Problem 1. In Exercise 97 in the textbook, we asked you to implement the Rooms interface, reproduced in Fig. 1. The Rooms class manages a collection of rooms, indexed from 0 to \( m \) (where \( m \) is a known constant). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. The last thread to leave a room triggers an onEmpty() handler, which runs while all rooms are empty.

Fig. 2 shows an incorrect concurrent stack implementation.

1. Explain why this stack implementation does not work.
2. Fix it by adding calls to a two-room Rooms class: one room for pushing and one for popping.

Problem 2. Recall that when a token visits a balancer, it atomically reads the toggle value and complements it, and then departs on the output wire indicated by the old toggle value. Instead, an antitoken complements the toggle value, and then departs on the output wire indicated by the new toggle value.

Let \( B \) be a balancing network in a quiescent state \( s \), and suppose a token enters on wire \( i \) and passes through the network, leaving the network in state \( s' \). Show that if an antitoken now enters on wire \( i \) and passes through the network, then the network goes back to state \( s \).

Problem 3. Figure 12.14 of the textbook gives a synchronized Balancer implementation. Provide an efficient lock-free implementation of a Balancer, using Java’s AtomicBoolean class. Provide traverse and antitraverse methods for tokens and antitokens.

Problem 4. Let \( B \) be a width-\( w \) balancing network of depth \( d \) in a quiescent state \( s \). Let \( n = 2^d \). Prove that if \( n \) tokens enter the network on the same wire, pass through the network, and exit, then \( B \) will have the same state after the tokens exit as it did before they entered.

```java
public interface Rooms {
    public interface Handler {
        void onEmpty();
    }
    void enter(int i);
    boolean exit();
    public void setExitHandler(int i, Rooms.Handler h);
}
```

Figure 1: The Rooms interface
public class Stack<T> {
    private AtomicInteger top;
    private T[] items;
    public Stack(int capacity) {
        top = new AtomicInteger();
        items = (T[]) new Object[capacity];
    }
    public void push(T x) throws FullException {
        int i = top.getAndIncrement();
        if (i >= items.length) { // stack is full
            top.getAndDecrement(); // restore state
            throw new FullException();
        }
        items[i] = x;
    }
    public T pop() throws EmptyException {
        int i = top.getAndDecrement() - 1;
        if (i < 0) { // stack is empty
            top.getAndIncrement(); // restore state
            throw new EmptyException();
        }
        return items[i];
    }
}

Figure 2: Unsynchronized concurrent stack