CPSC 320: Prune and Search Solutions*

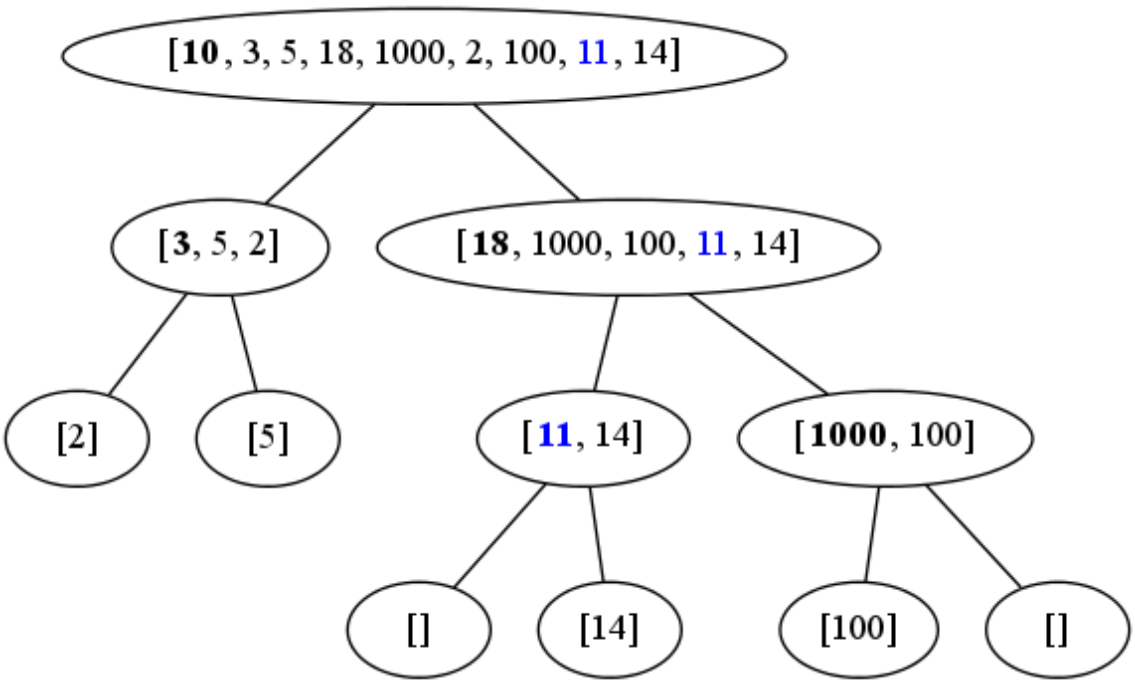
In this worksheet, we investigate an algorithm design technique called *Prune and Search*. It is used to search for an element (or a group of elements) with a specific property, and it is closely related to Divide and Conquer. The main difference between the two is that we only recurse on one subproblem, instead of all of them.

1 An Algorithm for Finding the Median

Suppose that you want to find the *median* value in an array (not necessarily sorted) of length n. The algorithms we will discuss can, in fact, be used to find the element of rank k, i.e., the kth smallest element in an array, for any $1 \leq k \leq n$. (Note: the larger the rank, the larger the element). The median is the element of rank $\lceil n/2 \rceil$. For example, the median in the array [10, 3, 5, 18, 1000, 2, 100, 11, 14] is the element of rank 5 (or $5th$ smallest element), namely 11.

1. We will start by thinking about how we would use Quicksort to find the median of an array. Draw the recursion tree generated by the call `QuickSort([10, 3, 5, 18, 1000, 2, 100, 11, 14])`. Assume that 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.



*Copyright Notice: UBC retains the rights to this document. You may not distribute this document without permission.

2. In your specific recursion tree above, mark the nodes in which the median (11) appears. (The first of these is the root.)

   SOLUTION: These are the ones with the blue 11 in them above.

3. Look at the **third** recursive call you marked. What is the rank of 11 in this array? How does this relate to 11's rank in the second recursive call, and why?

   SOLUTION: Note that we're looking at the node containing [**11**, 14]. This is the Lesser array of its parent, and the rank of 11 is 1. The rank did not change because to obtain the Lesser array from its parent array, only elements larger than 11 were removed, namely the pivot and the two elements in Greater.

4. Now, 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!

   SOLUTION:

   (a) In this array, what is the median? **18**

   (b) In this array, what is the rank of the median? The element 18 has rank 3, i.e., 18 is the 3rd smallest element.

   (c) In this array, what is the rank of 11? The element 11 has rank 1; it is the smallest element.

   (d) How does the rank of 11 in this array relate to the rank of 11 in the original array (at the root)? Why does this relationship hold?

   Note that we're looking at the node containing [**18**, 1000, 100, 11, 14]. The rank of 11 has changed: it is 1 here, but it was 5 at the root. Why? 11 ended up on the right side (in `Greater`). So, everything that was smaller than 11 in the original array (at the root) is no longer in the array [**18**, 1000, 100, 11, 14]. We discarded four elements (3, 5, 2 and the median 10) from the array at the root of the tree, so 11's rank has gone down by four.

5. If you're looking for the element of rank 42 (i.e., the 42*nd* smallest element) in an array of 100 elements, and `Lesser` has 41 elements, where is the element you're looking for?

   SOLUTION: If `Lesser` has 41 elements, then there are 41 elements smaller than the pivot. That makes the pivot the 42*nd* smallest element. In other words, if the size of `Lesser` is $k-1$, then the pivot has rank k. (This is what happened—with much smaller numbers—in the node where 11 is also the pivot.)

6. How could you determine **before** making `QuickSort`'s recursive calls whether the element of rank k is the pivot or appears in `Lesser` or `Greater`?

   SOLUTION: Putting together what we saw above:

   - If |Lesser| is equal to $k-1$, then the *k-th* smallest element is the pivot.
   - If |Lesser| is larger than $k-1$, then the pivot is larger than the *k-th* smallest element, which puts it in the `Lesser` group (and the left recursive call).
   - If `|Lesser|` is smaller than $k-1$, then the *k-th* smallest element is in `Greater` (and the right recursive call).

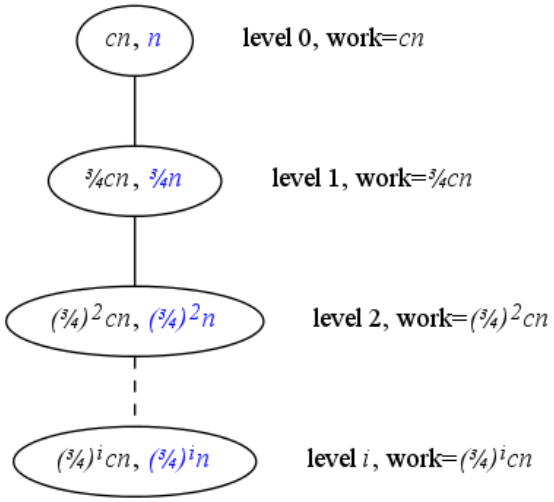
7. Modify the `QuickSort` algorithm so that it finds the element of rank k. Just cross out or change parts of the code below as you see fit. Change the function's name! Add a parameter! Feel the power!

SOLUTION: The key difference is that we no longer need the recursive calls on both sides, only on the side with the element we seek.

function QUICKSELECT($A[1..n]$, k) // returns the element of rank k in an array A of n distinct numbers, where $1 \leq k \leq n$
  if $n > 1$ **then**
    Choose pivot element $p = A[1]$
    Let Lesser be an array of all elements from A less than p
    Let Greater be an array of all elements from A greater than p
    if |Lesser| = k - 1 **then**
      return p
    else
      if |Lesser| > k - 1 **then** // all smaller elts are in Lesser; k is unchanged
        return QuickSelect(Lesser, k)
      else// |Lesser| $< k - 1$
        // subtract from k the number of smaller elts removed
        return QuickSelect(Greater, $k-$ |Lesser| -1)
  else
    return $A[1]$

8. Once again, suppose that the rank of the pivot in your median-finding algorithm on a problem of size n is always in the range between $\lceil \frac{n}{4} \rceil$ and $\lfloor \frac{3n}{4} \rfloor$. Draw the recursion tree that corresponds to the worst-case running time of the algorithm, and give a tight big-O bound in the algorithm's running time. Also, provide an asymptotic lower bound on the algorithm's running time.

SOLUTION: Here's the recursion 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 $4cn \in O(n)$.

Unlike the Quicksort algorithm, there's no need to analyze recursion tree structure to conclude that the algorithm takes $\Omega(n)$ time. Since the algorithm partitions the n elements of A before any recursive call, it must take $\Omega(n)$ time, and thus $\Theta(n)$ time.

In case you're curious, here's one way to analyse the above 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 $S = 4$.

While recursion trees provide a reliable way to solve pretty much all recurrences that we'll see in this class, it's good to know about alternative methods too. One popular method is the Master theorem, provided in a separate handout. Again, if we assume that we "get lucky" with the choice of pivot at every recursive call, so the size of the subproblem is at most $3n/4$, we can express the QuickSelect runtime using the following recurrence (ignoring floors and ceilings):

$$T_{QS}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ T_{QS}(3n/4) + cn & \text{otherwise} \end{cases}$$

This has the form of the Master theorem, where $a = 1$, $b = 4/3$, and $f(n) = cn$. Since $\log_b a = \log_{4/3} 1 = 0$ (the log of 1 is 0, no matter what the base is), we see that $f(n) = cn = \Omega(n^{(\log_b a + \epsilon)})$, where we can choose ϵ to be any positive number that is at most 1. So case 3 of the Master theorem applies, and we can conclude that $T_{QS}(n) = \Theta(n)$.