CPSC 320 Notes: Physics, Tug-o-War, and Divide-and-Conquer

October 16, 2017

In tug-o-war, two teams face each other and **carefully** pull on a **well-selected** rope (to avoid injury). The team that pulls the other past the centerline wins. The heavier team generally has the advantage.

1 Algorithmic Tug-o-War

1. Assuming you're given $2n$ people (for $n > 0$) and their weights and want to generate two teams that are fairly well-balanced, specify and solve at least two small examples.

2. Now, imagine you decide make these two tug-o-war teams $\{\text{heaviest}, 3rd \text{ heaviest}, 5th \text{ heaviest}, \ldots\}$ and $\{2nd \text{ heaviest}, 4th \text{ heaviest}, 6th \text{ heaviest}, \ldots\}$.

   (a) Create and solve a small instance to critique this approach. (I.e., make an instance that shows *dramatically* either that this produces an invalid solution or not a very good one. You'll need to decide what "invalid" and "good" mean!) Be sure to briefly explain what the problem is.

   (b) Assume you press on with this approach anyway using Algorithm #1: "While people remain, scan the line of people for the heaviest and move that person into Team A then scan the line for the heaviest (remaining) and move that person onto Team B."
   Give a good asymptotic bound on the runtime of this algorithm.

   (c) Give a more efficient algorithm—Algorithm #2—to implement the same approach and analyse its asymptotic runtime.

3. Now let's **switch problems** to a somewhat similar one. Imagine you are analysing weight measurements and want to find the *k-th* largest weight.[1]

   (a) Adapt your small instances above into instances of this problem—including choosing values of k—and solve them. (Note: **nothing** on this page is a reduction between this and the previous page's problems. We're just taking advantage of the fact that these problems are similar to avoid thinking too hard!)

   (b) Adapt Algorithm #1 above to solve this new problem and give a good asymptotic bound on its runtime.

   (c) Adapt your Algorithm #2 above to solve this new problem and give a good asymptotic bound on its runtime.

[1] A common element to search for would be the median. If you can solve the problem of finding the *k-th* largest, then you can solve the median problem as well by setting $k = \lceil \frac{n}{2} \rceil$.

2 Analysing QuickSort

Remember the QuickSort algorithm:

```
// Note: for simplicity, we assume all elements of A are unique
QuickSort(list A):
  If length of A is greater than 1:
    Select a pivot element p from A // Let's assume we use A[1] as the pivot
    Let Lesser  = all elements from A less than p
    Let Greater = all elements from A greater than p
    Let LesserSorted  = QuickSort(Lesser)
    Let GreaterSorted = QuickSort(Greater)
    Return the concatenation of LesserSorted, [p], and GreaterSorted
  Else:
    Return A
```

1. Assuming that QuickSort gets "lucky" and happens to always selects the $\lceil \frac{n}{4} \rceil$-$th$ largest element as its pivot, give a recurrence relation for the runtime of QuickSort.

2. Draw a recursion tree for QuickSort labeled on each node by the number of elements in the array at that node's call (n) and the amount of time taken by that node (but not its children); also label the total time for each "level" of calls. (For simplicity, ignore ceilings, floors, and the effect of the removal of the pivot element on the list sizes in recursive calls.)

3. Find the following two quantities. *Hint:* if you describe the problem size at level i as a function of i (like $i^2 + \frac{1}{2}i$), then you can set that equal to the problem size you expect at the leaves and solve for i.

   (a) The number of levels in the tree down to the shallowest leaf (base case):

   (b) The number of levels in the tree down to the deepest leaf:

4. Use these to asymptotically upper- and lower-bound the solution to your recurrence. (Note: if, on average, QuickSort takes two pivot selections to find a pivot at least this good, then your upper-bound also upper-bounds QuickSort's average-case performance.)

5. Draw the **specific** recursion tree generated by `QuickSort([10, 3, 5, 18, 1000, 2, 100, 11, 14])`. Assume QuickSort: (1) selects the first element as pivot and (2) maintains elements' relative order when producing `Lesser` and `Greater`.

3 Tug-o-War Winner (for Median)

Let's return to the *k-th* largest problem. We'll focus our attention on the median, but ensure we can generalize to finding the *k-th* largest element for any $1 \leq k \leq n$. The median in the call to `QuickSort([10, 3, 5, 18, 1000, 2, 100, 11, 14])` is the $5th$ largest element, 11.

1. Circle the nodes in your specific recursion tree above in which the median (11) appears.
2. Look at the first recursive call—below the root—that you circled. 11 is **not** the median of the array in that recursive call!
   (a) For what k is the median of that array the *k-th* largest?
   (b) What is the median?
   (c) For what k is 11 the *k-th* largest element of that array?
   (d) How does that relate to 11's original k value, and why?
3. Look at the second recursive call you circled. For what k is 11 the *k-th* largest element of that array? How does that relate to 11's k value in the first recursive call, and why?
4. If you're looking for the $42nd$ largest element in an array of 100 elements, and `Greater` has 41 elements, where is the element you're looking for?
5. How could you determine **before** making `QuickSort`'s recursive calls whether the *k-th* largest element is the pivot or appears in `Lesser` or `Greater`?
6. Modify the `QuickSort` algorithm above to make it a *k-th* largest element-finding algorithm. (Really, go up there and modify it directly with that pen/pencil you're holding. Change the function's name! Add a parameter! Feel the power!)
7. Give a good asymptotic bound on the average-case runtime of your algorithm by summing the runtime of only the $\frac{3n}{4}$ branch of your abstract recursion tree for QuickSort.

4 Challenge

Note: assume all elements are unique for the first two problems below.

1. Compare the average-case performance of your `QuickSelect` algorithm above against the performance of one that picks a random pivot (rather than using the first element).

2. Explain how this statement can possibly make sense for the random-pivot version of `QuickSelect`: "There is no difference between the best- and worst-case performance of this algorithm." Note: we instead use "expected performance" to describe this scenario.

3. Which one is better and why: good average-case performance or good expected performance?

4. Compare the actual (not asymptotic) number of comparisons made by the standard algorithm for finding the largest element of a list and `QuickSelect` used to do the same, assuming `QuickSelect` always "gets lucky" and picks the median of what remains as its pivot. P.S. Don't look here until after class, but more about the physics of tug-o-war and the reason for all the cautionary notes at the start are at what-if.xkcd.com/127.

5. Solve the tug-o-war team selection problem—selecting well-balanced teams every time—in polynomial time. **Warning:** there's a sense in which this is likely not possible and another (less theoretically accurate) sense in which it's possible but a tricky problem using a technique we haven't learned.