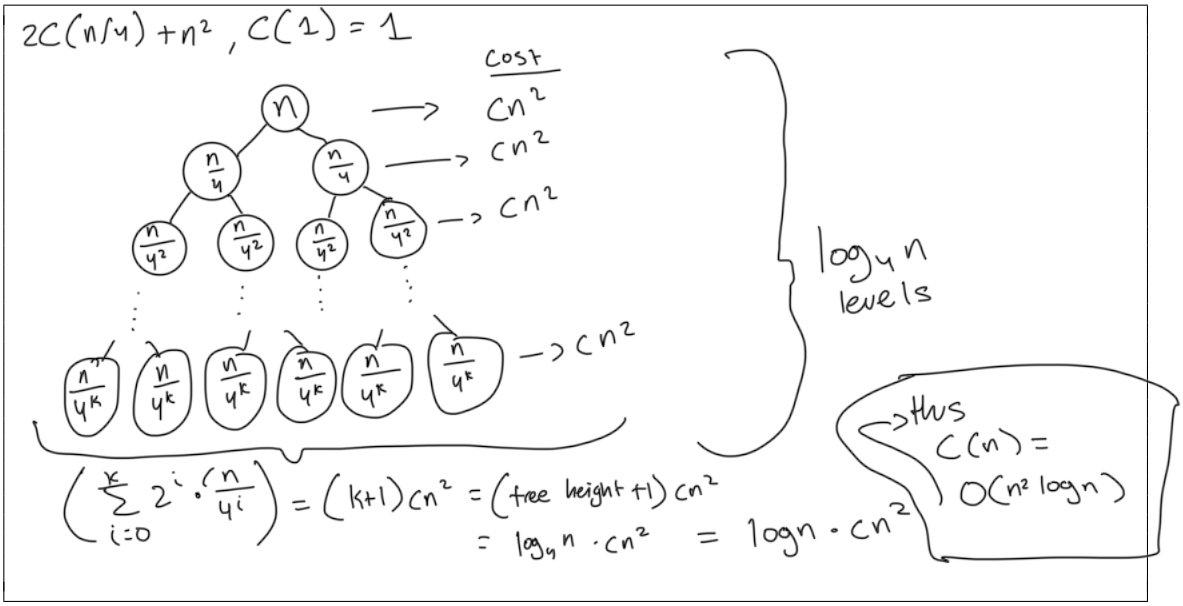
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

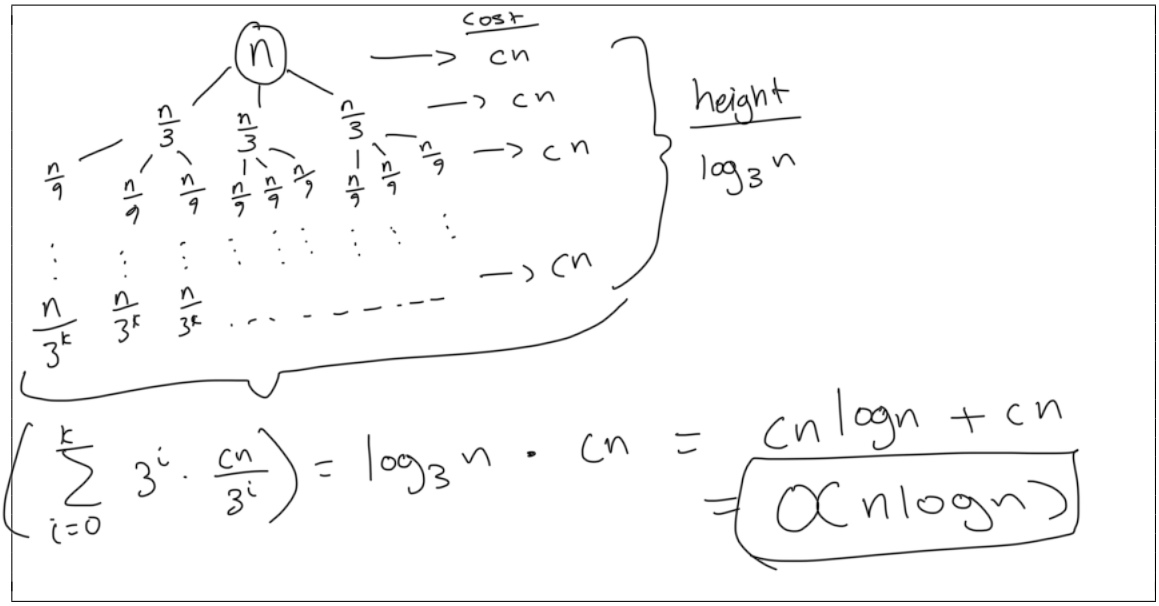
Name: Mahir Khan Wisc id: mhkhan9@wisc.edu

Divide and Conquer

1. *Erickson, Jeff. Algorithms (p.49, q. 6).* Use recursion trees to solve each of the following recurrences.
   (a) $C(n) = 2C(n/4) + n^2$; $C(1) = 1$.



   (b) $E(n) = 3E(n/3) + n$; $E(1) = 1$.



2. *Kleinberg, Jon. Algorithm Design (p. 246, q. 1).* You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the nth smallest value.

   However, the only way you can access these values is 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 kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

   (a) Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Algorithm: Database Median

Input: Two lists A and B (representing the 2 databases) of size n, with distinct elements in each list.

Output: The median element of both A and B, m

If n == 1, return min(A[0], B[0]) // Base case

k := $\lceil \frac{n}{2} \rceil$ // kth smallest element = median of both A and B.

a := A[k]
b := B[k]

if a < b then:
    Database Median (A[k+1 ... n], B[0 ... k], k) // Basically, if a < b, then B[k] must be greater than the first half of A. Furthermore, this means B[k+1 ... n] greater than combined median, thus remove back half of B. Now both lists are size k.

else if a > b then:
    Database Median (A[0 ... k], B[k+1 ... n], k) // same as above, but opposite.

else if a == b then:
    return a // this median already found.

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

Recurrence $T(n) \leq \underbrace{T(\frac{n}{2})}_{\text{the one recursive call in if statements}} + \underbrace{c}_{\text{Finding k or other constant operations}}$, $T(1) = c$

$T(n) \leq T(\frac{n}{2}) + c$
$\leq T(T(\frac{n}{4}) + c) + c$
$\leq T(T(T(\frac{n}{8}) + c) + c) + c$
$\leq T(\frac{n}{2^k}) + kc$

$1 = \frac{n}{2^k}$
$\Leftrightarrow 2^k = n$
$\Leftrightarrow k = \log_2(n)$

$\Rightarrow T(1) + c \cdot \log n = c + c \cdot \log n \leq O(\log n)$

(c) Prove correctness of your algorithm in part (a).

Correctness Proof

Base Case: Size of database A and B are both 1, thus the median must be the same in both cases, thus base case holds.

Inductive Step: Assume that for databases A and B of size $n = j$, that all rewrsive calls on $j/2$ return the correct median value found in each respective database. Thus, we know that a_j and b_j are valid medians, and so if $a_j > b_j$ or vice versa, then another rewrsive call will be made to half the respective halves of each database until a median is found, or if $a == b$, then the program returns a, which is the correct median.

3. *Kleinberg, Jon. Algorithm Design (p. 246, q. 2).* Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, ..., a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, this measure is very sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$.

(a) Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

```
Significant Inversions ( )
Input: A list A of a sequence of n numbers {a_0, a_1, ... a_n}
output: A sorted list and count "c" of significant inversions.

If |A| == 1, then return (A, 0)
(A_1, c_1) = Significant Inversions (A[0, ... n/2])   // Front and Back half of list A, where c_1 and c_2
(A_2, c_2) = Significant Inversions (A[n/2+1, ... n])   hold numbers of inversions in each half.
(A, c) = MergeCount (A_1, A_2)
return (A, c + c_1 + c_2)

MergeCount( ) Input: 2 lists with n numbers, A and B
              output: merged list and number of significant inversions.
Initialize S as an empty list, and c := 0
while either A or B are not empty, do:
    Pop and append min {A[0], 2 · B[0]}   // 2 · B[0] to check for significant inversions
      If appended item is from B then:
          c := c + |A|
    end
return (S, c)
```

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

$$
\begin{aligned}
T(n) &\le 2T\left(\tfrac{n}{2}\right) + cn \\
&\le 2\left(2T\left(\tfrac{n}{4}\right) + c\tfrac{n}{2}\right) + cn \\
&\le 2\left(2\left(2T\left(\tfrac{n}{8}\right) + c\tfrac{n}{2^2}\right) + c\tfrac{n}{2}\right) + cn \\
&\vdots \\
&\le 2^k T\left(\tfrac{n}{2^k}\right) + kcn \qquad \longrightarrow 1 = \tfrac{1}{2^k} \\
&= nT(1) + cn\log n \qquad\qquad 2^k = n \\
&= cn + cn\log n \qquad\qquad k = \log_2(n) \\
&\le O(n\log n)
\end{aligned}
$$

(c) Prove correctness of your algorithm in part (a).

Correctness

Base Case: Size of list = 1. In this case, the program returns 0 since a list of size 1 has no significant inversions possible.

Inductive Step: Assume that the recursive calls on to both halves of the total list return the correct number of significant inversions, as well as returning sorted lists. Then, MergeCount is called on the 2 returned sorted list. In MergeCount, both lists have elements popped iteratively from A and B, and the minimum is put into sorted list S. Furthermore, if $\min(A[0], 2\cdot B[0])$ returns $2\cdot B[0]$ as the minimum, then it must mean that $A[0] > 2\cdot B[0]$. Thus, this is a significant inversion, and the size of list A is added to c, since if $2\cdot B[0] < A[0]$, then $2\cdot B[0] < A[1]$, since A is a sorted list.

Afterwards, the elements continue being popped until both lists are empty, and continuing the process.

Thus, mergecount() returns the correctly sorted list + correct # of significant inversions!

4. *Kleinberg, Jon. Algorithm Design (p. 246, q. 3).* You're consulting for a bank that's concerned about fraud detection. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud.

   It's difficult to read the account number off a bank card directly, but the bank has an "equivalence tester" that takes two bank cards and determines whether they correspond to the same account.

   Their question is the following: among the collection of n cards, is there a set of more than $\frac{n}{2}$ of them that all correspond to the same account? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester.

   (a) Give an algorithm to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

```
FraudDetection()
Input: A list A of n Bank Cards that have been confiscated for fraud
Output: A value e that is equivalent to n/2 or more cards in List A.

If |A| = 1, then return True

Left := FraudDetection(Front half of A, n/2)
Right := FraudDetection(Back half of A, n/2)

for cards in A:                                          } left
    left_count = equivalenceTester(A[card], Left)        } side
if left_count > n/2:
    return Left
for cards in A:                                          } right
    right_count := equivalenceTester(A[card], Right)     } side
if right_count > n/2:
    return Right

return 0
```

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

$$T(n) \le 2T(n/2) + 2n, \quad T(1) = 2$$
$$\le 2(2T(n/4)+2n)+2n$$
$$\le 2(2(2T(n/8)+2n)+2n)+2n$$
$$\vdots$$
$$\le 2^k T\left(\frac{n}{2^k}\right) + k \cdot 2n$$

$$1 = \frac{1}{2^k}$$
$$2^k = n$$
$$k = \log_2(n)$$

$$nT(1) + \log n \cdot 2n$$
$$\hookrightarrow 2n + 2n\log n \le O(n \log n)$$

(c) Prove correctness of your algorithm in part (a).

Base Case: When there is only 1 card, it is the majority card by definition, thus the base case holds.

Inductive Step: Assume that in any set of k<n cards, the algorithm correctly finds the majority account in the set of cards, showing the cards come from the same account.

Now lets say we have $n = k$ cards.

The algorithm divides the n cards into $n/2$ cards and rewsively calls itself. (twice) By the inductive hypothesis above, the rewsive call returns a card from the majority account.

The algorithm then runs the equivalence tester to find out which of the 2 majority account cards in the $\frac{n}{2}$ lists is the majority card in the full set of n cards.

↳ There are 3 possible cases:

- The same account is the majority owner of cards in both halves. Thus, this account is the majority account in the full set of n cards since it appears $\frac{n}{2}$ times, and is returned
- Different accounts are the majority owner of cards in the two halves, and one of them appears more than $n/2$ times in the full set of cards. Then, the account with more appearances is returned.
- Different accounts have majority cards in the two halves, and neither of them appears more than $n/2$ times in the full set of n cards. Thus, there is no majority account, and the algorithm returns 0.

Thus, through induction, we have proven correctness