Question 1. (Credits to UT Austin archives) Although merge sort runs in $O(n \lg n)$ worst-case time and insertion sort runs in $O\left(n^2\right)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. So it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

1. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.
2. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.
3. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?
4. How should we choose k in practice?

Question 2. What is the running time of Quicksort when all elements of array A have the same value? Write down the resulting recursion relation and solve it.

Question 3. Show that the running time of QuickSort is $\Theta\left(n^2\right)$ when the array A contains distinct elements and is sorted in decreasing order.

Question 4: Both quicksort and mergesort have an expected time of $O(n \log n)$ to sort a list containing n items. However, in the worst case, quicksort would require $O\left(n^2\right)$ time, while mergesort's worst case running time is the same as its expected time, $O(n \log n)$.

1: What accounts for the difference in worst case times? How is it that quicksort can require $O\left(n^2\right)$ time, while mergesort always guarantees $O(n \log n)$?

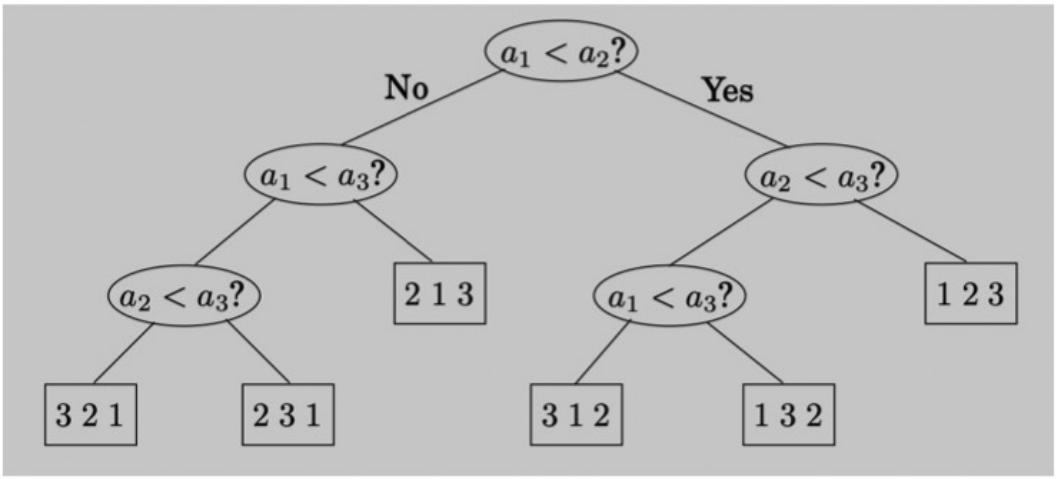
2: Given that mergesort's worst case time is better than quicksort's, why bother ever to use quicksort at all?

Question 5. (See Dasgupta, Papadimitriou, and Vazirani) Prove the optimality of mergesort with $\Omega(n \log n)$ time complexity by showing that the time complexity of ANY comparison-based sorting algorithm would require (at least) such runtime complexity.

Answer: Sorting algorithms can be depicted as trees. See figure below where the tree sorts an array of three elements, a_1, a_2, a_3. It starts by comparing a_1 to a_2 and, if the first is larger, compares it with a_3; otherwise it compares a_2 and a_3, and so on. Eventually we end up at a leaf, and this leaf is labeled with the true order of the three elements as a permutation of $1, 2, 3$. For example, if $a_2 < a_1 < a_3$, we get the leaf labeled 213.

Recall that the depth of the tree –the number of comparisons on the longest path from root to leaf, in this case 3 – is exactly the worst-case time complexity of the algorithm. Now consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a permutation of $\{1, 2, \ldots, n\}$. In fact, every permutation must appear as the label of a leaf. The reason is simple: if a particular permutation is missing, what happens if we feed the algorithm an input ordered according to this same permutation? And since there are $n!$ permutations of n elements, it follows that the tree has at least $n!$ leaves. So this

is a binary tree, and we argued that it has at least $n!$ leaves. Recall now that a binary tree of depth d has at most 2^d leaves (proof: an easy induction on d). So the depth of our tree-and the complexity of our algorithm-must be at least $\log(n!)$. And it is well known that $\log(n!) \geq c \cdot n \log n$ for some $c > 0$.



Prove this statement above!

$$\begin{aligned}
\log(1) + \ldots + \log(n/2) + \ldots + \log(n) &\geq \log(n/2) + \ldots + \log(n) \\
&= \log(n/2) + \log(n/2+1) + \ldots + \log(n-1) \\
&\geq \log(n/2) + \ldots + \log(n/2) \\
&= (n/2)\log(n/2)
\end{aligned}$$

So we establish that any comparison tree that sorts n elements must make, in the worst case, $\Omega(n \log n)$ comparisons, and hence mergesort is optimal.

Question 6: Consider a k-way merge operation. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. You can simply use the merge procedure from mergesort, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n? Can you find a more efficient solution to this problem, using divide-and-conquer?

Answer: If we proceed with the proposed solution then there will be n comparisons for the first array, $2n$ comparisons for the second array, $3n$ comparisons for the third array, etc., so the time complexity will be a sum over all such comparisons in the following form:

$$n + 2n + 3n + 4n + \ldots + (k-1)n = O(nk^2) \tag{1}$$

Instead if we consider merging arrays (1 and 2), arrays (3 and 4), and so on, and then repeatedly merge the previously merged arrays (merge (1, 2) with (3, 4)) there will be $\log(k)$ steps of merging each with kn cost, so the resulting time complexity would be $O(kn \log k)$.

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```