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 kth item removed.

   Suppose we use the obvious algorithm to implement PullMany:

```
PullMany($k$):
  for $i \leftarrow 1$ to $k$
      $x \leftarrow$ 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:

```
PushClones($x, k$):
  for $i \leftarrow 1$ to $k$
      Push($x$)
```

   Prove that for any integers ℓ and n, there is a sequence of ℓ 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 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 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.

```
DECIMATE():
  $n \leftarrow$ SIZE()
  for $i \leftarrow 0$ to $n-1$
      if $i \bmod 10 = 0$
          PULL()      ⟨⟨result discarded⟩⟩
      else
          PUSH(PULL())
```

   Prove that in any intermixed sequence of PUSH, PULL, and 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.

```
INCREMENT($i$):
  $X[i] \leftarrow X[i] + 1$
  if $X[i] = 3$
      $X[i] \leftarrow 0$
      INCREMENT($(i-1) \bmod n$)
      INCREMENT($(i+1) \bmod n$)
```

   (a) Suppose $n = 5$ and $X = [2, 2, 2, 2, 2]$. What does X contain after we call INCREMENT(2)?

   (b) Suppose $n \geq 3$ and all counters are initially 0. ***Prove*** that INCREMENT runs in $O(1)$ amortized time.

4. The simplest form of the *dictionary* or *map* data type maintains a set S of items, all of the same arbitrary type, subject to the following operations:

   - FIND(): If $x \in S$, return a pointer directly to x's record in the data structure; otherwise, return NONE.
   - INSERT(x): Add item x to the set S.
   - DELETE(x): Remove item x from the set S.

   If the data set is *static*, meaning that we will never call INSERT or DELETE, one of the simplest dictionary data structures is a sorted array. Then we can implement FIND using binary search in $O(\log n)$ worst-case time.

   In the honors lecture on Monday, we showed that if we mark DELETED items with tombstones and rebuild the array when most of its items are marked, DELETE can be implemented with one call to FIND plus $O(1)$ amortized time.

But what about 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 *active* if and only if the ith bit of the binary representation of n is equal to 1, and *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 $26 = 11010_2 = 2^1 + 2^3 + 2^4$, possibly as follows:

$$\boxed{\texttt{J Q}}\ \boxed{\texttt{A E I O R U W Y}}\ \boxed{\texttt{B C D F G H K L M N P S T V X Z}}$$

(a) Describe an implementation of FIND(x) and analyze its worst-case running time.

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

```
INSERT($x$):
  $NewA \leftarrow$ new array of size 1 containing $x$
  $i \leftarrow 0$
  while $A_i$ is active
      $NewA \leftarrow$ MERGE($NewA, A_i$)    ⟨⟨$O(2^i)$ time⟩⟩
      mark $A_i$ inactive
      $i \leftarrow i + 1$
  $A_i \leftarrow NewA$
  mark $A_i$ active
```

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 INSERTS starting with an empty data structure, each *Insert* runs in $O(\log n)$ amortized time.

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

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

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