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].

   Solution: We can work backwards from the base case. Given an array of length 3, the median-of-three approach will always split it into two arrays of length 1. Going backwards, we want to find an array of length 5 that gets split into an array of length 3 and another one of length 1. This can be done by sorting the array, and then putting the second largest element in the middle; e.g., [1, 2, 4, 3, 5].

   Continuing to work backwards, we now want a 7-element list that will cause the worst-case split. We add two elements, 6 and 7, by placing 6 in the middle and 7 at the end: [1, 2, 4, 6, 3, 5, 7]. We can continue on to build a 9-element list: [1, 2, 4, 6, 8, 3, 5, 7, 9].

   It is now easy to identify a pattern. Given a $2n+1$ element list, with the elements ordered as $[1, 2, 4, \ldots, 2n, 3, 5, 7, \ldots, 2n+1]$, the median of 3 approach will consider 1, $2n$, and $2n+1$ as the three possible pivot choices and pick $2n$ as the pivot. The split will result in $2n+1$ being in the upper partition, and $1, 2, 4, \ldots, 2(n-1), 3, 5, 7, \ldots, 2n-1$ in the lower partition in that order.

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: $(n/2)^{n/2} < n! < n^n$

Solution: $n!$ takes $\log n!$ bits to represent. By taking the log of the inequality, we see that:

$$\begin{aligned} \log\left((n/2)^{n/2}\right) \leq \log n! &\qquad \leq \log(n^n) \\ n/2(\log n - \log 2) \leq \log n! &\qquad \leq n \log n \end{aligned}$$

But $n/2(\log n - \log 2)$ is $\Theta(n \log n)$, therefore $\log n!$ is $\Theta(n \log n)$

(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.

Solution: After the recursive call, we need to multiply $(n-1)!$, which has $\Theta(n \log n)$ bits by n, which has $\Theta(\log n)$ bits. Therefore:

$$T(n) = T(n-1) + \Theta(n \log n \log n)$$

In other words:

$$T(n) = \sum_{i=1}^{n} i \log^2 i$$

(skipping the constant multipliers). We then have:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} i \log^2 i \\ &\leq \sum_{i=1}^{n} i \log^2 n \\ &= \log^2 n \sum_{i=1}^{n} i \\ &= \log^2 n \frac{n(n-1)}{2} \\ &= \Theta(n^2 \log^2 n) \end{aligned}$$

On the other hand

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} i \log^2 i \\ &\geq \sum_{i=n/2}^{n} i \log^2 i \\ &\geq \sum_{i=n/2}^{n} i \log^2 n/2 \\ &= \log^2(n/2) \sum_{i=n/2}^{n} i \\ &= \log^2(n/2)(\sum_{i=1}^{n} i - \sum_{i=1}^{n/2} i) \\ &= \Theta(n^2 \log^2 n) \end{aligned}$$

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)
```

Solution: Let $T(n)$ represent the worst-case running time of LIS on a list of size n. Note that the implementation performs constant work and makes at most 2 calls to to LIS with a list of $n-1$. So:

$$T(n) \leq 2T(n-1) + \Theta(1)$$

This is the same recurrence as for subset sum analyzed in class, resulting in $T(n) = O(2^n)$.

Next we need to find a worst-case example for LIS. Note that we make two recursive calls unless the bound (which is based on an earlier list element) is not smaller than the first element of the remaining list to consider. If the list is strictly increasing (i.e., $L[i] < L[i+1]$), the **if** branch will always be taken, resulting in 2^n recursive calls.

Next we can make one of two arguments:

- Since the increasing list explores all branches of the recursion tree, it exhibits worst-case performance. Since it takes $\Theta(2^n)$ steps, $T(n) = \Theta(2^n)$.
- Even if this is not the worst-case, the worst-case performance is bounded by below by the performance of any case. Therefore $T(n) = \Omega(2^n)$. But since we've also shown that $T(n) = O(2^n)$, we now have a tight asymptotic bound, $T(n) = \Theta(n)$