CSE 40113: Algorithms
Homework 1

You may complete this homework in groups of 3 or less students. Note that the integrity policy applies: your group should write up your own work, although you're welcome to work on the problems in a larger group. If you have any questions, please re-read both the homework guidelines and the academic integrity policy carefully, and then come discuss any questions or concerns with me.

Required Problems

1. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but it's a good idea to work through these for practice. Assume reasonable but nontrivial base cases if none are supplied. More exact solutions are better.

   (a) $A(n) = 4A(n/3) + n^2$

   Solution: We can use Master theorem or recursion trees here. If you use Master Theorem, then $a = 4$, $b = 3$, and $f(n) = O(n^2)$, so $d = 2$. Since comparing a and b^d gives $4 < 3^2 = 9$, we have $A(n) = \Theta(n^2)$. ■

   (b) $B(n) = 3B(3n/5) + n$

   Solution: Using the version of Master theorem from lecture: Here, $a = 3$, $b = 5/3$ (since $3n/5 = n/(5/3)$), and $f(n) = O(n)$, so $d = 1$. Since comparing a and b^d gives $3 > (5/3)^1$, we have $B(n) = \Theta(n^{\log_{5/3} 3})$. ■

   (c) $C(n) = 2C(n/2) + n \log n$

   Solution: Here, $f(n) = O(n \log n)$, which is not a polynomial. However, we can still use the version of Master theorem from lecture, or we can do recursion trees. I'll go with trees: The root (or level 0) of the recursion tree holds the value $n \log n$. There are 2 recursive calls of size $n/2$, so the two nodes on level 1 hold $n/2 \log(n/2)$ each. The next level down has 4 nodes, and each holds $n/4 \log(n/4)$. Continuing the pattern: level i has 2^i nodes, and the work in each node is $n/2^i \log(n/2^i)$. Since we divide by 2 each time, the overall depth of the tree is $\log_2 n$.

   We turn this into a sum:

$$\begin{aligned} C(n) &= \sum_{i=0}^{\log_2 n} (2^i)(n/2^i \log(n/2^i)) \\ &= \sum_{i=0}^{\log_2 n} (n/2 \log(n/2^i)) \\ &= n \cdot (\sum_{i=0}^{\log_2 n} \log(n/2^i)) \end{aligned}$$

Using a logarithm identity that $\log(a/b) = \log a - \log b$, we simplify the summation further: $\sum\limits_{i=0}^{\log_2 n} \log_2(n/2^i)) = \sum\limits_{i=0}^{\log_2 n} (\log_2 n - \log_2(2^i))$, and since $\log 2^i = i$ for i ranging from 0 to $\log_2 n$, this is just adding $1 + 2 + 3 + \ldots \log_2 n = \sum_{i=0}^{\log_2 n} i$. Plugging into our identity, this gives $(\log_2 n)(\log_2 n + 1)/2$ total.

Plugging this back into our summation and simplifying, the final answer is $C(n) = Theta(n(\log n)^2)$. ■

(d) $D(n) = 4D(n-1) + 2$

Solution: This one is not divide and conquer, so you need to go back to linear inhomogenous recurrences to solve, or else unplug the recurrence to see the patter. Using the characteristic equation method, for example, we get that the characteristic equation is $x - 4$, and since the inhomogeneous terms is a constant, $D(n) = \Theta(4^n)$. ■

(e) $E(n) = 3E(\lfloor n/3 \rfloor + 3) + 100n^3 - 6n$

Solution: ■

We can ignore floors and ceilings, as well as lower order terms of a polynomial. So here, using Master theorem, $a = 3$, $b = 3$, and $f(n) = \Theta(n^3)$ so $d = 3$. Then, $a = 3 < b^d = 9$, so we get $E(n) = \Theta(n^3)$.

2. Suppose you are given a sorted array of n distinct numbers that has been rotated k steps, for some unknown integer k between 1 and $n-1$. That is, you are given an array $A[1 \ldots n]$ such that some prefix $A[1 \ldots k]$ is sorted in increasing order, the corresponding suffix $A[k+1 \ldots n]$ is sorted in increasing order, and $A[n] < A[1]$. For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	1	3	4	5	7	8

(a) Describe and analyze an algorithm to compute the unknown integer k.

Solution: We solve this via a variant of binary search. However, instead of comparing a value against the middle element, we compare it to the next smallest value to see if it is the rotation. If not, we will check the first, middle, and last element. If $A[1] < A[m] < A[n]$ (where $m = \frac{n}{2}$), then it is sorted and there was no rotation. If $A[1] < A[m]$ and $A[m] > A[n]$, then we know the first half is in fact in sorted order, so the rotation number k must be greater than $m = n/2$. If not, then we must have $A[1] > A[m]$, and the rotation number k must be on less than $m = n/2$. In any case, we can recurse on just one side, and the half we recurse on satisfies the same property of being a rotated sorted list.

This gives the following algorithm, where I assume the array A is globally accessible, and our initial call is to FindRotation$(1, n)$:

```
FindRotation(low,high):
   if low = high
         return low
   m ← ⌊(low−high)/2⌋
   if $m > 0$ and $A[m] < A[m-1]$
         return m
   if $A[m] > A[high]$
         return FindRotation($m+1$,high)
   else
         return FindRotation(low,m)
```

(Note that you could also pass in A as in input parameter, if you prefer.)

Alternatively, you can code this using a while look that does the same recursion iteratively:

```
FindRotation($A[1..n]$):
   low ← 0, high ← $n$
   while low < high
         m ← ⌊(low−high)/2⌋
         if $m > 0$ and $A[m] < A[m-1]$
               return m
         if $A[m] > A[high]$low ← $m+1$
         else
               high ← $m-1$
   return low
```

The runtime is analogous to binary search, as in each iteration of the loop, we reduce the size by 1/2. That means the loop runs at most $\log_2 n$ times. In each iteration, it does $O(1)$ work to compare as well as to do calculations and updates. In total, this means the runtime is $O(\log_2 n)$.

Note that this can also be phrased recursively - I've done it this way to match ways you have seen it before (hopefully).

Proof of correctness: Induction on the size of array, which I'll say is high $-$ low (following our code).

Base case: If low = high, then we correctly exist the loop and return the value. Since it is the only value left to consider, it must be the rotation index.

Inductive Hypothesis: The algorithms works correctly for arrays of size less than n.

Consider an array of size n. Since k (the index of rotation) exists, we know if m is the value for k, we will return it correctly in the first if statement of the loop. If it is not, then it must lie in either the first or second half, and we can detect this by comparing $A[m]$ to the first and last elements. Since it is rotated, only one pair is in sorted order, and the inversion must be in the other half. We reset our loop boundaries appropriately to search that half, and by our inductive hypothesis, we know that the algorithm works correctly as the array has gotten smaller by a factor of 2, and so low − high will be $< n$.

■

(b) Describe and analyze an algorithm to determine if the given array contains a given number x.

Solution: My solution will use part (a), as well as binary search, which I have coded as follows (and again assuming a global array A, although you could also pass it as an input parameter):

```
BinarySearch(x,low,high):
  if low $>$ high
      return -1
  m $\leftarrow \lfloor$(low$-$high)/2$\rfloor$
  if $A[m] = k$
      return m
  if $A[m] < x]$
      return BinarySearch($m + 1$,high)
  else
      return BinarySearch(low,m-1)
```

Then, our solution is as follows, again assuming a globally accessible array A (or otherwise just add A as an additional input parameter in every call) as well as a target value x:

```
SearchRotated(A[1..n],x):
  $k \leftarrow$ FindRotation(1,n)
  $l \leftarrow$ BinarySearch(x,1,k-1)
  $r \leftarrow$ BinarySearch(x,k,n)
  if $l \neq -1$
      return l
  else if $r \neq -1$
      return r
  else
      return -1
```

The runtime is $O(\log n)$ total, as part (a) takes $O(\log n)$ time as well as two calls to binary search, which my discrete math/data structures book has already studied.

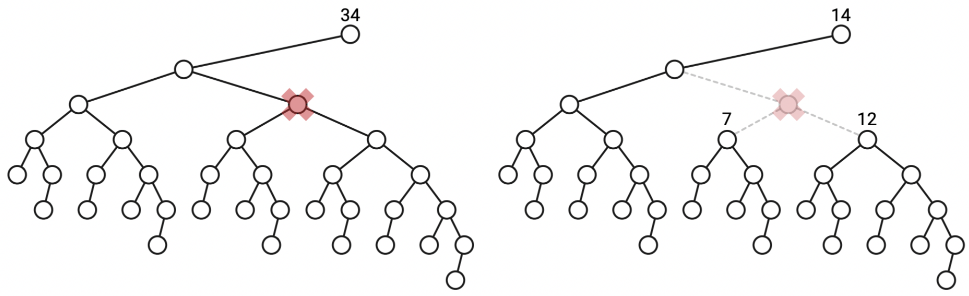
Correctness: If the algorithm from part a is correct, we have the value k such that both $A[1..k-1]$ and $A[k..n]$ are sorted. I can cite my discrete math or data structure text again for why binary search will work in a sorted array. If the value is present, it must be in one of the two sorted sections, so my code will find it.

Note that we could modify binary search directly to search in a rotated array, or (equivalently) modify our algorithm from part a to search as it recurses, if that is preferable. In this case, you'd prove correctness via induction, and the runtime would be the same recurrence as part (a) or as binary search, just with a few extra comparisons. Here is the code:

```
SearchRotated($A[0..n-1], x$)
  left $\leftarrow 0$
  right $\leftarrow n-1$
  while left $<$ right
      mid $\leftarrow$ left + (right $-$ left) //2
      if A[left] $\leq$ A[mid] and A[left] $\leq x \leq$ A[mid]
          right $\leftarrow$ mid
      else if A[mid] $<$ A[right] and ($x >$ A[right] or $x \leq$ A[mid])
          right $\leftarrow$ mid
      else
          left $\leftarrow$ mid + 1
      return left if A[left] = x else $-1$
```

■

3. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a central vertex if each of these smaller trees has at most n/2 vertices. See below for a 34 node binary trees, where deleting the central vertex leaves 3 subtrees, with 14, 7, and 12 nodes each.



Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree. [Hint: It might be helpful to first prove that every tree has a central vertex, then chase it down - recursively, if at all possible!]

Solution: Claim: A central vertex must exist.

Proof: Consider the root of the tree. If it is central, we are done. If not, then exactly one of the left and right subtrees must have size $> n/2$, as they cannot both have size greater if we only have n nodes; the other has $< n/2$ nodes. We know the central vertex cannot be in the smaller subtree of size k, since any cut inside of it will have the larger subtree on the parent side of its cut, and hence will be too large.

Let v be the root of the larger subtree. If v is central, we are done. If not, removing v leaves 3 smaller subtrees: the parent component T_1, and the two children which I'll denote by T_2 and T_3. Let $|T_i|$ be the number of vertices in subtree T_i, so that $1 + |T_1| + |T_2| + |T_3| = n$. We know that the parent cut is $< n/2$ in size, since we moved from the root of the tree to the larger subtree, so $|T_1| < n/2$. Since v is not central, we also know that one of the two subtrees has $> n/2$ vertices; without loss of generality, suppose this is T_2.

We again move into this subtree, and update v be the root of this subtree. We have the same situation: the parent cut must be smaller than $n/2$, and if v is not central, then one of the subtrees must be of size > 2. But, the larger subtree's size must have decreased, as we moved down one vertex in the tree. So, as we iterate this process, we know the larger subtree's size is decreasing, and eventually such a path terminates at a leaf (in which case the subtree sizes are 0). So at some point along this path to the leaf, we hit a vertex v where both child trees have $< n/2$ nodes, and the parent will still be $< n/2$ because we chose to move down to v. This is our central vertex. □

This leads to an immediate algorithm: First, use a postorder traversal to calculate the size of every rooted subtree. This takes $O(n)$ time, and then sizes can be stored inside each node. Then, starting at the root, use these sizes to check if the node is central, moving down in the tree to the larger subtree until we find such a vertex. Since we traverse a root to leaf path, this takes worst case $O(n)$ time as well.

The runtime of this is $\Theta(n)$ total, as the postorder traversal takes $\Theta(n)$ time (and space) to store the size of every rooted subtree, and then the main algorithm traverses a root to leaf path, which is $O(n)$ time in the worst case. (Note that these trees are not balanced, so you cannot claim $O(\log n)$ as in balanced binary search trees.)

■

4. Sample solved problem:

   You are interested in analyzing some hard-to-obtain data from two separate databases, which I'll call A and B. Each database contains n numerical values, so that there are $2n$ total, and you may assume that no two are the same. You'd like to determine the median value of this set of $2n$ values, which we define to be the n^{th} value.

   However, the situation is complicated by the fact that you can only access these values through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

   Given an algorithm that finds the median value using at most $O(\log n)$ queries. Be sure to specify the algorithm, the analysis for the number of queries, and a justification (i.e. a proof) that your algorithm returns the median value. (To keep things simpler, you are welcome to assume that n is a power of 2.)

   Solution:

   I'll index the elements of each database from 1 to n, so that query(A, i) looks up the i^{th} entry in the database A. Suppose we first query each database for its median—that is, we query each one for element $n/2$. Let m_1 be the median from database 1 and let m_2 be the median from database 2. Suppose, without loss of generality, that $m_1 < m_2$. We know that $n/2$ elements in database 1 are less than or equal to m_1, and therefore must also be less than m_2. (We don't yet know how the remaining $n/2$ elements in database 1 are ordered with respect to m_2). Similarly, we know that $n/2$ elements in database 2 are greater than m_2 and therefore also greater than m_1 (but we don't know how the remaining $n/2$ elements in database 2 are ordered with respect to m_1.

   At this point, we can conclude something: the median of all $2n$ elements must be in the largest half of database 1, or in the smallest half of database 2. At this point, we can chop of databases in half, and recurse!

   Pseudocode:

```
FindMedian($A, a_{\min}, a_{\max}, B, b_{min}, b_{max}$):
    if $a_{\max} == a_{\min}$
        $m_a \leftarrow$ query$(A, 1)$
        $m_b \leftarrow$ query$(B, q)$
        return min$(m_a, m_b)$
    $m_a \leftarrow$ query$(A, (a_{\max} + a_{\min} - 1)/2)$
    $m_b \leftarrow$ query$(B, (b_{\max} + b_{\min} - 1)/2)$
    if $m_a < m_b$
        return FindMedian$(A, m_a + 1, a_{\max}, B, b_{\min}, m_b)$
    else
        return FindMedian$(A, a_{\min}, m_a, B, m_b + 1, b_{\max})$
```

   Our initial call is then to FindMedian$(A, 1, n, B, 1, n)$; from there on out, $a_{\min}$ and $a_{\max}$ (as well as the similar values in B) represent the subdatabase that we are working with in that recursive call.

   Proof of correctness: Induction on n:

   Base case: There is a single element in each database, so the median is (by definition) the smaller of the two. This is done in the first if statement of our pseudocode.

Inductive Hypothesis: For any value $< n$ of items in each database, our algorithm correctly returns the median of all elements in the two databases.

Inductive Step: Consider a database of $2n$ elements (so exactly n in each individual one). We first find the two medians, m_a and m_2. As described above, if we have $m_a < m_b$, then elements in the first half of database A are less than both medians, and elements in the second half of database B are greater than both. Since there are $n/2$ elements in each of these halves, we know that the overall median cannot be in those sections: if it were, we would have a contradiction, because there are $n+1$ elements greater than m_a, and $n+1$ elements less than m_b. Since the median is the n^{th} smallest, this is impossible.

Since we discard the same number of elements from A and B, the median overall will be equal to the median of the smaller databases of size n each. By our inductive hypothesis, our algorithm will correctly find the median of these smaller databases, and thus will return the overall correct value.

Runtime: Our base case makes two queries and a comparison, which is total time $O(1)$.

We then can construct the recurrence as follows: Our pseudocode makes 2 queries, then does a comparison, then calculates two additions and two divisions, before making a recursive call on a database which is half as big. The resulting recurrence is

$$T(n) = T(n/2) + O(1)$$

Plugging into Master theorem, we get $T(n) = \Theta(\log n)$.