The pivot is: 77
-> Partition is correct (trumpets please)
Output [54, 26, 22, 17, 77]
```

(b) output

P2:

I opted for a while loop instead of the given for loop in the write up.

Code with random partitioned array:

```
#---------------Partition----------------#

def lomutoTriple(a, l, r):
    pivot = a[r-1]              #pivot is the last element, by def. of lomuto
    i=0                         #set i=0, rather than i=-1
    k=r-1                       #k is the last index
    j=0                         #set j=0

    while j <= k:
        if a[j] < pivot:        #less than pivot -> swap, increment i,j up
            swap(a,i,j)
            i = i + 1
            j = j + 1
        elif a[j] is pivot:     #equal to pivot, leave in-place
            j = j + 1
        else:                   #greater than pivot -> swap, increment k down
            swap(a,j,k)
            k = k - 1

    return i

#----------------------------------------#
```

(a) code

```
In [35]: %run lomutoTriple.py
Before partition:  [8, 5, 4, 8, 4, 8, 5, 4, 10, 5]

Input [8, 5, 4, 8, 4, 8, 5, 4, 10, 5] Pivot: 5
The pivot is: 5
-> Partition is correct (trumpets please)
Output [4, 4, 4, 5, 5, 5, 8, 10, 8, 8]


Before partition:  [1, 9, 10, 9, 6, 3, 7, 1, 10, 5]

Input [1, 9, 10, 9, 6, 3, 7, 1, 10, 5] Pivot: 5
The pivot is: 5
-> Partition is correct (trumpets please)
Output [1, 1, 3, 5, 6, 7, 9, 10, 10, 9]


Before partition:  [7, 7, 2, 8, 9, 9, 9, 2, 0, 3]

Input [7, 7, 2, 8, 9, 9, 9, 2, 0, 3] Pivot: 3
The pivot is: 3
-> Partition is correct (trumpets please)
Output [0, 2, 2, 3, 9, 9, 9, 8, 7, 7]
```

(b) output

P3:

A) By choosing $2\lfloor\sqrt{n}\rfloor$ random elements, sorting, and using the median as a pivot, we lower the probability, on average, of a $\Theta(n^2)$. On average, because we chose a random element as our pivot, the subarrays $A[1\ldots q-1]$ and $A[q+1\ldots n]$ will be close to equal size.

B) From (A) we know that we have close to equal subarrays, and partially sorted arrays as a result also. Since we will not have to compare every element in the subarrays, insertion sort will run linearly, $\Theta(n)$.

C) The general structure of the recurrence relation for quick-sort is

$$T(n) = T(k) + T(n-k) + Cn.$$

But we do not have an ordinary, average runtime, quick-sort because we picked a pivot that made subarrays $A[1\ldots q-1]$ and $A[q+1\ldots n]$ close to equal size the majority of the time. Then, on average, our quick-sort should run much quicker since we do not have as much overhead. For a definitive recurrence relation, we let $k=\frac{n}{2}$ and $k-n=\frac{n}{2}$ (partitioning our array in half). Therefore, the recurrence relation is

$$T(n) = 2T(\frac{n}{2}) + Cn.$$

P4: From (C) our recurrence relation is

$$T(n) = 2T(\frac{n}{2}) + Cn.$$

We could use the expansion method, but this relation is suitable for the Master Method. By Master Method, $a=2, b=2, f(n)=n$. Then

$$n^{log_2 2} \Rightarrow 2^n = 2, n = 1 \Rightarrow n^{(1)} = n.$$

Thus, $f(n) = n^{log_2 2} = n$. By case 2, if $f(n) = n$ then $T(n) = \Theta(n^{log_b a} lgn)$. Therefore,

$$T(n) = \Theta(nlgn).$$

This was exactly the goal in-mind for the runtime.