CPSC 320 Sample Solution: Programming competitions, CPSC 320 and Divide-and-Conquer

February 27, 2018

You have been put in charge of sending invitation emails to students whom the department of Computer Science and the department of Electrical and Computer Engineering would like to join one of the teams that UBC sends to the ACM international programming competition. You decide to contact the students who obtained the top k grades in CPSC 320 over the last year.

1 Algorithmic Largest k grades

1. Assuming you're given n students along with their CPSC 320 grades, and you want to get the students with the top k grades. Specify and solve at least two small examples.

   SOLUTION: We want examples that are small, but not trivial (so $k = 1$ does not work). Here are two examples what will suffice:

   - { 67, 98, 85, 73, 88 } with $k = 2$
   - { 91, 90, 77, 90, 82 } with $k = 3$

2. There are several ways of solving the problem of finding the students with the highest k grades. Design and describe short algorithms with the following properties:

   a. An algorithm based on the `max` function.

   SOLUTION:

```
while k > 0:
    find the maximum grade G; yield G
    remove G from the list
    decrement k
```

   b. An algorithm based on sorting.

   SOLUTION: We sort the list and return the last k elements.

   c. An algorithm that uses a max-heap.

   SOLUTION:

```
h = make_heap(list)
while k > 0:
    G = delete_max(h)
    yield G
    decrement k
```

3. Now give good asymptotic bounds on the runtime of each algorithm

   a. An algorithm based on the `max` function.

   SOLUTION: $\Theta(kn)$

   b. An algorithm based on sorting.

   SOLUTION: $\Theta(n \log n + k)$

   c. An algorithm that uses a max-heap.

   SOLUTION: $\Theta(n + k \log n)$

4. Now let's **switch problems** to a somewhat similar one. Imagine you simply want to find the *k-th* highest grade. [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 want the 2nd largest value of { 67, 98, 85, 73, 88} which is 88, and the 3rd largest value of { 91, 90, 77, 90, 82 } which is 90.

   b. Adapt the algorithm based on the `max` function to solve this new problem and give a good asymptotic bound on its runtime.

   SOLUTION:

```
while k > 0:
    find the maximum grade G
    remove G from the list
    decrement k
return G
```

   This algorithm runs in $\Theta(nk)$ time.

   c. Adapt the algorithm based on sorting to solve this new problem and give a good asymptotic bound on its runtime.

   SOLUTION: We sort the list and return the element at position `len(list) - k`. This takes $\Theta(n \log n)$ time.

   d. Adapt the algorithm that uses a max-heap to solve this new problem and give a good asymptotic bound on its runtime.

   c. An algorithm that uses a max-heap.

   SOLUTION:

```
h = make_heap(list)
while k > 0:
    G = delete_max(h)
    decrement k
return G
```

   This algorithm runs in $\Theta(n + k \log n)$ time.

[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.

   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!**

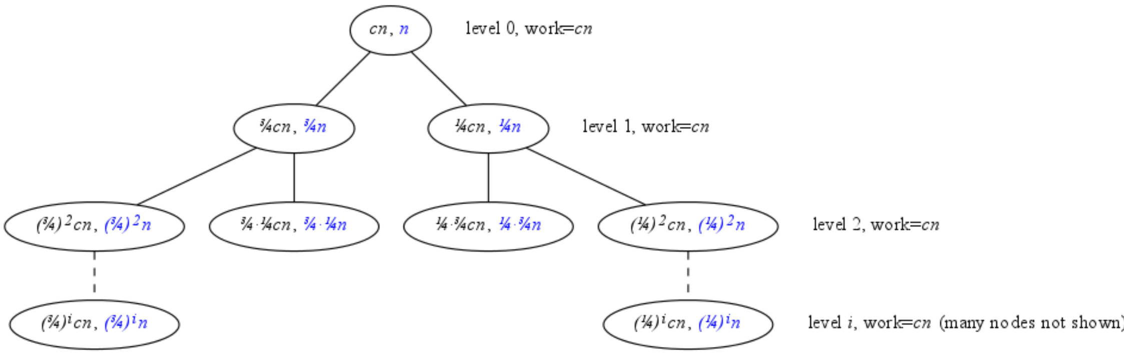
$$T_Q(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ T(\lceil \frac{n}{4} \rceil - 1) + T(\lfloor \frac{3n}{4} \rfloor) + O(n) & \text{otherwise} \end{cases}$$

   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:



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.)

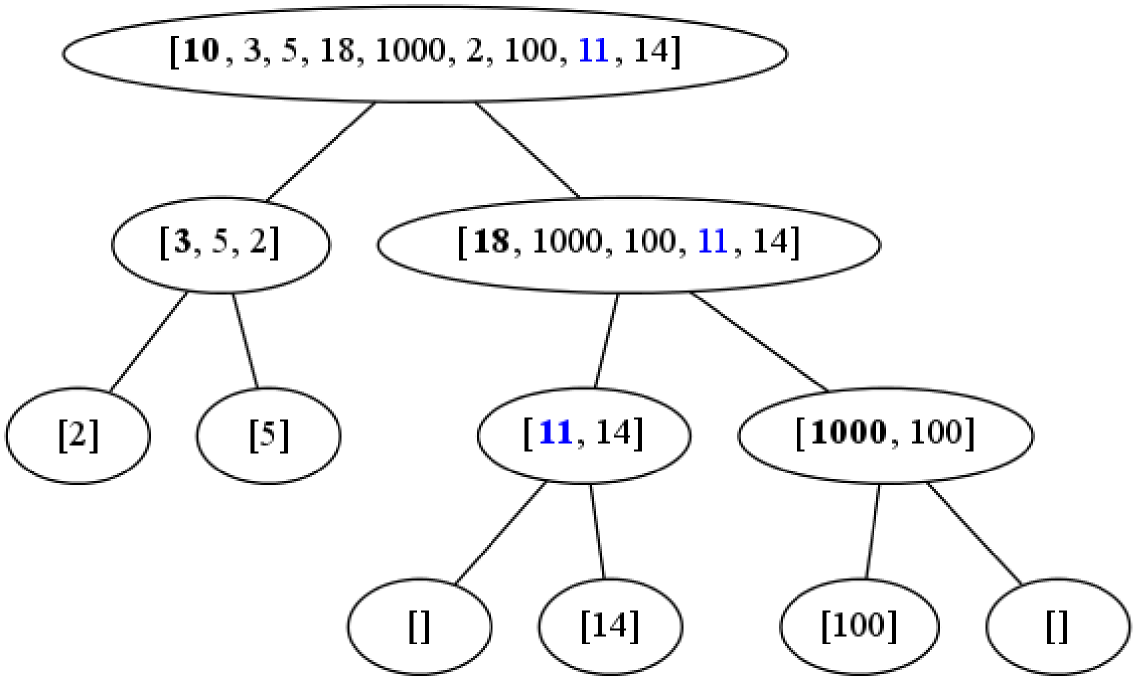
4. Use these to asymptotically upper- and lower-bound the solution to your recurrence. (Note: if, on average, QuickSort takes two pivot selections to find a pivot at least this good, then your upper-bound also upper-bounds QuickSort's average-case performance.)

   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 Largest k grades 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. Mark 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 second recursive call you marked—the one below the root. 11 is **not** the median of the array in that recursive call!

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

   (b) For what k is the median of that array the k-th largest? $k = 3$

   (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 $5th$ largest before, it still is the $5th$ largest element.

3. Look at the **third** recursive call you marked. 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 $3rd$ 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 $2nd$ largest elementh rather than the $5th$ largest.

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?

   SOLUTION: If `Greater` has 41 elements, then there are 41 elements larger than the pivot. That makes the pivot the $42nd$ 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).

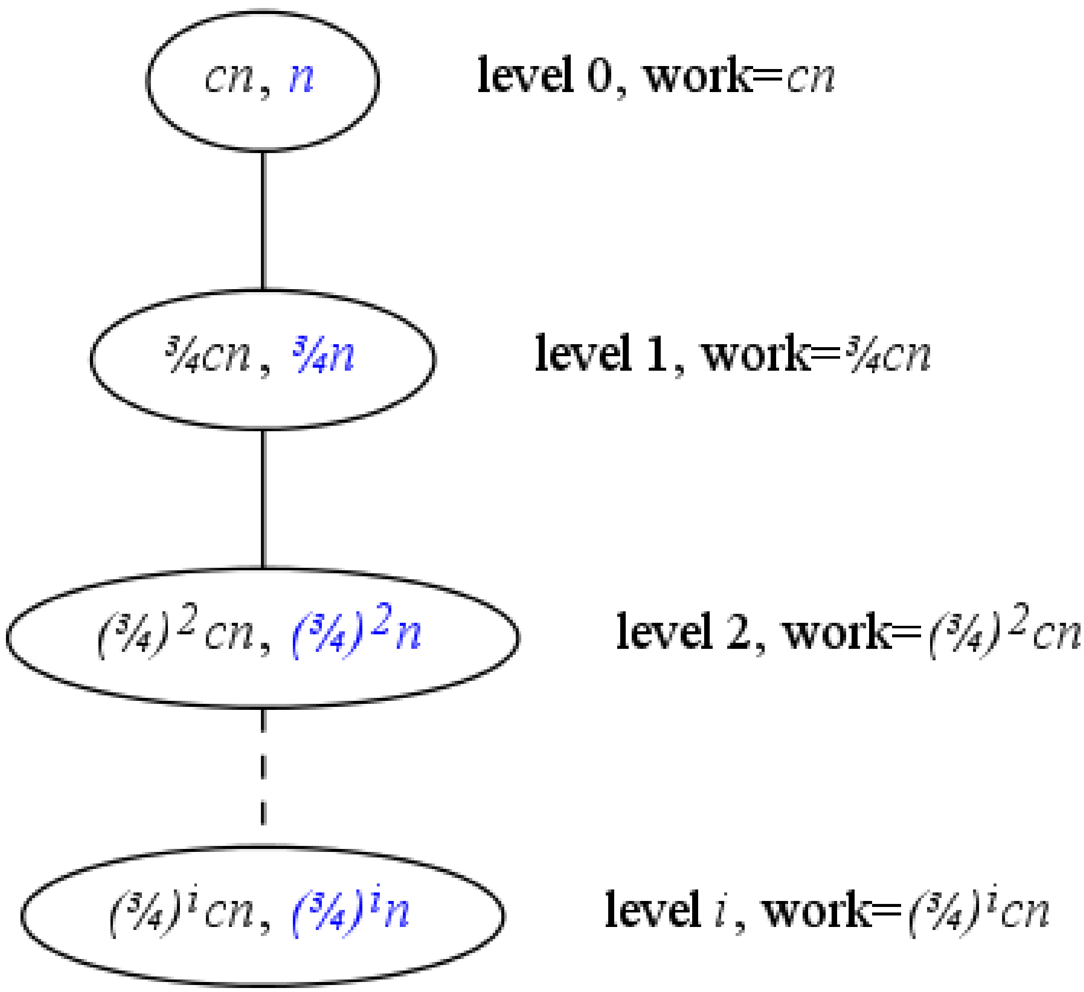
6. Modify the `QuickSort` algorithm above to make it a k-th largest element-finding algorithm. (Really, go up there and modify it directly with that pen/pencil you're holding. Change the function's name! Add a parameter! Feel the power!)

   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):



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 $S = 4$.