1. (10 pts) A server has \(n\) customers waiting to be served. The service time required by each customer is known in advance: it is \(t_i\) minutes for customer \(i\). So if, for example, the customers are served in order of increasing \(i\), then the \(i\)th customer has to wait \(\sum_{j=1}^{i} t_j\) minutes.

We wish to minimize the total waiting time

\[
T = \sum_{i=1}^{n} \text{(time spent waiting by customer } i)\]

Describe an efficient algorithm for computing the optimal order in which to process the customers and give a brief justification of its running time and correctness.

**ANSWER:** We first observe that no matter what order we choose to serve the customers in, the total time taken does not change. The total time (lets call it \(X\)) required to serve all of the customers is always the sum of their service times. Thus, the total waiting time can be reformulated as:

\[
T = \sum_{i=1}^{X} \text{(number of customers still waiting at time } i)\]

Thus, we wish to minimize the number of customers that are still waiting at any given time. This means we want to serve the customers that require less time first, so our optimal order is the customers sorted by increasing service time (which could be proved by induction on \(X\)). Sorting takes \(O(n \log n)\) time.

2. (10 pts) Consider the task of searching a sorted array \(A[1 \ldots n]\) for a given element \(x\): a task we usually perform by binary search in time \(O(\log n)\). Show that any algorithm that accesses the array only via comparisons (that is, by asking questions of the form “is \(A[i] \leq z\)?”), must take \(\Omega(\log n)\) steps.

**ANSWER:** To prove the lower bound for binary search, we use essentially the same argument as for the sorting lower bound. We can view any comparison-based algorithm as a binary tree. Each internal node of this binary tree is one comparison. Instead of having each leaf be a permutation, as in the proof of the sorting lower bound, each leaf is a single element of the array. There must be at least one leaf for each element of the array, as our algorithm would not be able to find an element at a
location not represented by a leaf. There are $n$ elements in the array, so the tree must have at least $n$ leaves. Thus, the depth of the tree is $\Omega \left(\lg n\right)$, and, therefore any such algorithm must take $\Omega \left(\lg n\right)$ comparisons.

3. 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.

(a) (5 pts) Here’s one strategy: Using the merge procedure, 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 $n$?

**ANSWER:** Merging two sorted arrays $A_1$ and $A_2$ with $n_1$ and $n_2$ elements, respectively, takes $O \left(n_1 + n_2\right)$ time. This strategy begins by merging two arrays of size $n$ to create an array of size $2n$. It then merges that with an array of size $n$, and so on. Thus, the running time is

\[
(n + n) + (2n + n) + (3n + n) + \ldots + ((k - 1)n + n)
\]

\[
= 2n + 3n + 4n + \ldots + kn
\]

\[
= O \left(k^2 n\right)
\]

(b) (5 pts) Give a more efficient solution to this problem, using divide-and-conquer. What is its time complexity in terms of $k$ and $n$?

**ANSWER:** This problem is essentially the same as sorting one array of size $kn$ except that each block of $n$ elements is already sorted. Observe that this is exactly the situation we see in merge sort at the log $n$th level of the recursion tree if the data breaks up appropriately! We can thus apply the same idea as the merge sort algorithm, by merging the first array with the second, the third with the fourth, and so on and then repeatedly apply this until all of the arrays have been merged. We do $O \left(kn\right)$ work to merge the $k$ arrays of size $n$ into $k/2$ arrays of size $2n$, and then continue doing $O \left(kn\right)$ work $O \left(\log k\right)$ times until we have a single array of size $kn$. Thus, the running time of this approach is $O \left(k \log kn\right)$.

4. (10 pts) Show that any array of integers $X[1 \ldots n]$ can be sorted in $O \left(n + M\right)$ time, where $M = \max_i \{X[i]\} - \min_i \{X[i]\}$. For small $M$, this is linear time: why doesn’t the $\Omega \left(n \lg n\right)$ lower bound apply in this case?

**ANSWER:** This type of sorting is called CountingSort. We first scan through the array to determine the minimum and maximum numbers. We then create a new array $A$ of size $M$ with each value initially 0. We scan through the array again and, for each element $X[i]$, we increment $A[\min_i \{X[i]\} + X[i]]$. We create a new array $Y[1 \ldots n]$. Finally, we scan through our array $A[]$ and, for each element $A[i]$, we put $A[i]$ values of $i$ into the next $A[i]$ empty slots of $Y[]$.

Essentially, we count the number of times that a value $i$ occurs in our array and store this in $A[i]$. When it comes time to output the sorted array we scan through $A[i]$ and output these values in order.
We scan through arrays of size $n$ and $M$ and do a constant amount of work for each element of these arrays so this takes $O(n + M)$ time. The reason that the $\Omega(n \lg n)$ bound does not apply to this algorithm is that it is not comparison based.