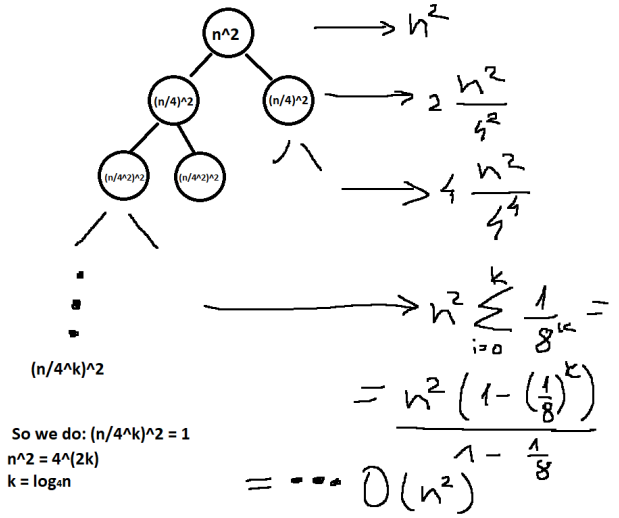
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Divide and Conquer

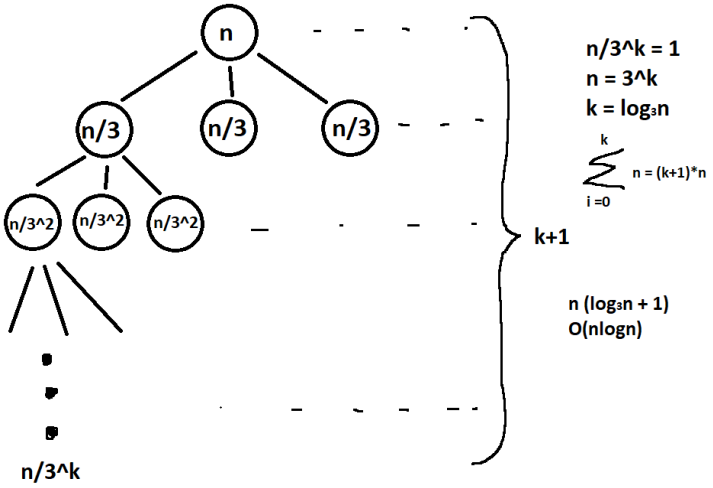
1. *Erickson, Jeff. Algorithms (p.49, q. 6).* Use recursion trees to solve each of the following recurrences.
   (a) $C(n) = 2C(n/4) + n^2$; $C(1) = 1$.

Solution:



$= \cdots O(n^2)$

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



Solution:

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:

We have 2 arrays A[n], B[n} of n elements sorted in ascending order and no two equal values. We determine the median for both arrays A[k] and B[k], k=⌈ n/2 ⌉
If A[k]<B[k] , this means that B[k] is greater than the first k elements of A (as also k-1 elements of B) so it is at least on position 2k >=n element. So the median can't be element in the second half of the array B[i], i=k+1,..,n.
Similarly first k-1 elements of A, are less than A[i], i=k,...n and less than B[i] i=k,...n, so median can't be element in the first half of the array A[i], i=1,...k-1. The process is repeated with reduced arrays.
Call with median(n,0,0)

```
median(n, a, b)
// Input 2 arrays A[n], B[n] of n elements sorted in ascending order and no two equal values
//.. output median
k=⌈ n/2 ⌉
If n=1  return (A[a+k}+B[b+k])/2
If n=2 return (max(A[a+k],B[b+k])+min(A[a+k+1],B(b+k+1]))/2
if A[a+k]<B[b+k]
 then return median(k,a+⌊1/2n⌋,b)  // upper half of A
 else return median(k,a, ⌊b+1/2n⌋) // Lowe half of B
```

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

Solution:

b) To analyze the runtime of this algorithm, let T(n) be the worst-case number of queries required to find the median of two sets of n elements. In each iteration of the algorithm, we perform a single query to each database, so the total number of queries is equal to the number of iterations. Each iteration reduces the size of the range of indices we are interested in by a factor of 2.

The base case occurs when n = 1, and in this case we can find the median with a single query, so T(1) = O(1). So we can write the recurrence:
T(n) = T(n/2) + O(1)
By the master theorem, this recurrence has solution T(n) = O(log n), so the algorithm requires at most O(log n) queries to find the median.

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

Solution:

c)
SOUNDNESS:
To prove the soundness of the algorithm, we need to show that it always returns the correct result, i.e., the nth smallest value among the 2n values. There are two cases:

Case 1: The median is the kth smallest value in the first database.
In this case, since the first database contains k - 1 smaller values, and the second database contains n - k larger values, we know that there are k - 1 + (n - k) = n - 1 values smaller than the median. Therefore, the median is the nth smallest value among the 2n values, which is what we want to find.

Case 2: The median is the kth smallest value in the second database.
In this case, we can similarly see that there are k - 1 + (n - k) = n - 1 values larger than the median, so again the median is the nth smallest value among the 2n values.
Therefore, in both cases, the algorithm returns the correct result.

COMPLETE: To prove the completeness of the algorithm, we need to show that it always terminates and returns a result. Algorithm handles two lists of size n, and each recursion makes progress by splitting the lists in half towards base case (a range of size 1 or 2 in each database). So the algorithm terminates, moreover returns the median, element from first or second database.

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:

```
Algorithm: CountSort(A)
Input : List A of n comparable items.
Output: A sorted array and the number of inversions
if |A| = 1 then return (A; 0)
(A1; c1) := CountSort(Front-half of A)
(A2; c2) := CountSort(Back-half of A)
(A; c) :=MergeCount(A1,A2)
return (A; c + c1 + c2)
//In the Count Sort inversion is: i < j (ai > 2*aj)
```

```
Algorithm: MergeCount (A1, A2)
Input : Two lists of comparable items: A1 and A2.
Output: A merged list and the count of inversions A.
Initialize A to an empty list and c := 0.
i,j=0;
while either A1 or A2 is not empty do
   Pop and append min{front of A1, front of A2} to A.
   if A1[i] > 2*A2[j]
      c += length(A1) – i
      j++;
   else
      i++ //no significant inversions
   end
end
return (A; c)
```

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

Solution:

b)

Let T(n) be the worst-case number of steps in algorithm. In each iteration of the CountSort, we split the array A in 2 arrays of half size, so we reduce the size by 2 and in MergeCount we make n comparisons. So we can write the recurrence:

$T(n) = 2T(n/2) + cn$, $T(1) = c$

By master theorem, we have $\Theta(n\log n)$

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

Solution:

(c) To prove the correctness of the algorithm, we need to show that it correctly counts the number of significant inversions between two orderings.

We can prove this by induction on the size of the input array. For the base case where the array has size 1, there are no significant inversions, and the algorithm correctly returns 0.

For the inductive step, suppose that the algorithm correctly counts the number of significant inversions for all inputs of size $k < n$, and let A be an input of size n. We can assume without loss of generality that the left subarray A[1..mid] has already been sorted, and that the right subarray A[mid+1..n] has already been sorted.

Suppose that there are m significant inversions between the two subarrays. We need to show that the algorithm correctly counts these inversions and returns the sum of this count with the counts obtained during the recursive calls on the subarrays.

During the merge step, we compare elements A[i] and A[j] from the left and right subarrays, respectively. If $A[i] > 2*A[j]$, then we know that A[i] is greater than all remaining elements in the right subarray, and so we increment the count by the number of elements remaining in the left sub

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.

Solution:

```
gerMajorityCard (array A)
    If |A|=1 return A[0]
    If |A|=2 test equivalence, if yes return either card and report fraud
Let A1 be the set of the first ⌊n/2⌋ cards
Let A2 be the set of the remaining cards
getMajorityCard( A1)
    If the card is returned
          test its equivalence against all cards and return if majority
    If no card with majority equivalence has yet been found
       getMajorityCard(A2)
       If a card is returned test its equivalence against all cards
          Return card with majority equivalence if found.
    Return no majority found.
```

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

Solution:

b) Let T(n) be the worst-case number of steps in algorithm. In each iteration of the getMajorityCard, we split the array A in 2 arrays of half size, so we reduce the size by 2. If we find the majority in forst half we compare it with all other cards. So n comparisons.

We can write the recurrence:

$T(n) = 2T(n/2) + cn$, $T(1) = c$

By master theorem, we have $\Theta(n\log n)$

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

c) To prove correctness, we need to show two things:
If there is a set of more than n/2 equivalent cards then the algorithm will correctly identify the fraud.
If there is no such set of cards, then the algorithm will correctly report, "no fraud detected".
Let k be the size of set S which in the case of fraud has to be greater than n/2. We split the set A in 2 subsets A1 and A2. To correctly identify the fraud we must detect majority card, which is equivalent with more than half of n/2 (meaning n/4) cards at least in one of the two subsets. If we found such a set of equivalent cards in the first subset A1, we have to test for the equivalence with all other cards. If its number is greater than n/2 we conclude for fraud. If number k is less or equal to n/2 we repeat the test with the second subset A2. If we find the majority card in second subset equivalent to at least n/4 cards in subset A2, we have to check equivalence with all other cards to get more than n/2 equivalent cards, to conclude for fraud, or if k is les or equal to n/2 that we do not have the fraud.

Solution:

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
```