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

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

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

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

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

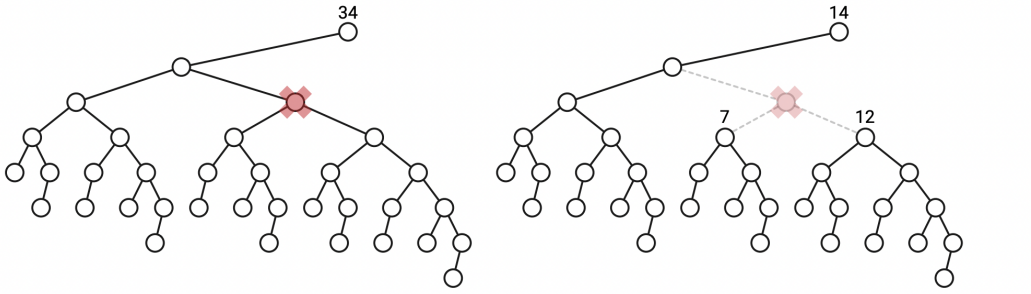
2. Suppose you are given a sorted array of n distinct numbers that has been rotated k steps, for some unknown integer k between 1 and $n-1$. That is, you are given an array $A[1 \ldots n]$ such that some prefix $A[1 \ldots k]$ is sorted in increasing order, the corresponding suffix $A[k+1 \ldots n]$ 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.

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

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!]

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 ← query(A, 1)
      m_b ← query(B, q)
      return min(m_a, m_b)
  m_a ← query(A, (a_max + a_min − 1)/2)
  m_b ← 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 $\leq n$, our algorithm correctly returns the median of all elements in the two databases.

Inductive Step: Consider a database of $2n$ elements. 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 $T(n) = \Theta(\log n)$.