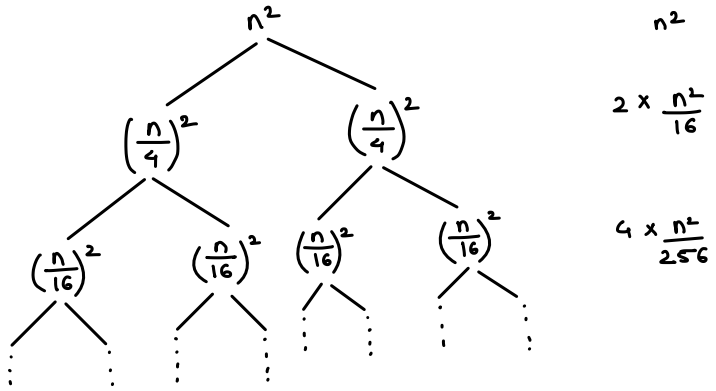
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Riya Kore Wisc id: rykore

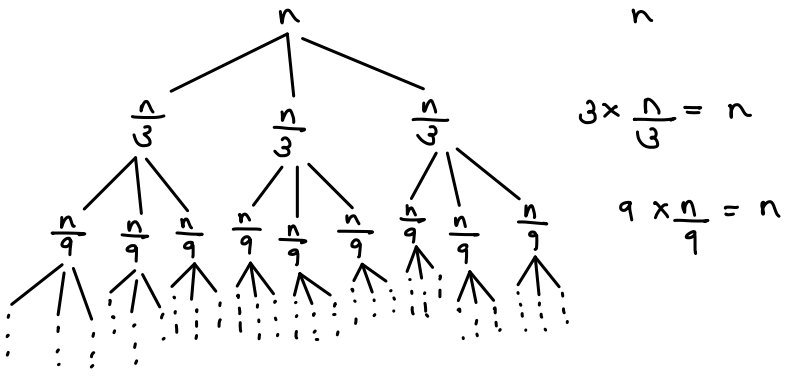
Divide and Conquer

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



The runtime complexity is: $O(n^2)$

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



The runtime complexity is: $O(n \log n)$

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.

Solution:

Let the be two databases P and Q.

Let's keep track of the lower bounds in P and Q. Let's call it P_{lower} and Q_{lower}. Initially, both of them are 1. Similarly, for the upper bonds for P and Q, we have P_{upper} and Q_{upper} initialized to n.

Let p and q be the results of querying P and Q respectively.

Base case: If $P_{upper} \leq P_{lower}$, the median is $\min(p, q)$.

Recursive step:

Query P on $k_1 = \frac{P_{upper} + P_{lower}}{2}$ and Q on $k_2 = \frac{Q_{upper} + Q_{lower}}{2}$

(assuming that $P_{upper}, P_{lower}, Q_{upper}$ and Q_{lower} are integers.)

If $p > q$, set $P_{upper} = k_1$ and $Q_{lower} = k_2 + 1$.

If $p < q$, set $Q_{upper} = k_2$ and $P_{lower} = k_1 + 1$.

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

In each recursive call, the range $P[P_{lower}, \ldots, P_{upper}] \cup Q[Q_{lower}, \ldots Q_{upper}]$ of values to find is cut in half (floors and ceilings, which don't affect the asymptotic runtime), if $a > b$:

$$P\left[P_{lower}, \ldots, \left\lceil \frac{P_{lower}, P_{upper}}{2} \right\rceil\right] \cup Q\left[\left\lfloor \frac{Q_{lower} + Q_{upper}}{2} \right\rfloor, \ldots, Q_{upper}\right] \quad -①$$

If $a < b$:

$$P\left[\left\lfloor \frac{P_{lower} + P_{upper}}{2} \right\rfloor, \ldots, P_{upper}\right] \cup Q\left[Q_{lower}, \ldots, \left\lceil \frac{Q_{lower} + Q_{upper}}{2} \right\rceil\right]$$

So, the recurrence relation is $T(n) = T(n/2) + O(1)$; $T(1) = 1$. This yeilds a recurrence tree with $\log(n)$ layers and $O(1)$ work at each layer, hence the solution is $O(\log n)$.

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

Let x be the desired nth smallest value, and $R_p = P[p_{lower}, \dots, p_{upper}]$ and $R_q = Q[q_{lower}, \dots, q_{upper}]$. We prove this statement (If the median of $R_p \cup R_q$ is x, then the algorithm returns x) by strong induction on the size: $p_{upper} - p_{lower} + 1 + q_{upper} - q_{lower} + 1$ of the remaining search range $R_p \cup R_q$.

Initially, $p_{lower} = q_{lower} = 1$ and $p_{upper} = q_{upper} = n$, so the statement confirms that our algorithm is correct. Our statement holds true for the base case. Now, lets assume that our statement holds for all ranges smaller than $R_p \cup R_q$. Observe that p and q are the medians of R_p and R_q, respectively.

Suppose $p > q$, then $x \le p$. Otherwise, if $x > p$, then x is larger than more than half the elements of R_p, and since, $x > p > q$, x is also larger than half the elements of R_q. This contradicts the assumption that x is the median of $R_p \cup R_q$. Similarly, $x \ge q$, in this case, the algorithm will have the range $R_p' \cup R_q'$ in (1). This removes $\left\lceil \frac{p_{upper} - p_{lower}}{2} \right\rceil$ elements smaller than p, hence smaller than x, and $\left\lceil \frac{q_{upper} - q_{lower}}{2} \right\rceil$ elements larger than q, hence larger than x. By symmetry of the P and Q indices, $\left\lceil \frac{p_{upper} - p_{lower}}{2} \right\rceil = \left\lceil \frac{q_{upper} - q_{lower}}{2} \right\rceil$, so to make $R_p' \cup R_q'$, we have removed from $R_p \cup R_q$ an equal number of elements larger than and smaller than x, which by assumption is the median of $R_p \cup R_q$. Hence, x is also the median of $R_p' \cup R_q'$. By induction, the statement also holds for the smaller range $R_p' \cup R_q'$, and x is the median of $R_p' \cup R_q'$, so using that statement, the algorithm returns x. A symmetric proof also applies for the case $p < q$.

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.

Solution:

Return value: The number of significant inversions and sorted version of input sequence.

Divide: Split the input sequence into two halves and recursively calculate the number of significant inversions in each half as well as the sorted version of the input sequence.

Merge: Since we have the result of inversions in the left and right halves, lets calculate the inversions between these halves. Let the left and right half be L and R respectively. Let i and j be initialized to the size of L and R respectively. Initialize result N to be the sum of the counts of inversions exclusively on the left and right halves. If $L[i] \le 2R[j]$, then, if $j > 1$, $j -= 1$, and if $j = 1$, then stop. If $L[i] > 2R[j]$, $N += j$. Then, if $i > 1$, $i -= 1$, and, if $i = 1$, then stop. Finally, use merge sort algorithm to generate sorted sequence to return. Return N and sorted sequence.

Base case: If the length of the input sequence is 1, just return 0 inversions and the input sequence

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

In the divide step, the algorithm makes two recursive calls, each on half of the array. $2T(n/2)$. The loop in the Merge step goes through the left and right half exactly once, then runs the merge sort algorithm, each of which takes linear time in n.

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

This is identical to the recurrence for the merge sort algorithm, and has solution $O(n \log n)$.

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

On input sequences of length K, the algorithm returns the correct number of inversions and the correctly sorted sequence

We prove the following statement by strong induction on length K of the input sequence.

for k=n, the statement confirms that our algorithm is correct.

Base case: For K=1, the algorithm correctly assess that a sequence of length 1 has no inversions and is already sorted.

Inductive hypothesis: Let's assume this statement holds for sequences of length less than K.

Inductive step: The Divide step recurses on the first and second half of the sequence, each of which have length less than K, so, by using the inductive hypothesis, the algorithm obtains a correct value N and the two half-sequences are properly sorted. This means that the merge step is correct. If $L[i] \leq 2R[j]$, then (i,j) don't form a significant inversion. If $L[i] < 2R[j]$ then (i, j) are a significant inversion. It follows that since the right half is sorted in increasing order, (i, l) are a significant inversion for every $1 \leq l \leq j$. Hence, the algorithm adds j significant inversions. As all the previous j satisfied $L[i] \leq 2R[j]$, there are no more significant inversions involving i, so this algorithm decrements i. Since the left half is sorted, we don't miss any significant inversions by maintaining the same j value while decrementing i. Hence, the merge steps correctly computes the number of significant inversions. In this algorithm, the Merge sort merge step is correct, so the statement holds.

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.

Return values: Card that represents majority element in a group.

Divide: Let's split the cards into two halves and using the recursive approach, find the majority element of each half.

Merge: As the majority element in a subproblem should also be the majority element in at least one of its halves, let's consider the majority element, if we have any, returned by each of the half.
If both sides have the same majority element, return this as the majority element.
If neither of the sides has a majority element, return 'none'.
Else, let p and q be the majority elements of the left and right halves respectively. Compare p and q with every element in both halves to obtain numbers n_p and n_q of elements matching p and q in a combined array. If $n_p > \frac{n}{2}$, return p.
Similarly, if $n_q > \frac{n}{2}$, return q, and if both $n_p, n_q \leq \frac{n}{2}$, return 'none'.

Base case: Only one card to compare. You don't have to invoke the equivalence tester, but this card is the majority element of its group (size = 1).

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

In the Divide step, this algorithm makes two recursive calls, each on half of the array. $2T(n/2)$. The Merge step, in its worst case scenario, compares every element in the array to both p and q, which requires 2n steps. So, the recurrence relation is:

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

This is the same as the recurrence for Merge sort, and has solution $O(n \log n)$.

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

Statement: On the input of K cards, the algorithm correctly determines whether a majority exists and, if so, returns a member of the majority.

To prove this statement, let's use strong induction on the length K of the input sequence.

For K=n, the statement confirms that our algorithm is correct.

Base case: For K=1, the algorithm correctly returns the only card that represents a majority.

Inductive hypothesis: Let's assume this statement holds for all sets of cards fewer than K cards.

Inductive step: The Divide step recurses on the first and second half of the cards, each of which is a set of less than K cards, so by our inductive hypothesis, the recursive call correctly determines the majority. This means that the Merge step is correct. Let p, q, n_p, n_q be defined as the majority elements and the numbers in part (a).

If $p = q$, and $n_p, n_q > \frac{(n/2)}{2}$ (both sides have the same majority element), then $n_p + n_q > \frac{n}{2}$, this confirms that $p = q$ is a majority element.

If neither side has a majority, then for any x in the left half and y in the right half, $n_x + n_y \leq \frac{(n/2)}{2} + \frac{(n/2)}{2} = \frac{n}{2}$, so the combined list has no majority.

If the sides have different majority elements p and q, then by similar reasoning to case 2, no element other than p or q can form a majority, so the algorithm behaves correctly.

This algorithm returns the majority element, so the statement holds.

5. **Inversion Counting:**
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 list.

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 list.

Note that the results of some of the test cases may not fit in a 32-bit integer.

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
```