1. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.
   - **Push** \((x)\): Add item \(x\) to the end ("back") of the sequence.
   - **Pull**(): Remove and return the item at the beginning ("front") of the sequence.

We've already seen how to implement a queue using either a doubly-linked list or an array-list, so that the data structure uses \(O(n)\) space (where \(n\) is the number of items in the queue) and each operation takes \(O(1)\) amortized time.

(a) Now suppose we want to support the following operation instead of **Pull**:
   - **PullMany**\((k)\): Remove the first \(k\) items from the front of the queue, and return the \(k\)th item removed.

Suppose we use the obvious algorithm to implement **PullMany**:

```plaintext
PullMany(k):
  for i ← 1 to k
    x ← Pull()
  return x
```

Prove that in any intermixed sequence of **Push** and **PullMany** operations, each operation runs in \(O(1)\) amortized time.

(b) Now suppose we also want to support the following operation instead of **Push**:
   - **PushClones**\((x, k)\): Insert \(k\) copies of item \(x\) into the back of the queue.

Suppose we use the obvious algorithm to implement **PushClones**:

```plaintext
PushClones(x, k):
  for i ← 1 to k
    Push(x)
```

Prove that for any integers \(\ell\) and \(n\), there is a sequence of \(\ell\) **PushClones** and **PullMany** operations that requires \(\Omega(n\ell)\) time, where \(n\) is the maximum number of items in the queue at any time. Such a sequence implies that the amortized time for each operation is \(\Omega(n)\).

(c) Describe a data structure that supports arbitrary intermixed sequences of **PushClones** and **PullMany** operations in \(O(1)\) amortized time per operation. Like a standard queue, your data structure should use only \(O(1)\) space per item. [Hint: At least one of the algorithms **PushClones** and **PullMany** must be different!]

---

The remaining problems are for you to play with on your own.
Discussion in office hours or on Discord is welcome, but don't submit solutions!
2. Suppose we have a standard queue with one additional operation \texttt{Size()}, which returns the current number of items in the queue. We’ve already seen how to implement a queue using either a doubly-linked list or an array-list, so that the data structure uses \( O(n) \) space (where \( n \) is the number of items in the queue) and each operation takes \( O(1) \) amortized time.

Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in \( \Theta(n) \) worst-case time.

\begin{verbatim}
Decimate():
n ← Size()
for i ← 0 to n − 1
  if i mod 10 = 0
    Pull() \hspace{1cm} \langle\text{result discarded}\rangle
  else
    Push(Pull())
\end{verbatim}

Prove that in any intermixed sequence of \texttt{Push}, \texttt{Pull}, and \texttt{Decimate} operations, the amortized time for each operation is \( O(1) \).

3. Suppose you are maintaining a circular array \( X[0..n−1] \) of counters, each taking a value from the set \( \{0, 1, 2\} \). The following algorithm increments one of the counters; if the counter overflows, the algorithm resets it 0 and recursively increments its two neighbors.

\begin{verbatim}
Increment(i):
  \hspace{1cm} X[i] ← X[i] + 1
  \hspace{1cm} if X[i] = 3
  \hspace{1.5cm} X[i] ← 0
  \hspace{1.5cm} Increment((i − 1) mod n)
  \hspace{1.5cm} Increment((i + 1) mod n)
\end{verbatim}

(a) Suppose \( n = 5 \) and \( X = [2, 2, 2, 2, 2] \). What does \( X \) contain after we call \texttt{Increment(2)}?

(b) Suppose \( n \geq 3 \) and all counters are initially 0. Prove that \texttt{Increment} runs in \( O(1) \) amortized time.

4. The simplest form of the \textit{dictionary} or \textit{map} data type maintains a set \( S \) of items, all of the same arbitrary type, subject to the following operations:

- \texttt{Find(\( x \))}: If \( x \in S \), return a pointer directly to \( x \)'s record in the data structure; otherwise, return \texttt{None}.
- \texttt{Insert(\( x \))}: Add item \( x \) to the set \( S \).
- \texttt{Delete(\( x \))}: Remove item \( x \) from the set \( S \).

If the data set is \textit{static}, meaning that we will never call \texttt{Insert} or \texttt{Delete}, one of the simplest dictionary data structures is a sorted array. Then we can implement \texttt{Find} using binary search in \( O(\log n) \) worst-case time.

In the honors lecture on Monday, we showed that if we mark \texttt{Deleted} items with tombstones and rebuild the array when most of its items are marked, \texttt{Delete} can be implemented with one call to \texttt{Find} plus \( O(1) \) amortized time.
But what about \texttt{INSERT}? After all, we can’t just “pretend to insert” new items into the array! One simple idea proposed by Jon Bentley and James Saxe in 1980, is to insert new items in a second recursively-defined data structure. When the new data structure stores the same number of items as the main sorted array, we merge them together into a new larger sorted array.

Expanding the recursion, we can describe the Bentley-Saxe data structure as follows. Conceptually, data structure consists of an infinite sequence $A_0,A_1,A_2,\ldots$ of sorted arrays; for each index $i$, the array $A_i$ stores exactly $2^i$ items. Each array $A_i$ is \textit{active} if and only if the $i$th bit of the binary representation of $n$ is equal to 1, and \textit{inactive} otherwise. The $n$ items in the dictionary are stored in the active arrays; any data stored in the inactive arrays is meaningless. (In practice, inactive arrays are not stored at all.) Each active array is sorted, but items in different active arrays can be in either order.

For example, the 26 letters of the English alphabet would be stored in three sorted arrays $A_4,A_3,$ and $A_1$, because $26=11010_2=2^4+2^3+2^1$, possibly as follows:

\begin{center}
\begin{tabular}{ccccccccccccccccccc}
\textbf{J} & \textbf{Q} & \textbf{A} & \textbf{E} & \textbf{I} & \textbf{O} & \textbf{R} & \textbf{U} & \textbf{W} & \textbf{Y} & | & \textbf{B} & \textbf{C} & \textbf{D} & \textbf{F} & \textbf{G} & \textbf{H} & \textbf{K} & \textbf{L} & \textbf{M} & \textbf{N} & \textbf{P} & \textbf{S} & \textbf{T} & \textbf{V} & \textbf{X} & \textbf{Z}
\end{tabular}
\end{center}

(a) Describe an implementation of \texttt{Find}(x) and analyze its worst-case running time.

(b) Bentley and Saxe’s algorithm for \texttt{INSERT} mirrors the standard algorithm for incrementing a binary counter. Their algorithm uses the \texttt{MERGE} subroutine from mergesort, which merges two sorted arrays into a single sorted array in linear time.

\begin{algorithm}
\begin{algorithmic}
\State $\texttt{INSERT}(x)$:
\State \hspace{1em} $\texttt{NewA} \leftarrow$ new array of size 1 containing $x$
\State \hspace{1em} $i \leftarrow 0$
\While{$A_i$ is active}
\State \hspace{1em} $\texttt{NewA} \leftarrow$ \texttt{MERGE(}$\texttt{NewA},A_i$\texttt{)}  \hspace{1em} ($O(2^i)$ time)
\State \hspace{1em} mark $A_i$ inactive
\State \hspace{1em} $i \leftarrow i + 1$
\EndWhile
\State $A_i \leftarrow \texttt{NewA}$
\State mark $A_i$ active
\end{algorithmic}
\end{algorithm}

\texttt{INSERT} runs in $\Theta(n)$ time in the worst case (for example, when $n$ becomes a power of 2). Prove that in any sequence of \texttt{INSERTS} starting with an empty data structure, each \texttt{Insert} runs in $O(\log n)$ amortized time.

(c) Finally, Bentley and Saxe support \texttt{DELETE}(x) by tombstoning. (The \texttt{INSERT} algorithm keeps all items in the data structure, even if they are “dead”.)

Prove that in any intermixed sequence of \texttt{INSERTS} and \texttt{DELETES}, starting with an empty data structure, each \texttt{INSERT} runs in $O(\log n)$ amortized time, and each \texttt{DELETE} uses one call to \texttt{Find} plus $O(1)$ amortized time.

\begin{itemize}
\item[(d)] Modify this data structure so that \texttt{Find} runs in $O(\log n)$ time, and other amortized time bounds are the same (up to constant factors). \textit{[Hint: Your modified data structure should still consist of $O(\log n)$ sorted arrays whose sizes grow \textbf{roughly} exponentially. Sure, you could use a balanced binary search tree instead, but that would be too easy.]}
\end{itemize}