Lab 8 — Performance Analysis

CS/ECE 374 B

October 16, 2019

- The goal of this lab is to practice coming up with asymptotic worst-case runtime complexity of algorithms. Note that in many cases, this involves showing an asymptotic upper and lower bound on the performance. An upper bound is often created by bounding components in the runtime or a recurrence. A lower bound can be created in the same way *or* by analyzing the performance of *some* case, since the worst-case performance is at least as bad as any given case.

1. Here you will analyze the performance of QuickSort that uses the median-of-three approach to select a pivot. In particular, this approach takes the first, last, and middle elements of the list and picks the median of these as the pivot. For example, given a list: [67, 19, 4, 44, 34, 45, 69, 81, 48, 70, 80], the algorithm would consider 67 (first), 45 (middle), and 80 (last) and pick the median of these (67) as the pivot.

   In this algorithm, the worst-case partition is when the pivot is the second smallest (or largest) element in the array, leading to the recurrence $T(n) = T(1) + T(n-2) + O(n)$. Given a list of elements, arrange them in an order to elicit this worst-case behavior on *every* call.

   For simplicity, assume that the Partition algorithm is stable, i.e., the algorithm preserves the ordering of elements in the two partitions. E.g., splitting the above list using 67 as the pivot results in the lists [19, 4, 44, 34, 45, 48] and [69, 81, 70, 80].

2. Consider the factorial program below:

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

   (a) Give a tight asymptotic bound on the number of bits it takes to represent $n!$. Hint: $(n/2)^{n/2} < n! < n^n$

   (b) Analyze the complexity of this algorithm given that multiplying a k-bit number by an l-bit number takes $\Theta(k \cdot l)$ time.

3. Analyze the complexity of the longest increasing subsequence algorithm from the last lab. Here is an implementation:

```
def LIS(L, bound=None):
    """ Computes the length of the longest increasing subsequence in list 'L',
    whose first element is greater than 'bound'. Call with 'bound=None'
    to represent no bound. """
    # base case: empty list
    if L == []:
        return 0
    # if we can include L[0] and keep the bound
    if bound is None or bound < L[0]:
        # the first term is the LIS that doesn't include L[0]
        # (so bound stays the same), the second term is
        # the LIS that does include L[0]
        return max(LIS(L[1:], bound), 1+LIS(L[1:], L[0]))
    else:
        # can't include L[0]
        return LIS(L[1:], bound)
```