1. Let $A$ be an array of size $n \geq 2$ containing integers from 1 to $n - 1$ inclusive, one of which is repeated. Describe an algorithm for finding the integer in $A$ that is repeated.

**Algorithm.**

The solution is shown in Algorithm 1. The key idea is to subtract the sum of the array from the sum of numbers from 1 to $n$. This informs the number of elements $m$ in the array that are “out of place.” Meaning the number repeated is $n - m$.

**Algorithm 1** Find repeat

**Input:** Array $A$ of length $n$

**Output:** Repeated integer $x$

1: $s \leftarrow \sum_{i=1}^{n-1}i$
2: $m \leftarrow \sum_{a \in A}$
3: return $n - (s - m)$

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in $O(n)$ time, where $n$ is the number of elements in the array.

**Proof.** Summing all of the elements in the array requires a loop visit all elements. This clearly takes $O(n)$ additions and memory accesses. Thus, the total time complexity is $O(n)$.

**Memory Complexity.**

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

**Proof.** Only a constant number of temporary variables are used. Thus, the total memory usage is $O(1)$.

**Note:** It is also acceptable to use extra memory. Fully sorting the data or doing a nested for loop is unacceptable for this problem. They would be slow and inefficient.
2. Describe in detail an algorithm for reversing a singly linked list \( L \) using only a constant amount of additional space.

**Algorithm.**

The algorithm shown in Algorithm 2 will reverse a singly linked list. It iterates over the list and reverses the pointers appropriately. We keep three markers, the “head” of the new list \( n \), the current element \( t \), and the “head” of the old list \( c \). In each step, we make the current element’s next field point to the new head. Then, we increment the pointers appropriately.

**Algorithm 2 Reverse**

**Input:** Singly linked list \( L \)

1: Singly linked list \( L' \leftarrow \emptyset \)
2: Node \( n \leftarrow \) null; Node \( c \leftarrow L.\text{head} \)
3: \( L.\text{head} \leftarrow L.\text{head} \)
4: \( L.\text{tail} \leftarrow c \)
5: while \( c \neq \) null do
6: Language \( t \leftarrow c \)
7: Language \( c \leftarrow c.\text{next} \)
8: Language \( t.\text{next} \leftarrow n \)
9: Language \( n \leftarrow t \)

**Time Complexity.**

**Theorem 3.** Algorithm 2 runs in \( O(n) \) time, where \( n \) is the number of elements in the linked list.

*Proof.* The core loop of the algorithm visits each node once. At each node, three pointer assignments are executed, which each take constant amount of time. In total, the algorithm takes \( O(n) \)-time to execute.

**Memory Complexity.**

**Theorem 4.** Algorithm 2 uses \( O(1) \) additional memory.

*Proof.* Only a constant number of temporary variables are used. Thus, the total memory usage is \( O(1) \).

**Note:** Another possible solution is to construct \( L' \) by removing from \( L \) and inserting in the reversed order in \( L' \).
3. **Bonus.** Write a method, `shuffle(A)`, that rearranges the elements of array $A$ so that every possible ordering is equally likely. You may rely on a function `rand(n)`, which returns a random number between 0 and $n - 1$ inclusive.

**Algorithm.**

Algorithm 3 proceeds as follows: in reverse, for each element of the array, pick a random element in the un-shuffled portion to swap with. The key intuition of why this algorithm yields an unbiased permutation is akin to picking elements at random from a hat. The “right” portion of the array represents the already picked elements, while the “left” is the “hat”. By using this process, each element is equally likely to be in any specific location of the array.

<table>
<thead>
<tr>
<th>Algorithm 3 Shuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Array $A$</td>
</tr>
<tr>
<td>1. for $i \leftarrow n - 1 \ldots 1$ do</td>
</tr>
<tr>
<td>2. $j \leftarrow \text{rand}(i)$</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 5.** Algorithm 3 runs in $O(n)$ time, where $n$ is the number of elements in the array.

*Proof.* The core loop of the algorithm visits each cell of the array once. At each visit two cells are swapped (three assignments). Without loss of generality, assume an element copy takes constant time, then the algorithm also takes $O(n)$-time to execute. \(\square\)

**Memory Complexity.**

**Theorem 6.** Algorithm 3 uses $O(1)$ additional memory.

*Proof.* Only a constant number of temporary variables are used. Thus, the total memory usage is $O(1)$. \(\square\)