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

February 4, 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.

   SOLUTION: We could give lots of examples. Here's a couple as sets of weights in kilograms.

   (a) $\{10, 30\}$. The best teams we can make are $\{10\}$ and $\{30\}$, which is not very good. (The eight-year-old will totally beat the toddler.)

   (b) $\{70, 50, 30, 10\}$. There's a perfectly balanced team here in terms of total weights, which may be a good target: $\{70, 10\}$ and $\{50, 30\}$.

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.

   SOLUTION: A good way to show a **dramatic** difference is to put one large person (an adult?) on one side and several small ones (children?) on the other: $\{100, 20, 20, 20, 20, 20\}$.

   A good solution here is $\{100\}$ and $\{20, 20, 20, 20, 20\}$, which is perfectly balanced in terms of total weight.

   However, the algorithm above produces an absurdly unbalanced solution: $\{100, 20, 20\}$ and $\{20, 20, 20\}$, which gives the first team a 80kg (or factor of $2\frac{1}{3}$) advantage.

   If we didn't want to be realistic in the weights, we could exaggerate this issue arbitrarily.

   (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.

   SOLUTION: We'll look through n people to put the first person on a team, $n - 1$ for the second, $n - 2$ for the third, and so forth. That's a pattern that should be familiar: $n + (n - 1) + (n - 2) + \ldots + 2 + 1 + 0 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$.

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

SOLUTION: We don't need to look through each time. If we just sort in decreasing order of weight, we can send person 1 to Team A, person 2 to Team B, person 3 to Team A, and so on. Odd-numbered people go to Team A while even-number people go to Team B.

The sorting (assuming we use an efficient comparison-based algorithm) takes $O(n \lg n)$ time, which dominates the linear time needed to send people off to their teams in the single pass over the sorted array.

This $O(n \lg n)$ algorithm is clearly better than Algorithm #1.

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!)

SOLUTION: We had three small instances in the end.

i. $\{10, 30\}$, and we'll use $k = 1$. The result should be 30 (the "first largest" value, since $k = 1$).
ii. $\{70, 50, 30, 10\}$, and we'll use $k = 2$. That gives us 50, the second largest value.
iii. Modifying the last example a bit so we can tell people apart: $\{100, 22, 21, 20, 19, 18\}$, and we'll finish with $k = 5$, which gives us 19.

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

SOLUTION: We no longer need to put people on teams. So, while $k > 1$, we'll scan the list of weights remaining for the largest and throw that largest weight away. When $k = 1$, we do one last scan and return the largest remaining weight.

This is only the largest k iterations of the old algorithm's sum. That sounds like $O(nk)$, but let's work it through for fun:

$$\begin{aligned}
\sum_{i=n-k+1}^{n} i &= (\sum_{i=1}^{n} i) - (\sum_{i=1}^{n-k} i) \\
&= \frac{n(n+1)}{2} - \frac{(n-k)(n-k+1)}{2} \\
&= \frac{n^2 + n - n^2 - k^2 + 2nk - n + k}{2} \\
&= \frac{2nk - k^2 + k}{2} \\
&\leq \frac{2nk + k}{2} \\
&\in O(nk)
\end{aligned}$$

[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 $k = \lceil \frac{n}{2} \rceil$.

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

SOLUTION: Sort by descending order of weight and return the k-th element.

This is simpler to state than the previous algorithm, and the last part takes constant time, but the dominant factor is still the sorting: $O(n \lg n)$.

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.

   SOLUTION: A recurrence is just one of many models we make of code. In this case, we simplify the code by caring just about how many "primitive operations" (steps) it takes on an input of a particular size. Let's give a name to this: $T_Q(n)$ is the runtime (number of steps) of QuickSort on an array of length n.

   The conditional that chooses between the base and recursive cases takes constant time to evaluate.

   There's a base case when the array has 0 or 1 elements that takes constant time (presuming no copying) to just give back the existing list.

   The recursive case selects the pivot in constant time (though one could imagine slower ways to select the pivot), takes linear time to partition the elements into `Lesser` and `Greater` (and the pivot, in its own subset), makes two recursive calls, and then concatenates everything back together (likely in constant or linear time, depending on implementation).

   How long do those recursive calls take? We can describe them in terms of $T_Q(n)$. **Because the algorithm is recursive, our function modeling it is recursive, too!**

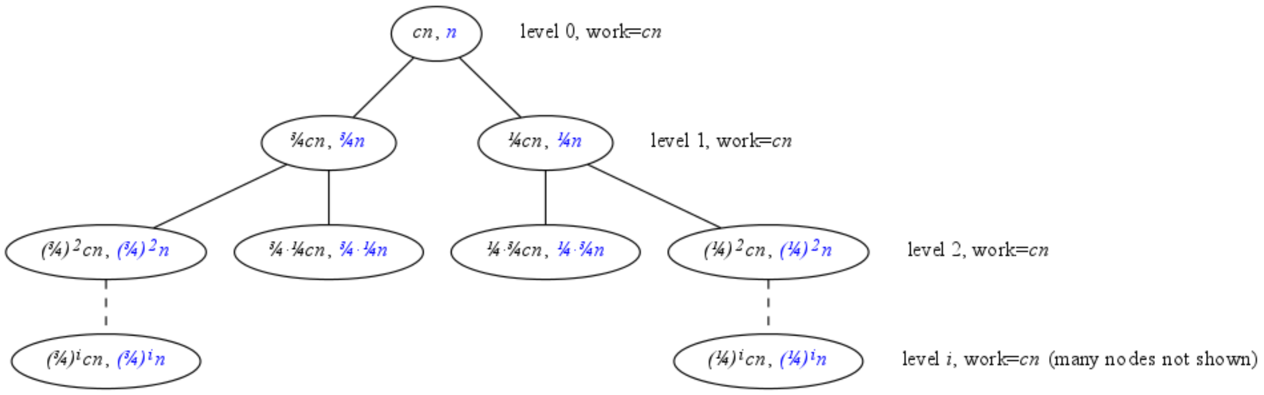
$$T_Q(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ T(\lceil \frac{n}{4} \rceil - 1) + T(\lfloor \frac{3n}{4} \rfloor) + O(n) & \text{otherwise} \end{cases}$$

   Note that adding "$O(n)$" is an abuse of terminology. We could instead add something like cn or even $cn + d$, but this will work fine. In fact, we'll also drop the floors, ceilings, and the minus one to simplify, since they won't change our analysis in the end:

$$T_Q(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ T(\frac{n}{4}) + T(\frac{3n}{4}) + O(n) & \text{otherwise} \end{cases}$$

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.)

   SOLUTION: Here are the first three levels of the recursion tree with the work done at that node (and not its children) in black and the array size at that node in blue. We also show a generalized form of the leftmost and rightmost branches at level i:


cn, n
level 0, work=cn
$\frac{3}{4}cn$, $\frac{3}{4}n$
$\frac{1}{4}cn$, $\frac{1}{4}n$
level 1, work=cn
$(\frac{3}{4})^2cn$, $(\frac{3}{4})^2n$
$\frac{3}{4}\cdot\frac{1}{4}cn$, $\frac{3}{4}\cdot\frac{1}{4}n$
$\frac{1}{4}\cdot\frac{3}{4}cn$, $\frac{1}{4}\cdot\frac{3}{4}n$
$(\frac{1}{4})^2cn$, $(\frac{1}{4})^2n$
level 2, work=cn
$(\frac{3}{4})^icn$, $(\frac{3}{4})^in$
$(\frac{1}{4})^icn$, $(\frac{1}{4})^in$
level i, work=cn (many nodes not shown)

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):

   SOLUTION: The $\frac{n}{4^i}$ branch will reach the base case fastest. If we set that equal to 1, we get $\frac{n}{4^i} = 1$, $n = 4^i$, and $\lg n = lg4^i = i \lg 4$. $\lg 4$ is just a constant (the constant 2), but we can also move it over to the far side to more clearly describe the number of levels: $i = \log_4 n$.

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

   SOLUTION: Similarly, the $(\frac{3}{4})^i n$ branch reaches the base case slowest, and by a similar analysis, we find $i = \log_{\frac{4}{3}} n$. That looks nasty but is only a constant factor away from $\log_4 n$. (Critically, $\frac{4}{3} > 1$; so, this really is a nice, normal log.)

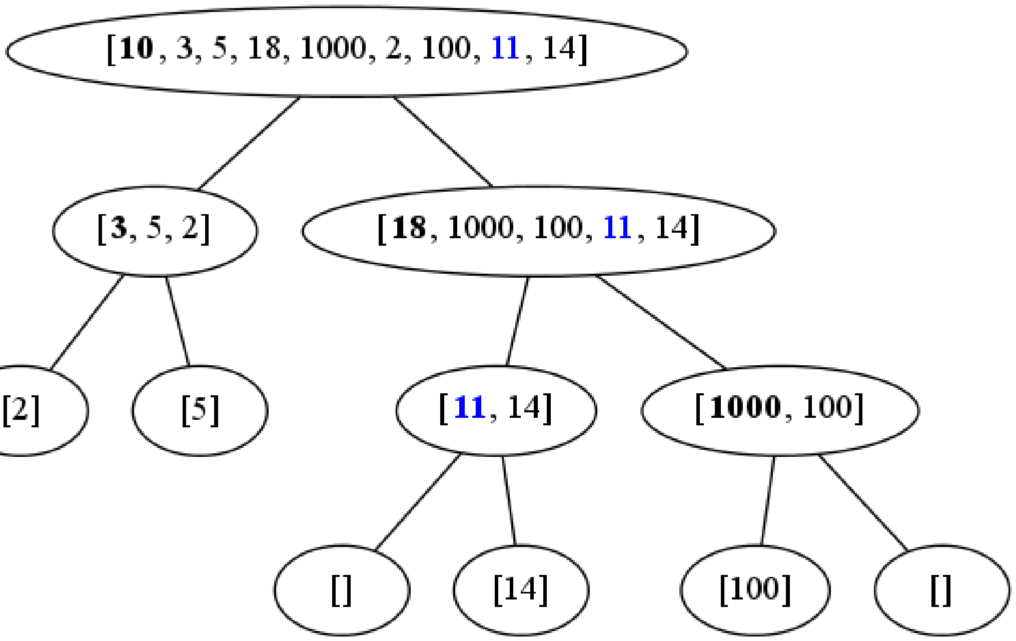
4. Use these to asymptotically upper- and lower-bound the solution to your recurrence. (Note: if, on average, QuickSort takes two pivot selections to find a pivot at least this good, then your upper-bound also upper-bounds QuickSort's average-case performance.)

   SOLUTION: We perform $O(n)$ work at each level, and in either case (lower or upper bound), there are $O(\lg n)$ levels. That $O(n \lg n)$ work total.

   (This is the "balanced" case of distribution of the work, as with MergeSort.)

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`.

   SOLUTION: Here it is with each node containing the array passed into that node's call, the pivot in **bold**, and the number 11 in blue.



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 5*th* largest element, 11.

1. Circle the nodes in your specific recursion tree above in which the median (11) appears.

   SOLUTION: The ones with the blue 11 in them above.

2. Look at the first recursive call 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? $k = 3$

   (b) What is the median? **18**

   (c) For what k is 11 the *k-th* largest element of that array? $k = 5$

   (d) How does that relate to 11's original k value, and why? **see below**

   SOLUTION: Note that we're looking at the node containing $[\mathbf{18}, 1000, 100, 11, 14]$. Most solutions are above. The new k value of **5** is the same as the old one. Why? 11 ended up on the right side (in `Greater`). So, everything that was larger than it before is still in the array. Thus, if it was the 5*th* largest before, it still is the 5*th* largest element.

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?

   SOLUTION: Note that we're looking at the node containing $[\mathbf{11}, 14]$. $k = 2$ this time. Why did it go down by 3? The pivot in the previous level was the 3*rd* largest element, and 11 ended up in `Lesser`. So, the pivot and everything larger is no longer "with" 11. With 3 fewer larger elements, it is now the 2*nd* largest elementh rather than the 5*th* largest.

4. If you're looking for the 42*nd* largest element in an array of 100 elements, and `Greater` has 41 elements, where is the element you're looking for?

   SOLUTION: If `Greater` has 41 elements, then there are 41 elements larger than the pivot. That makes the pivot the 42*nd* largest element. In other words, if the size of `Greater` is $k - 1$, then the pivot is the *k-th* largest element.

   (This is what happened—with much smaller numbers—in the node where 11 is also the pivot.)

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`?

   SOLUTION: Putting together what we saw above, if `|Greater|` is equal to $k - 1$, then the *k-th* largest element is the pivot. If `|Greater|` is smaller, then the pivot is larger than the *k-th* largest element, which puts it in the `Lesser` group (and the left recursive call). If `|Greater|` is larger, then the *k-th* largest element is in `Greater` (and the right recursive call).

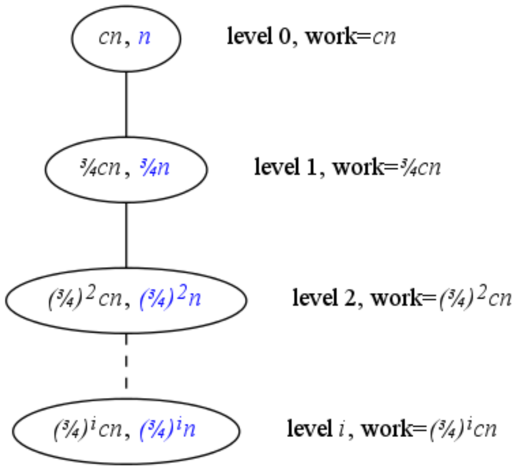
6. Modify the `QuickSort` algorithm above to make it a *k-th* largest element-finding algorithm. (Really, go up there and modify it directly with that pen/pencil you're holding. Change the function's name! Add a parameter! Feel the power!)

   SOLUTION: We copy the algorithm down and modify it. In particular, we no longer need the recursive calls on both sides, only on the side with the element we seek.

```
// Note: for simplicity, we assume all elements of A are unique
// Return the kth largest element of A. Precondition: |A| >= k.
QuickSelect(list A, k):
  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
  if |Greater| = k - 1:
    return p
  else if |Greater| > k - 1:
    // all larger elts are in Greater; k is unchanged
    return QuickSelect(Greater, k)
  else:  // |Greater| < k - 1
    // subtract from k the # of larger elts removed (Greater and the pivot)
    return QuickSelect(Lesser, k - |Greater| - 1)
```

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.

SOLUTION: Here's the new tree (well, stick):



cn, n
level 0, work=cn
$\frac{3}{4}cn$, $\frac{3}{4}n$
level 1, work=$\frac{3}{4}cn$
$(\frac{3}{4})^2cn$, $(\frac{3}{4})^2n$
level 2, work=$(\frac{3}{4})^2cn$
$(\frac{3}{4})^icn$, $(\frac{3}{4})^in$
level i, work=$(\frac{3}{4})^icn$

Summing the work at each level, we get: $\sum_{i=0}^{?}(\frac{3}{4})^i cn = cn\sum_{i=0}^{?}(\frac{3}{4})^i$. We've left the top of that sum as ? because it turns out not to be critical. This is a *converging* sum. If we let the sum go to infinity (which is fine for an upper-bound, since we're not making the sum any smaller by adding positive terms!), it still approaches a constant: $\frac{1}{1-\frac{3}{4}} = \frac{1}{\frac{1}{4}} = 4$. So the whole quantity approaches $4cn \in O(n)$.

In case you're curious, here's one way to analyse the sum. Let $S = \sum_{i=0}^{\infty}(\frac{3}{4})^i$. Then:

$$\begin{aligned}
S &= \sum_{i=0}^{\infty}(\frac{3}{4})^i \\
&= 1 + \sum_{i=1}^{\infty}(\frac{3}{4})^i \\
&= 1 + \sum_{i=0}^{\infty}(\frac{3}{4})^{i+1} \\
&= 1 + \frac{3}{4}\sum_{i=0}^{\infty}(\frac{3}{4})^i \\
&= 1 + \frac{3}{4}S
\end{aligned}$$

Solving $S = 1 + \frac{3}{4}S$ for S, we get $\frac{1}{4}S = 1$, and $S = 4$.