Shuffling overhand

Problem description

One of the standard ways that people shuffle a deck of cards is called the *overhand shuffle*. In this shuffle, the deck is held in one hand (say the left hand), a group of cards is split off the top and transferred to the right hand. Then another group is split off the top and placed on top of the group in the right hand. This is repeated until all the cards are in the right hand (and then generally the whole shuffle is repeated a few times).

For instance with six cards initially in order *abcdef* from top to bottom, and group sizes of 3, 2, and 1, followed by a second shuffle with group sizes of 2, 2, and 2 we would get:

$$abcdef \xrightarrow{[3,2,1]} fdeabc \xrightarrow{[2,2,2]} bceafd.$$

you will implement an interface for manipulating "decks of cards" (a.k.a., arrays of integers) using overhand shuffles, and then investigate some properties of these shuffles.

The interface

```java
public interface Overhand {

  public void makeNew(int size);
  public int[] getCurrent();
  public void shuffle(int[] blocks);
  public int order(int[] blocks);
  public int unbrokenPairs();


}
```

Detailed specification

You must write three classes **OverhandShuffler.java**, **OverhandApp.java**, and **BlockSizeException.java** which are described below:

Your class **OverhandShuffler** should implement the **Overhand** interface according to the following conditions:

- **makeNew(int size)**: Makes a new deck consisting of **size** cards numbered from **0** up to **size-1** from top to bottom.
- **int[] getCurrent()**: Returns the current state of the deck as an **int[]** (modifying the result shouldn't effect the original deck).
- **shuffle(int[] blocks)**: Shuffles the current state of the deck according to the array of block sizes given.
- **int order(int[] blocks)**: Returns the minimum number of times that the deck could be shuffled (from its initial state) using the same set of block sizes each time (as given by its argument) in order to return the deck to its initial state.
- **int unbrokenPairs()**: Returns the number of pairs of cards which were consecutive in the original deck, and are still consecutive (and in the same order) in the current state of the deck.

Both **shuffle(int[] blocks)** and **order(int[] blocks)** should throw a **BlockSizeException** if **blocks** contains any negative integers, or the sum of its entries is not equal to the total size of the deck. The **shuffle(int[] blocks)** operation should be of complexity no worse than $O(n)$ where n is the number of cards in the deck.

After a sequence of shuffles we can get the current state of the deck. Is it possible to figure out (without having a record of what shuffles were used) what the state of the deck would be if we repeated the same sequence of shuffles (starting from the current state)? If you think the answer is yes, add a function **tryRepeat()** to implement it. If you think the answer is no, add a function **tryRepeat()** which prints your explaination why to *stderr*.

The concept of "unbroken pairs" applies to arbitrary permutations of a deck of cards. One way to see whether or not a shuffle is (close to) random is to look at statistics like the number of unbroken pairs, and compare them to what one would expect to see in a random permutation. In this part of the assignment you're asked to carry out this experiment on a deck of 50 cards.

- By using random permutations of the numbers from 0 to 49, work out (by experiment) how the number of unbroken pairs seems to be distributed.
- To carry out a random overhand shuffle, take each of the 49 possible places where there might be a break between blocks and make it a break with probability 0.1. For various numbers of repeated random shuffles of this type, work out (by experiment) how the number of unbroken pairs seems to be distributed.

You should add these methods to your **OverhandShuffler** class as you perform the experiments described above:

- **randomShuffle()**: Perform a random overhand shuffle using a break probability of 0.1.
- **int countShuffles(int unbrokenPairs)**: Return how many random shuffles need to be done before the number of unbroken pairs is less than the given parameter.
- **load(int[] cards)**: Load the deck using the given numbers. No error checking is required.

Report the results of your experiments in an easily readable form and write one or two paragraphs analysing the results of the experiments. Specifically, address the question "how many random overhand shuffles seem to be necessary so that the number of unbroken pairs behaves as it would for random permutations?" No formal justification is required - but your data should convince the reader that your answer is not unreasonably large or small. Your report should be a one page PDF (single side), containing your analysis and graphs/tables.

In addition to your **OverhandShuffler** and **BlockSizeException** classes you should write an **OverhandApp** class which creates an instance of **OverhandShuffler** and manipulates it based on lines of input read from *stdin* as specified in the examples below:

Command	Arguments	Description
make-new	18	Make a new ordered deck from 0 to 17
print		Print a string representation of the current deck
shuffle	2 3 10 3	Call **shuffle([2,3,10,3])**
order	1 4 11 2	Print the result of calling **order([1,4,11,2])**
unbroken-pairs		Print result of calling **unbrokenPairs()**
random-shuffle		Call **randomShuffle()** on the current deck
count-shuffles	15	Print the result of **countShuffles(15)**
load	4 3 1 0 2 5	Load deck with given numbers (no error checking)

When entering lines of input the first letter of each command may be used as an abbreviation for the full command. When a command expects numbers to be given as an argument, they should be space separated and on the same line as the command.

```java
import java.util.*;

public class ExampleApp {

    public static void main(String[] args) {
        StringBuilder s = new StringBuilder();
        Scanner input = new Scanner(System.in);
        while (input.hasNextLine()) {
            handleLine(s, input.nextLine());
        }
    }

    private static int[] getNums(Scanner input) {
        List<Integer> numlist = new ArrayList<Integer>();
        while (input.hasNextInt()) {
            numlist.add(input.nextInt());
        }
        int[] nums = new int[numlist.size()];
        for (int i = 0; i < nums.length; i++) {
            nums[i] = numlist.get(i);
        }
        return nums;
    }

    public static void handleLine(StringBuilder str, String input) {
        Scanner scan = new Scanner(input);
        if (scan.hasNext()) {
            String command = scan.next();
            switch (command) {
                case "add": case "a": // add words + trailing space
                    while (scan.hasNext()) {
                        str.append(scan.next() + " ");
                    }
                    break;
                case "load-nums": case "l": // load ints as a string
                    int[] nums = getNums(scan);
                    str.append(Arrays.toString(nums));
                    break;
                case "new-length": case "n": // set new length
                    if (scan.hasNextInt()) {
                        str.setLength(scan.nextInt());
                    }
                    break;
                case "print": case "p": // print with double quotes
                    System.out.printf("\"%s\"\n", str);
                    break;
                case "reverse": case "r": // reverse
                    str.reverse();
                    break;
                case "size": case "s": // print size
                    System.out.println(str.length());
                    break;
            }
        }
    }
}
```