1. Design a variation of MergeSort which, instead of recurring until the subarray has length 0 or 1, recurs at most a constant \( k \) times, then calls InsertionSort on the resulting \( 2^k \) subarrays of length (approximately) \( n/2^k \). Call this algorithm Depth-\( k \)-MergeSort.

For Example let \( q = \left\lfloor \frac{1+n}{2} \right\rfloor \), \( u = \left\lfloor \frac{1+q}{2} \right\rfloor \), and \( v = \left\lfloor \frac{(q+1)+n}{2} \right\rfloor \). We can picture the operation of Depth-2-MergeSort by the following recursion tree.

The nodes at depth \( i \) represent subarrays of length \( n/2^i \). Depth-2-MergeSort calls InsertionSort on the 4 subarrays at Depth 2. Depth-1-MergeSort recurs down to depth 1 and calls InsertionSort on the 2 subarrays at Depth 1. Depth-0-MergeSort simply calls InsertionSort on the full array.

a. Write pseudo-code for this algorithm. (Hint: Notice that each call to Depth-\( k \)-MergeSort must know its own level in the recursion tree in order to know whether it should recur again, or call InsertionSort.)

b. Determine the asymptotic run time of Depth-\( k \)-MergeSort as a function of both \( k \) and \( n \). In order to simplify the analysis, you may assume that \( n \) is always an exact power of 2. (Hint: First review the discussion of InsertionSort in the text, and note that it runs in time \( \Theta(n^2) \).)

2. Recall the RandSelect\((A, p, r, i)\) algorithm which returns the \( i \)th order statistic of the subarray \( A[p \cdots r] \). The recurrence

\[
t(n) = (n-1) + \left( \frac{n-1}{n^2} \right) \sum_{q=1}^{n-1} t(q)
\]

was derived in class for the average number of comparisons performed by RandSelect on subarrays of length \( n = r - p + 1 \). Use this recurrence to prove that \( t(n) = \Theta(n) \). (Hint: first show directly that \( t(n) \geq \Omega(n) \). Then prove by induction that \( \forall n \geq 1: t(n) \leq 2n \), showing \( t(n) = O(n) \).)
3. Prove the correctness of RadixSort by (finite) induction on $i$, the column being sorted. Be sure to carefully state your induction hypothesis. It should be clear from your proof why the sort on line 2 must be stable, and why the algorithm must sort the digits from least to most significant.

RadixSort($A$, $d$) (Pre: $A[1..n]$ consists of $d$ digit numbers)
1. for $i ← 1$ to $d$
2. sort $A$ on digit $i$ using a stable sort

4. Prove that Counting Sort is stable. Also show that if we reverse the order in which the final loop is executed, the resulting algorithm is correct, but not stable. (This is problem 8.2-3 on page 170).