Problem 1: 6 You need to sort $n$ integers in the range 1 to $n^2$. Give an $O(n)$ algorithm.

Solution: Consider the representation of integers in base-$n$. An integer $m$ requires $\lceil \log_n m + 1 \rceil$ digits in base-$n$ representation. In case that $m = n^2$, the above formula gives $\lceil \log_n n^2 + 1 \rceil = 3$ digits.

We can use radix-sort on this representation of integers. Each digit is in the range 0 to $k = n - 1$. Using counting sort as the intermediate sort for the input of $n$ integers takes time $\Theta(n + k) = \Theta(n)$.

There are $d = 3$ passes which equals the number of digits. Thus, the total complexity remains $\Theta(n)$ since $d$ is constant.

Problem 2: 8 pts You have $n$ integers in the range 1 to $k$. You need to preprocess the input suitably, so that after preprocessing you can answer some queries in constant time. More specifically: given any two real numbers $a$ and $b$, you have to answer queries about how many of the integers fall into the range $(a, b]$ in $O(1)$ time (the range excludes $a$ and includes $b$). During preprocessing, you do not know the values of $a$ and $b$. You can use $O(k)$ additional storage and $O(n + k)$ preprocessing time.

Solution: We use an extra array $C[1 \ldots k]$ which provides temporary working storage. The array is initialized to zero for each element. Here is the preprocessing procedure:

for j=1 to n
  $C[A[j]] = C[A[j]] + 1$
for i=2 to k
  $C[i] = C[i] + C[i-1]$
After the first loop, $C[i]$ contains the number of elements equal to $i$. After the second loop, $C[i]$ contains the number of elements less than or equal to $i$.

The number of integers in the range $(a, b]$ is computed as $C[b] - C[a]$.

The construction of the array $C$ takes $O(n + k)$ time and requires $O(k)$ additional storage. Finally, the subtraction operation costs $O(1)$ time, so the queries are answered in constant time.

**Problem 3: 8 pts** You have an array of $n$ data records. Each record has a key 0 or 1. Design an $O(n)$ algorithm to sort the data records according to the key values (those with key 0 should come before those with key 1). You can only use constant amount of additional storage during the sorting. Can you use your solution in radix sort so as to sort $n$ records with $b$ bit keys in $O(bm)$ time? (every key has $b$ bits). Justify your answer.

**Solution:** We have an array $A$ of $n$ data records. We use a technique similar to the partition procedure of Quicksort. Keep two indices $i, j$. Initially $i = 0$ and $j = n - 1$. Keep increment $i$ till $A[i] = 1$. Do the same for $j$ till $A[j] = 0$. If $i < j$ exchange the values $A[i], A[j]$. Notice that the algorithm has take care so that none of the indices go out of bounds (i.e in case the array has just 0’s or 1’s). When $i \geq j$ the procedure terminates. The running time of this algorithm is $O(n)$.

Radix sort requires that the intermediate sorting algorithm for each pass be stable (like counting sort). The above algorithm is not stable, so it cannot be used for sorting $n$ records with $b$ bit keys.

**Problem 4: 8 pts** You have to sort a sequence of $n$ elements. The $n$ elements have $n/k$ subsequences of size $k$ each. The subsequences have the following property: All elements of a subsequence are less than those of the preceding one and greater than those of the following subsequence. An example sequence of 6 elements with $k = 2$ is 2, 1, 5, 6, 21, 12. The subsequences here are 2, 1, 5, 6 and 21, 12. Note that all elements of 2, 1 are less than those of 5, 6 and so on. You know the value of $k$. Give an $O(n \log k)$ algorithm to sort the entire sequence. Show that any comparison based sorting needs at least $\Omega(n \log k)$ operations. (It is not rigorous enough to combine the lower bounds for the individual subsequences.)

**Solution:** In order to sort the sequence of $n$ elements, the algorithm must sort the $k$ elements in each of the $n/k$ subsequences. Using one of the comparison-based sorting algorithms that you already know, this step will take $O(k \log k)$ time in the worst case (i.e heapsort, mergesort) or in average
case (i.e Quicksort). This is done for all \( n/k \) subsequences, so total time is \( O(n \log k) \).

In order to prove the lower bound we have to consider a decision tree. There are \( k! \) permutations for each subsequence. The number of possible choices for all \( n/k \) subsequences is \( (k!)^{n/k} \). The first subsequence has \( k! \) choices times the \( k! \) choices of the second subsequence etc. This number of choices represents the number of leaves in the decision tree. The height of this decision tree is:

\[
\begin{align*}
h & \geq \log (k!)^{n/k} \\
& \geq \frac{n}{k} \log k! (1)
\end{align*}
\]

We know that \( \log k! = \Omega(k \log k) \) (2).

Combining relations (1), (2) \( \Rightarrow \):

\[
h = \Omega(n \log k)
\]

Since the worst case of comparisons corresponds to the height of its decision tree, a lower bound on the height of the decision tree is a lower bound on the running time of any comparison based sorting.

QED