Dealing with it

Problem description

In these unusual times, what passes for amusement is a pretty broad set of activities (so long as they can be carried out in the bubble!) Lately, I've found myself idling away time just dealing out cards and picking them up again in various different ways.

For instance, if I have a pile of 12 cards I might deal them out face up in four rows of three starting at the top left, dealing a row at a time from left to right, and finishing in the bottom right. If I pick them up again in the same way (keeping them face up in hand and putting new cards on the top of the pile) then that just leaves the pile in the same order as I started. But, there are all sorts of exciting things I might do instead:

- I might pick them up by columns from top to bottom and left to right; or
- I might pick them up by columns from top to bottom and right to left; or
- etc.

In the example above (12 cards, dealt in four rows of three) if the cards were initially numbered 1 through 12 from top to bottom, after I laid them out I'd see:

1	2	3
4	5	6
7	8	9
10	11	12

Then if I picked them up bottom to top and left to right the new pile of cards (reading from the top) would be 10, 7, 4, 1, 11, 8, 5, 2, 12, 9, 6, 3.

Generally speaking, the goal of this assignment is to write code that simulates this process and saves me from actually having to do it physically.

The interface

```
public interface CardPile {

  /*
     Loads a copy of the given array as the pile of cards.
  */
  public void load(int[] cards);

  /*
     Creates a pile of n cards numbered (top to bottom) from 1 to n.
  */
  public void load(int n);

  /*
     Returns a copy of the pile of cards.
  */
  public int[] getPile();

  /*
     Transforms the pile of cards given a row length and a specification
     for picking them up. See assignment details for required behaviour
     if the number of cards is not a multiple of the given row length or
     the specification is invalid.
  */
  public void transform(int rowLength, String spec);

  /*
     Returns the minimum positive number of times we would need to call
     transform(rowLength, spec) on the current pile and return it to its
     original order.
  */
  public int count(int rowLength, String spec);


}
```

Detailed specification

The assignment consists of three parts.

1. Provide a class **`CP`** which implements the **`CardPile`** interface according to the following conditions:

   - **`load(int[] cards)`**: Initialises the pile of cards to consist of the contents of the array provided (the elements of the array represent the cards in order from top to bottom of the pile). The internal representation of the pile

need not be as an array, but regardless, no methods of **CP** should modify the originally loaded array.

- **load(int n)**: Initialises the pile of cards to have size **n** in order from **1** through **n** from top to bottom.
- **getPile()**: Returns an array representing the contents of the current pile. Modifying this array should not affect the **CP** object.
- **transform(int rowLength, String spec)**: Transforms the pile of cards by laying them out left to right and top to bottom in rows of length **rowLength** and then picking the up according to the specification **spec**. The string **spec** should be one of: **TL**, **TR**, **BL**, **BR**, **LT**, **RT**, **LB**, or **RB**. The first four represent picking the cards up by columns starting either at the top (**T**) or bottom (**B**) and with the leftmost (**L**) or rightmost (**R**) column first. The remaining four are similar but refer to picking the cards up by rows.
- **count(int rowLength, String spec)**: Returns the minimum (positive) number of times the pile needs to be transformed under the given specification in order that it returns to its original state.

If the argument to **load** (in the second case) is not a positive integer, or if (in **transform** or **count**) the size of the pile is not a multiple of **rowLength** or the string **spec** is not one of the eight specified then a **CardPileException** with an appropriate message should be thrown.

2. Add a **main** method to your class **CP** which operates as specified in the "Input and Output" section below.
3. The reason for automating this kind of a procedure is so that one can experiment and investigate without having to tediously deal out and pick up lots of cards. In a short report answer the following questions:
   (a) Consider the **count** values resulting from the "pick up by rows" specifications (those beginning with an **L** or **R**). What values do they take and why?
   (b) What is the maximum **count** value produced for any specification and any pile size of 20 or less? What pile size(s), row length(s) and specification(s) produce it? Given a pile size, row length, and specification can you think of a way of computing its **count** that doesn't rely on actually carrying out that many transformations?
   (c) There are 720 possible card piles consisting of the numbers 1 through 6 in some order. Call such a pile *accessible* if it can be reached from the original pile 123456 by some sequence of transformations. How many accessible piles are there? What about seven, eight or nine card piles? For how large a value of **n** do you think it might be feasible to compute the number of accessible piles (and why)?

Your report should be a one- or two-page PDF (in 12-point font or thereabouts) containing both your answers and discussions. A certain amount of flexibility is allowed. However, reports of half a page or less are unlikely to contain the required information (or at least not in a readable form or sufficient depth). On the other hand, if you feel inspired to do a more in-depth analysis which produces a longer report, then the first page should include a stand-alone summary of your findings. Note that some of the questions (for instance the last part of the third one) are quite open-ended – the answer is not the most important part. A decent and well-written explanation of why you think your answer is reasonable is more significant.

Input and output

For checking, your program's **`main`** method should accept input either from the *command-line* or *stdin* as follows:

- If there are command-line arguments then the first two items will always be a positive integer **`n`** representing the size of the pile (initialised using **`load(int n)`**), and a positive integer **`r`** representing the row length. If they are present, then the third and later items should be two-character strings representing specifications.
  - If there are exactly two command-line arguments, then the output should be the counts associated with each of the possible specifications for that pile size and row length as shown below:

    ```
    java week11/CP 6 3
    ```

    should result in this output

    ```
    TL 4
    BL 3
    TR 6
    BR 4
    LT 1
    LB 2
    RT 2
    RB 2
    ```

  - If there are three or more command-line arguments, then the output should be a series of lines representing the original pile (as a space separated list of integers) and then the pile after each of the successive transformations is applied as shown below:

    ```
    java week11/CP 6 3 BL TR RB
    ```

should result in this output

```
1 2 3 4 5 6
4 1 5 2 6 3
5 3 1 6 4 2
2 4 6 1 3 5
```

- If there are no command line arguments then it should then read any lines of input from *stdin* and deal with them as described in the following table:

Input	Action
c r s	call and print **count(r,s)** with the given row length and spec
l n	load the pile with cards from 1 to n
L n n n n ...	load the pile with given numbers (note: capital L)
p	print the cards as a white space separated list
P n	print the cards in rows of length n (note: capital P)
t r s	call **transform(r,s)** with the given row length and spec

So, for example, if a file `script.txt` consisted of the following lines:

```
l 8
c 4 RB
L 1 3 5 7 9 11 11 9 7 5 3 1
p
P 6
t 6 BL
p
```

Then, the output of

```
java week11/CP < script.txt
```

would be

```
2
1 3 5 7 9 11 11 9 7 5 3 1
1 3 5 7 9 11
11 9 7 5 3 1
11 1 9 3 7 5 5 7 3 9 1 11
```

Note that these behaviours are exclusive – the program should either use the command-line arguments (if any are present) or read from *stdin* (if there are no command-line arguments), but not both.

Group work and Submission

For this assignment you may work in teams of up to three people. You may select your own group and inform us of your choice by **4pm Friday April 24^{th}.**

Send an email to ihewson@cs.otago.ac.nz with the names and University user codes of all students in your group. For example, John Smith smijo123, Sue Lee leesu456, Kate Brown broka789. If you're planning to work on your own, then please also send Iain an email to indicate this (we recommend that you work in a group!)

If you'd like to work in a group but haven't been able to find anyone to work with please indicate that in an email and we'll attempt to match up any of those requests we receive.

By week 10 of the semester, you will have been notified by email how to check your assignment against some simple test files to make sure that it is basically working correctly.

Your assignment code should be in the package **`week11`**. You will be able to submit your assignment using the command:

```
asgn-submit
```

Do *not* work with anyone who is not in your group, or ask the demonstrators for help with the assignment. However, feel free to raise any questions you may have with Iain or Michael.

All submissions will be checked for similarity.

Marking

This assignment is worth 20% of your final mark for COSC 241. It is possible to get full marks. In order to do this you must write correct, well-commented code which meets the specifications, and a sensible, well-presented report.

Marks are awarded for your program based on both implementation (9%) and style (5%), and for your report (6%). It should be noted however that it is very bad to style to have an implementation that doesn't work.

In order to maximise your marks please take note of the following points:

- Your code should compile without errors or warnings.
- Your program should use good Java layout (use the **`checkstyle`** tool to check for layout problems).
- Avoid duplicated code. Strive for conciseness, but not at the expense of clarity.

- Make sure each file is clearly commented.
- Most of your comments should be in your method headers. A method header should include:
  - A description of what the method does.
  - A description of all the parameters passed to it.
  - A description of the return value if there is one.
  - Any special notes.
- Your report should be grammatically correct, free of typos, and formatted appropriately. You should imagine that you are writing it for a client who is considering putting some of this functionality into an app. As noted earlier, explanations are at least as important as answers.

Part of this assignment involves you clarifying exactly what your program is required to do. Don't make assumptions, only to find out that they were incorrect when your assignment gets marked.

If you have any questions about this assignment, or the way it will be assessed, please see Iain or send an email to ihewson@cs.otago.ac.nz.