1. (Modified from Goodrich & Tamassia R-8.4-5) Consider the modification of the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$, that is, an element in the middle of the sequence.

(a) What is the running time of this version of quick-sort on a sequence that is already sorted? Use O-notation; you do not need to explain your answer.

Solution. Picking the middle element of a sorted array as the pivot results in a balanced split. So the running time satisfies the recursion $T(n) = 2T(n/2) + n - 1$. This is the same recursion as for merge-sort, which has running time $n \log n - n + 1$ or $\Theta(n \log n)$. •

(b) Describe an input sequence that would cause this version of quick-sort to run in $\Theta(n^2)$ time.

Solution. For an input of size 8, use $8, 6, 4, 2, 1, 3, 5, 7$ or $1, 3, 5, 7, 8, 6, 4, 2$ or $4, 3, 2, 1, 8, 7, 6, 5$ or $6, 4, 7, 1, 8, 2, 3, 5$, etc. The pattern should be that, at any point, the middle element, which is the one being removed, is the smallest largest number of the current array. •

2. (Modified from Goodrich & Tamassia R-8.7) Suppose that the in-place quicksort algorithm (Algorithm 8.9 in the textbook, or the same algorithm as presented in class) is executed on a sequence with duplicate elements. What happens in the partition step when all the elements of the input sequence are equal?

Solution. If we are using the version of Quicksort in which the partitioning scheme produces two subarrays plus the pivot, and one subarray contains elements less than or equal the pivot, and the other contains elements that are greater than or equal to the pivot, then the partitioning will be unbalanced. Quicksort will do $n - 1$ comparisons, and recurse into the lower subarray of size $n - 1$. Therefore, the total amount of work will be $\Theta(n^2)$.

If we are using the version of Quicksort in which the partitioning scheme produces three subarrays containing respectively all elements strictly less than, all elements equal to, or all elements strictly greater than the pivot, then Quicksort will put all the elements in the middle subarray and will not recurse into any array. So the amount of work done is $O(n)$. •

3. (Modified from Goodrich & Tamassia A-8.1) Suppose that you are given a new hardware device that can merge $k > 2$ different sorted lists of total size n into a single sorted list in $O(n)$ time, independent of the value of k. Show that you can use this device to sort n elements in $O(n \log n / \log k)$ time. (Hint: you may use the fact that $\log_k n = \log n / \log k$.)

Solution. Consider the following recursive algorithm:

```
def k_way_mergesort(a):
    if len(a) = 0 or 1:
        return a
    split a into k subarrays of roughly equal size
    for each of the k subarrays b:
        k_way_mergesort(b)
    merge the (now sorted) subarrays into one sorted array using the device
    return a
```

This algorithm has a running time $T(n)$ which satisfies

$$T(n) = kT(n/k) + cn$$

for some constant $c > 0$. We claim that $T(n)$ is $O(n \log n/\log k)$. Specifically, we will show that

$$T(n) \le (c+1)n\log n/\log k \quad \text{for all } n \ge k. \tag{$*$}$$

We now prove this claim by induction on n.

In the base case when $n = k$, the algorithm splits the array into k arrays, recurses into each to do a constant amount of work, and then merges these subarrays in time ck. So the total amount of work is $(c+1)k$.

To prove the inductive step, let us assume that $T(n') \le (c+1)n\log n/\log k$ for all n' such that $k \le n' < n$. We have that

$$\begin{aligned}
T(n) &= k\ T(n/k) + cn && \text{by the recursion}\\
&\le k\left((c+1)\frac{(n/k)\log(n/k)}{\log k}\right) + cn && \text{by the induction hypothesis}\\
&\le (c+1)\frac{n\log n - n\log k}{\log k} + cn && \text{since } \log(n/k) = \log n - \log k\\
&\le c\frac{n\log n}{\log k} + n && \text{simplifying}\\
&\le (c+1)\frac{n\log n}{\log k} && n \le \frac{n\log n}{\log k} \quad \text{for } n \ge k
\end{aligned}$$

Therefore, we proved the claim that $(*)$. So $T(n)$ is $O(n\log n/\log k)$. •

4. Define the "first drop problem", on an input array A of n numbers, to be the problem of finding the first position at which one number drops to a smaller number after it. More formally, the output should be the smallest index i for which $A[i] > A[i+1]$, or $i = n-1$ if no such index exists. For instance, on the input [5,9,2,4,1] the output should be 1, because the first drop is from the values 9 to 2, and the value 9 occurs at position $A[1]$. But on the input [1,2,4,5,9] there is no first drop and the output should be 4. You may assume that no two input numbers are equal.

(a) When $n = 4$, how many different outcomes does the first drop problem have?

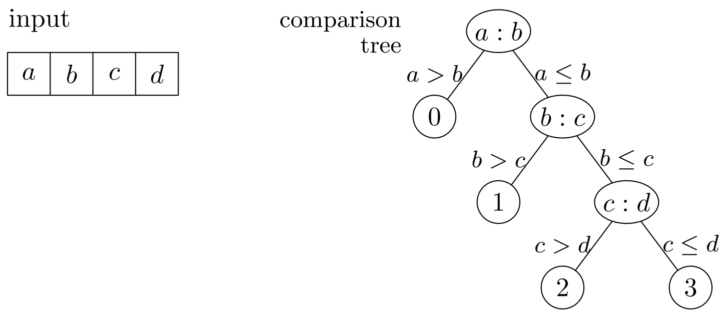
Solution. There are 4 outcomes, one for each index of the array. ●

(b) Based only on your answer to part (a), and the $\lceil \log_2(\#\text{outcomes}) \rceil$ lower bound on the height of any binary comparison tree, how many comparisons would be necessary to correctly solve the first drop problem for $n = 4$?

Solution. A lower bound is $\lceil \log_2(\#\text{outcomes}) \rceil = \log_2 4 = 2$. ●

(c) Draw a comparison tree that uses as few comparisons as possible (in the worst case) to solve the first drop problem for $n = 4$. How many comparisons in the worst case does your tree use?

Solution. The following tree does 3 comparisons in the worst case.



●