CS 61B Lab 14
May 3, 2005

Goal: This lab will help you get comfortable with amortized analysis.

Consider a binary counter implemented as an array of booleans. Each digit of the counter is represented by a boolean in the array. The least significant bit is at index zero, and has a value of one. Each subsequent digit is worth twice the previous digit.

There is only one operation that changes the data structure: incrementing the counter. Therefore, the array increases like this.

```
index          ...   5   4   3   2   1   0
              ------------------------------
digits         ... | 0 | 0 | 0 | 0 | 0 | 0 |  -----\
              ------------------------------     increment()
              ------------------------------  <----/
               ... | 0 | 0 | 0 | 0 | 0 | 1 |  -----\
              ------------------------------     increment()
              ------------------------------  <----/
               ... | 0 | 0 | 0 | 0 | 1 | 0 |  -----\
              ------------------------------     increment()
              ------------------------------  <----/
               ... | 0 | 0 | 0 | 0 | 1 | 1 |  -----\
              ------------------------------     increment()
              ------------------------------  <----/
               ... | 0 | 0 | 0 | 1 | 0 | 0 |
              ------------------------------
                             .
                             .
                             .
              ------------------------------
               ... | 0 | 1 | 1 | 1 | 1 | 1 |  -----\
              ------------------------------     increment()
              ------------------------------  <----/
               ... | 1 | 0 | 0 | 0 | 0 | 0 |
              ------------------------------
```

Here's the code for incrementing the counter.

```
public class Counter {
  private boolean[] digits;

  public Counter(int size) {
    digits = new boolean[size];
  }

  public void increment() {
    int i = 0;
    while (digits[i]) {                *
      digits[i] = false;               *
      i++;                             *
    }                                  *
    digits[i] = true;                  *
  }
}
```

Obviously, the inner loop runs d times if the first d digits are ones, and d can be arbitrarily large. Your job is to prove that increment() runs in O(1) amortized time.

Assume that:
- The cost (in time) of testing a bit, changing it, and incrementing i is at most one dollar (for all three computations together).
- The cost for the method call to increment(), plus initializing i, is exactly one dollar. This cost does not include the lines marked with asterisks (*).

Your job is to figure out:
- The amortized cost of an increment() operation, in dollars. Give the smallest amortized cost possible (given the information you have) that will ensure that your bank account never goes below zero.
- An explanation for why this analysis is correct. In other words, show that your bank balance will never go below zero.
- Suppose we add a decrement() operation, which decreases the counter by one. Why is the amortized cost of the counter operations no longer in O(1)?

Here's your big hint: simply by looking at the counter, you can tell very easily exactly how many dollars are in the bank.

If you're having trouble figuring it out, take a guess at the right dollar amount, then modify the code in Counter.java to keep track of the bank balance. See if you can see a connection between the counter and the bank balance.

Check-off
----

2 points: How many dollars are in the bank for any given counter value?
1 point: What is the amortized cost of increment(), in dollars? (Give the smallest answer that can be justified by the information above.)
1 point: Explain why adding a decrement() operation invalidates the analysis.