CS 315 - Day 15, More Quicksort

Today we'll discuss some modifications to Quicksort. As we mentioned, it's still an active research topic, and in particular we will explore ways to make it more efficient and to avoid the worst-case data ordering.

First, let's update your code to include timing for large N. We need to only time when we're sorting, which requires turning off output of intermediate results. In general, we would have something like the following.

```
import time
start = time.time()
# call sort function – nothing else
end = time.time()
print("N= ",N,"time= ",end-start)
```

But we want to systematize this, running for large N, calculating the ratio to ensure the algorithm is performing at the expected asymptotic efficiency. The below would also require `import math`.

```
for i in range(1,3):
  N = int(math.pow(10,i))
  A = list(range( N ) )
  random.shuffle(A)
  #print(A)
  start = time.time()
  Quicksort(A,0,N-1,False)
  end = time.time()
  ratio=(end-start)/(N*math.log(N,2))
  print(N,end-start,ratio)
  #print(A)
```

Note that the function above calculating the ratio is specifically for the ***average*** case for Quicksort, which theoretically is $O(N \cdot \lg(N))$. In practice, we use Python's `math.log` function, specifying that it's base 2.

Also, to confirm that Quicksort works, you could run for a smaller set of N, which above is 10^1 and 10^2, and uncommenting the `#print(A)` to ensure the result is in sorted order.

Once you feel confident that this mechanism is working, you can recomment the `#print(A)` and increase the range of the exponent i to run larger problems.

Upon review, the second rescursive call to Quicksort seems redundant. In general, if the last statement in a recursive function is a recursive call, this is called ***Tail Recursion***, and it can be replaced, which may improve efficiency.

Tail calls can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call, modified as appropriate (similar to overlay for processes, but for function calls). The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called **tail-call elimination** or **tail-call optimization**. Tail-call elimination allows procedure calls in tail position to be implemented as efficiently as goto statements, thus allowing efficient structured programming. In the words of

```
def Quicksort(A,p,r):
    while p < r:
        q  = Partition(A , p , r)
        Quicksort(A , p , q-1)
        p=q+1
```

Last class we explored a worst-case for Quicksort, which was already sorted (or reverse sorted) data. This happens because the element partitioned around is always the largest or smallest in the sub-list being partitioned, leading to a recurrence tree of height N. We can work to mitigate this by choosing an element to partition around more wisely.

One such method is called ***Median of Three*** partitioning. Rather than always take the last element in the sublist to be partitioned, take the median value of the elements in the first, middle, and last element.

For example, in `17 12 20 35 8 3 1 10`, the value to partition around would be the median value of

```
          |          |          |
        A[p]  A[(p+r)//2]  A[r]
```

the three elements: 17, 35, and 10. Note that the middle location $\frac{p+r}{2}$ would be calculated using integer arithmetic, since $p, \frac{p+r}{2}, q$ are subscripts of the locations in the list.

And then the median value would need to be swapped with `A[r]`, for compatibility with the rest of the Partition algorithm.

<u>In-class Lab:</u>

1. Start with a copy of your previous `Quicksort` implementation for each of these, and add a mechanism to time, iterate over large values of N, and calculate the ratio test for each value of N.

2. Using a copy of the code from #1, implement `Quicksort` with tail recursion.
Run the code for a small value of N and confirm that your modifications are correct and the output is sorted.
Do you notice a difference in the elapsed time between #1 and #2?

3. Using a copy of the code from #1, implement a `Partition` function that implements the Median of Three algorithm.
Run the code for a small value of N and confirm that your modifications are correct and the output is sorted.
Do you notice a difference in the elapsed time between #1, #2, and #3?

4. Run the code from #1, #2, and #3 on already sorted data (you could merely comment out the `shuffle`).
What happens?