1. Suppose we are given two \( n \)-element sorted sequences \( A \) and \( B \) each with distinct elements, but potentially some elements that are in both sequences. Describe an \( O(n) \)-time method for computing a sequence representing the union \( A \cup B \) (with no duplicates) as a sorted sequence.

**Algorithm.**

The algorithm is described in Algorithm 1. Essentially, the algorithm proceeds like the merge step of merge sort, except that when two elements are compared only unique element are added to the final sequence.

**Algorithm 1 Union**

**Input:** Sorted sequences \( A \) and \( B \)

**Output:** Union \( A \cup B \)

1: List \( U \leftarrow \emptyset \)
2: while \( \neg A.\text{isEmpty()} \land \neg B.\text{isEmpty()} \) do
3: \( a \leftarrow A.\text{front}(); b \leftarrow B.\text{front}(); u \leftarrow U.\text{back}() \)
4: \( \text{if } a = b \text{ then} \)
5: \( \quad \text{if } a \neq u \text{ then} \)
6: \( \quad \quad U.\text{add}(a) \)
7: \( \quad A.\text{removeFront}(); B.\text{removeFront}() \)
8: \( \text{else if } a < b \text{ then} \)
9: \( \quad \text{if } a \neq u \text{ then} \)
10: \( \quad \quad U.\text{add}(a) \)
11: \( \quad A.\text{removeFront}() \)
12: \( \text{else if } a > b \text{ then} \)
13: \( \quad \text{if } b \neq u \text{ then} \)
14: \( \quad \quad U.\text{add}(b) \)
15: \( \quad B.\text{removeFront}() \)
16: while \( \neg A.\text{isEmpty()} \) do
17: \( \quad \text{if } A.\text{front()} \neq U.\text{front()} \text{ then} \)
18: \( \quad \quad U.\text{add}(A.\text{front}()); \)
19: \( \quad A.\text{removeFront}() \)
20: while \( \neg B.\text{isEmpty()} \) do
21: \( \quad \text{if } B.\text{front()} \neq U.\text{front()} \text{ then} \)
22: \( \quad \quad U.\text{add}(B.\text{front}()); \)
23: \( \quad B.\text{removeFront}() \)
24: return \( U \)

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in \( O(n) \) time, where \( n \) is the size of each of the input sets.
Proof. This is almost identical to the merge function of merge sort. For similar reasons, it takes $O(n)$ time – each element is visited once (2$n$ elements) and each visit takes $O(1)$ time.

Memory Complexity.

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

Proof. While the algorithm presented only moves elements around, it is reasonable to duplicate the input elements instead. Thus the operation would be bound by at most $O(n)$ elements appearing in the final union.

Note: $O(1)$ is an acceptable memory complexity.
2. Describe a radix-sort method for lexicographically sorting a sequence \( S \) of triplets \((k, l, m)\), where \( k \), \( l \), and \( m \) are integers in the range \([0, N - 1]\), for some \( N \geq 2 \). How could this scheme be extended to sequences of \( d \)-tuples \((k_1, k_2, \ldots, k_d)\), where each \( k_i \) is an integer in the range \([0, N - 1]\)?

**Algorithm.**

The algorithm is described in Algorithm 2 and Algorithm 3. Essentially, repeatedly apply a stable sort, such as bucket sort to the tuples. The buckets would be in the range of \( N \). Essentially, this is identical to radix sort.

**Algorithm 2** Lexicographical sort triplets

**Input:** Sequences \( S \) of triples \((k, l, m)\)

**Output:** Lexicographically ordered sequence

1. \textbf{BucketSort}(\( S \)) using key \( m \)
2. \textbf{BucketSort}(\( S \)) using key \( l \)
3. \textbf{BucketSort}(\( S \)) using key \( k \)
4. \textbf{return } \( S \)

**Algorithm 3** Lexicographical sort \( d \)-tuples

**Input:** Sequences \( S \) of \( d \)-tuples \((k_1, k_2, \ldots, k_d)\)

**Output:** Lexicographically ordered sequence

1. \textbf{for } \( i \leftarrow d \ldots 1 \) \textbf{do}
2. \textbf{BucketSort}(\( S \)) using key \( k_i \)
3. \textbf{return } \( S \)

**Time Complexity.**

**Theorem 3.** Algorithm 2 runs in \( O(n + N) \) time, where \( n \) is the size of the input set and \( N \) is the range of the keys.

**Proof.** Bucket sort takes \( O(n + N) \) time. It is applied 3 times in succession. Thus, the total is \( O(n + N) \). □

**Theorem 4.** Algorithm 3 runs in \( O(d(n + N)) \) time, where \( d \) is the dimensionality of the tuple, \( n \) is the size of the input set, and \( N \) is the range of the keys.

**Proof.** Bucket sort takes \( O(n + N) \) time. It is applied \( d \) times in succession. Thus, the total is \( O(d(n + N)) \). □

**Memory Complexity.**

**Theorem 5.** Algorithm 2 uses \( O(n + N) \) additional memory.

**Proof.** Essentially, \( n \) elements gets places in additional memory of \( N \) buckets akin to a chained hash map. This additional out-of-place memory usage is required and takes \( O(n + N) \) space. □

**Theorem 6.** Algorithm 3 uses \( O(n + N) \) additional memory.

**Proof.** Essentially, \( n \) elements gets places in additional memory of \( N \) buckets akin to a chained hash map. This additional out-of-place memory usage is required and takes \( O(n + N) \) space. □
3. 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, $\text{isSorted}()$, for testing in $O(n)$ time if $T$ is sorted.
(b) Explain why the algorithm $\text{isSorted}()$ 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 4. Essentially, compare adjacent elements and ensure they are in proper order. Without loss of generality, my algorithm assumes a non-decreasing order ($\leq$).

<table>
<thead>
<tr>
<th>Algorithm 4 $\text{isSorted}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Sequence $T$</td>
</tr>
<tr>
<td><strong>Output:</strong> Boolean stating if $T$ is in non-decreasing order</td>
</tr>
<tr>
<td>1: for $i \leftarrow 1 \ldots n - 1$ do</td>
</tr>
<tr>
<td>2: if $T_i &gt; T_{i+1}$ then</td>
</tr>
<tr>
<td>3: return false</td>
</tr>
<tr>
<td>4: return true</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 7.** Algorithm 4 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. $\square$

**Memory Complexity.**

**Theorem 8.** Algorithm 4 uses $O(1)$ additional memory.

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

(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 $\text{isSorted}()$ 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 $T$. 

4. **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 5. It follows Quick sort – call split, recursively sort each part, and then recombine subsequences together.

---

**Algorithm 5 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 9.** Algorithm 5 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 $O(n \log \log n) = O(n \log n / \log k)$ (by change of base formula).