1. **Binary Counters with Reset!** Consider the binary counter problem that we examined in class. The counter starts at 0. In addition to the INCREMENT operation, we also wish to support the RESET operation which resets the counter to 0. Show that any sequence of INCREMENT and RESET operations can be performed in \( O(n) \) time. (Hint: Keep a pointer to the high-order 1.) You are free to use either the potential method or the accounting method in your analysis. (Problem taken from HMC CS181b, taught by Ran Libeskind-Hadas.)

2. **Binary Counters with Decrement!** Consider a \( k \)-bit binary counter with the decrement operation. Show that the worst-case complexity of performing a sequence of \( n \) operations is now \( \Omega(kn) \) (thus averaging \( \Omega(k) \) per operation).

3. **Minqueues.** Consider an extension of the queue abstract data type, called a minqueue, which supports operations ENQUEUE, DEQUEUE, and FINDMIN. Assume that the elements to be stored are integers (or any other totally ordered data type). FINDMIN simply returns the smallest element in the minqueue, but does not remove it. Using a standard implementation of a queue, the FINDMIN operation takes \( O(n) \) time in the worst case. However, consider a more clever data structure for this abstract data type which works as follows: There are two regular queues used. The first queue is called the “real queue” and the second queue is called the “helper queue”. When the ENQUEUE(\( x \)) operation is performed, the element \( x \) is enqueued in the regular way on the real queue. The element \( x \) is also enqueued at the end of the helper queue. However, if the element \( y \) immediately “in front” of \( x \) on the helper queue is larger than \( x \), then \( y \) is removed from the helper queue. This process of \( x \) annihilating the element immediately in front of it is repeated until the element immediately in front of \( x \) is less than or equal to it or \( x \) is at the front of the helper queue.

   (a) Describe how the DEQUEUE and FINDMIN operation are implemented in this data structure.
   (b) Give the worst case running tim for each of ENQUEUE, DEQUEUE, and FINDMIN using this data structure.
   (c) Give an argument using the potential method to show that any sequence of \( n \) ENQUEUE, DE- QUEUE, and FINDMIN operations requires only \( O(n) \) time.

   (Problem taken from HMC CS181b, taught by Ran Libeskind-Hadas.)

4. **Random Access Queues.** Suppose we wish to implement a queue with random access, supporting the operations ENQUEUE, DEQUEUE, and PEEK. The PEEK operation accepts a non-negative integer argument \( j \) and returns the element that is \( j \) positions behind the head of the queue. Describe how to implement this data structure using a dynamic array. What is the worst-case and amortized complexity of each operation? You may assume results stated in class. (NOTE: Do not copy the entire array on every DEQUEUE.)

5. (Grads only!) **Making binary search dynamic.** CLRS, Problem 17-2. (If you do not have the CLRS textbook, please let me know and I will send you a copy of the problem.)