1. Linda claims to have an algorithm that takes an input sequence $S$ and produces an output sequence $T$ that is a sorting of the $n$ elements in $S$.

(a) Give an algorithm, `isSorted()`, for testing in $O(n)$ time if $T$ is sorted.

(b) Explain why the algorithm `isSorted()` is not sufficient to prove a particular output $T$ of Linda’s algorithm is a sorting of $S$.

(c) Describe what additional information Linda’s algorithm could output so that her algorithm’s correctness could be established on any given $S$ and $T$ in $O(n)$ time.

(a) Algorithm.

The algorithm is described in Algorithm 1. Essentially, compare adjacent elements and ensure they are in proper order. Without loss of generality, my algorithm assumes a non-decreasing order ($\leq$).

```
Algorithm 1 isSorted
Input: Sequence T
Output: Boolean stating if T is in non-decreasing order
1: for $i \leftarrow 1 \ldots n - 1$ do
2: if $T_i > T_{i+1}$ then
3: return false
4: return true
```

Time Complexity.

**Theorem 1.** Algorithm runs in $O(n)$ time, where $n$ is the size of the input set.

**Proof.** The loop simply visits each element comparing adjacent elements. This trivially takes $O(n)$ time.

Memory Complexity.

**Theorem 2.** Algorithm uses $O(1)$ additional memory.

**Proof.** There are only a constant number of temporary variables used. Thus, the algorithm has $O(1)$ memory overhead.

(b) Essentially, the algorithm is not sufficient to prove $T$ is a sorting of $S$ because there is no check that the elements of $S$ even show up/correspond to elements in $T$. Trivial example, Linda’s algorithm could always return the empty set and `isSorted()` would always return true.

(c) To fix this, Linda’s algorithm could provide a map between the elements in $S$ and $T$. From this map, lookup of an element in $S$ to its corresponding element in $T$ would take $O(1)$ time, trivially totalling $O(n)$ time. This includes a check that the size of $S$ and $T$ are identical and every $S$ appears once in $T$. 
2. Let $S_1, S_2, \ldots, S_k$ be $k$ different sequences whose elements have integer keys in the range $[0, N - 1]$, for some parameter $N \geq 2$. Describe an algorithm that produces $k$ respective sorted sequences in $O(n + N)$ time, where $n$ denotes the sum of the sizes of those sequences.

Algorithm.

Algorithm 2 first constructs $n$ pairs. Each element is paired with an integer value for the list it is in. Then a standard bucket sort is used on the second key to sort all elements. Finally, a bucket sort is used on the first key to reorder the elements based on their list. Because bucket sort is stable, the individual lists will remain sorted. Finally, the full list is split into the $k$ individual lists.

**Algorithm 2 $k$ Sequence Sort**

**Input:** $k$ sequences $S_1, S_2, \ldots, S_k$

**Output:** Sorted sequences $S_1, S_2, \ldots, S_k$

1: Sequence of pairs $S \leftarrow \emptyset$
2: for $i \leftarrow 1$ to $k$ do
3:    while $\neg S_i$.isEmpty() do
4:        $S.addLast(i, S_i.removeLast())$
5:    BucketSort($S$) using the key as the second pair element
6:    BucketSort($S$) using the key as the first pair element
7: $S_1, S_2, \ldots, S_k \leftarrow$ split($S$)
8: return $S_1, S_2, \ldots, S_k$

**Time Complexity.**

**Theorem 3.** Algorithm 2 runs in $O(n + N)$-time, where $n$ is the sum of the sizes of all input sequences.

*Proof.* The first for loop will execute $n$ move operations from the $i$th sequence to $S$. Using a good sequence implementation, e.g., linked-list this will take $O(1)$ time per move, and $O(n)$ total for this phase. The first bucket sort will execute in time $O(n + N)$ where $N$ is the range of the keys. The second bucket sort will run in time $O(n + k) = O(n)$ (because $n > k$). The final split operation is akin to the first phase and takes $O(n)$ time. In total, the algorithm takes $O(n + N)$ time. Each individual bucket sort will run in time $O(|S_i| + N)$.

**Memory Complexity.**

**Theorem 4.** Algorithm 2 uses $O(N)$ additional memory.

*Proof.* Since items are moved in the first and final phases of the algorithm, no extra memory is used there. The bucket sort needs space for the buckets, and thus required $O(N)$ additional memory. In total, the algorithm requires $O(N)$ additional memory.
3. **Bonus.** Space aliens have given us a program, `alienSplit`, that can take a sequence $S$ of $n$ integers and partition $S$ in $O(n)$ time into sequences $S_1, S_2, \ldots, S_k$ of size at most $\lceil n/k \rceil$ each, such that the elements in $S_i$ are less than or equal to every element in $S_{i+1}$, for $i = 1, 2, \ldots, k-1$, for a fixed number, $k < n$. Show how to use `alienSplit` to sort $S$ in $O(n \log n/\log k)$ time.

**Algorithm.**

The algorithm is described in Algorithm 3. It follows Quick sort – call split, recursively sort each part, and then recombine subsequences together.

**Algorithm 3 Alien sort**

**Input:** Sequence $S$

**Output:** $S$ will be in sorted order

1. if $|S| > 2$ then
2. $S_1, S_2, \ldots, S_k \leftarrow \text{alienSplit}(S)$
3. for $i \leftarrow 1 \ldots k$ do
4. $\text{alienSort}(S_i)$
5. Splice $S_1, S_2, \ldots, S_k$ together

**Time Complexity.**

**Theorem 5.** Algorithm 3 runs in $O(n \log n/\log k)$ time, where $n$ is the size of the input set.

*Proof.* Essentially, we can reason about the recursion tree. The depth of the recursion tree will be $O(\log_k n)$ because the sequence is recursively split into sequences of size $\lceil n/k \rceil$. At each level, $O(n + k)$ work is being done. However, since $n > k$, this simplifies to $O(n)$. So we have total work of $O(n \log_k n) = O(n \log n/\log k)$ (by change of base formula).