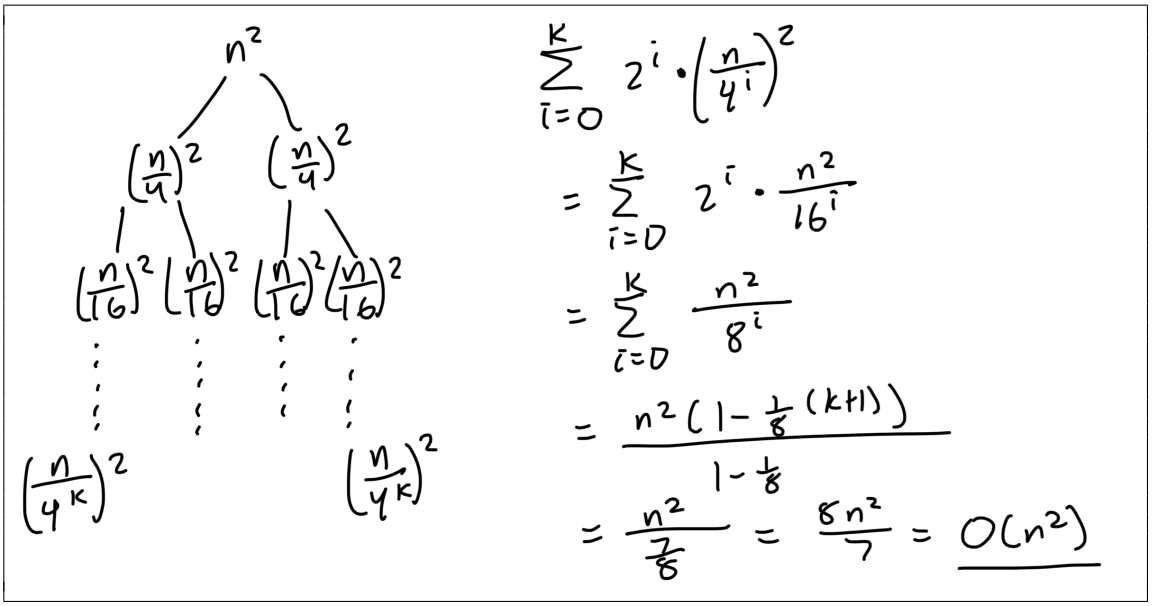
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

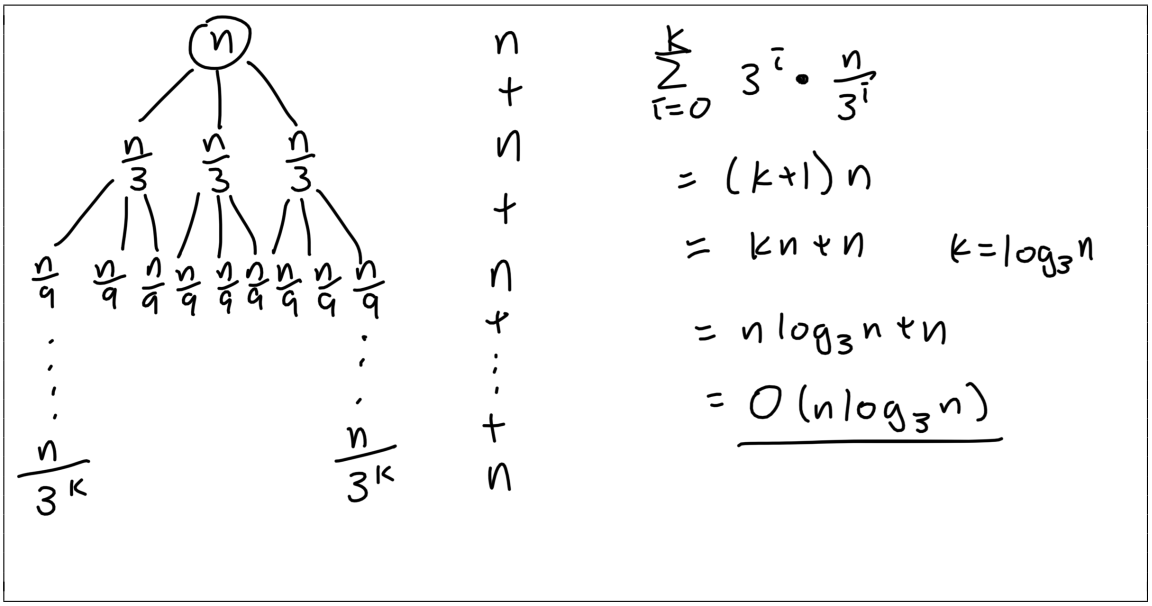
Name: Zinnia Nre Wisc id: 908 319 4044

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.

```
Find Med  Input: data1, l1, h1, data2, l2, h2

If (l1 = h1):
    m1 = query(data1, l1)
    m2 = query(data2, l2)
    return min(m1, m2)
m1 = query(data1, h1 + l1/2)
m2 = query(data2, h2 + l2/2)

If m1 < m2: (use top half of data1, bottom half of data2)
    return Find Med(data1, h1 + l1/2, h1, data2, l2,
                                          h2 + l2/2 - 1)
else:
    return Find Med(data1, l1, h1 + l1/2 - 1, data2, h2 + l2/2,
                                                      h1)
```

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

2 recursive calls logn queries

$\frac{n}{2}$ size

O(1) runtime each call, only queries

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

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

Soundness: Proof by induction on size of database

Base case: when $l_1 = h_1$, the database only has 1 entry that can be queried, so that is the median.

Inductive Hypothesis: The median of both databases are found when the size of queriable entries is $\frac{k}{2}$.

For database of size k:

Median of both databases will be between m_1 and m_2. So need to recurse on the middle area, which will be half of each database. By IH, the correct median of half the database is returned.

Termination:

Since each recursive call uses half the database, eventually we will decrease until the base case of 1. ∎

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.

Still using modified MergeSort

modified MergeCount

```
while A or B is not empty do
    Pop and append min(front A, front B)
    if append from B and frontA < 2·frontB
        count = count + |A|
    end
end
return count
```

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

2 recursive calls

$\frac{n}{2}$ size of recursive call

$O(n)$ runtime of mergecount

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

$= O(n \log n)$

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

Correctness: Proof by Induction

Base Case: size = 1, only one element, return 0 since there are no inversions.

Induction Hypothesis: The correct count is returned for front and back half of the array. (Recursion works).

Then in count sort, array A is front half where values less than back half (array B) go. If an array B element is chosen, that means it was less than all the rest of the elements of A, but located behind them, thus there is the remaining elements of A number of inversions due to that one element.

Then since by IH, the inversions in each half were already counted for, the total inversions are the two halves plus the new merge.

Termination: size of array is halved each time, so progress is made towards the base case.

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.

```
Equals
input: array of cards A[1...n]
output: subset of equivalent cards
if n==1: return card
if n==2:
    if equivalence tester(A[1], A[2]) == true:
        return A[1]
M1 = front half of A
M2 = back half of A
If Equals(M1) returns a card
    compare to rest of cards in A
    if is majority: return that card
If Equals(M2) returns a card
    compare to rest of cards in A
    if is majority: return that card.
```

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

2 recursive calls

$\frac{n}{2}$ size of recursive call

$2n$ = runtime of each call

$$T(n) = 2T(\tfrac{n}{2}) + 2n = 2T(\tfrac{n}{2}) + O(n) = O(n \log n)$$

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

Base Case: $n = 1$, the card is returned, as it is equal to more than $n/2$ cards in its set since it is the only card

$n = 2$, if both are equal, then it is equal to more than $n/2$ cards in its set

Induction Hypothesis: Equals returns the correct output for $n = \frac{k}{2}$

When array is size n, and a card is returned from one or both of its halves, it is checked again through the full A array. Only if it matches $> \frac{n}{2}$ there is it returned. If neither half's returned card is a majority, then there is no majority in the array, since if there is a majority in the whole array, it must also be a majority in one of the halves.

Termination:

The function is called recursively with half the input size each time and will eventually reach the base cases of 1 or 2. ∎

5. Implement the optimal algorithm for inversion counting in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n \log n)$ time, where n is the number of elements in the ranking.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of elements in the ranking. A sample input is the following:

```
2
5
5 4 3 2 1
4
1 5 9 8
```

   The sample input has two instances. The first instance has 5 elements and the second has 4. For each instance, your program should output the number of inversions on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

```
10
1
```