Word Salad

Problem description

Your Aunt Maude was fond of making what she called "word salad". She did this by taking blocks of text, cutting it up in various ways, and then reassembling it (again in various ways). It seemed a harmless enough diversion until, dealing with her estate, you realised that she'd turned all of her household documents into salads before saving them. Since these documents might contain a lot of important information (such as the location of the main water cut-off valve at her crib) you'd like to reconstruct them if at all possible.

First though, you'll need to implement the mechanisms she used to create her salads. Since you don't want to spend too much time with scissors cutting out words and gluing them down on paper (before possibly cutting them out again and rearranging those) you've decided to try and automate the process.

It seems that Aunt Maude had three ways by which she cut up blocks of text:

Distribute To distribute words into k blocks (k is a positive integer) you put the first word in the first block, the second in the second, ..., the k^{th} in the k^{th} and then start again (like dealing cards). If you run out of words you just stop (so the first few blocks may wind up with an extra word in them).

Chop To chop a block of words into k blocks you just divide them up into k nearly equal pieces. So if there were 17 words and 5 blocks the block lengths would be 4, 4, 3, 3, 3 and the first block would consist of the first four words, the second of the next four words, and so on.

Split Splitting is the most complicated construction. To split with a count of k you form the first block as if it were the first block when you were distributing into k blocks but you keep the remaining words in their original order – then you split the remainder with a count of k continuing until there are no words left. For instance to split 17 words with parameter 5 you'd wind up with blocks of sizes: 4, 3, 2, 2, 2, 1, 1, 1, 1. That's because the first block comes as if distributing (so has four words in it). There are then 13 words left and we're splitting with parameter 5 so the next block has three words in it (since $2 < 13/5 \leqslant 3$). Then the 10 words remaining are split with parameter 5 so we get a block of size two, and so on.

For example consider the text:

```
The quick brown fox jumps over the lazy dog
```

Distributing it into three groups gives:

```
[The, fox, the], [quick, jumps, lazy], [brown, over, dog]
```

Chopping it into three groups gives:
`[The, quick, brown], [fox, jumps, over], [the, lazy, dog]`

And splitting it with a count of three gives:
`[The, fox, the], [quick, over], [brown, dog], [jumps], [lazy]`

She put together her salads in three ways:

Merge To merge a sequence of blocks you take the first word from the first block then the first word from the second block, continuing until you run out of blocks. Then you take the second word from the first block and so on. If any block becomes empty at any point you just skip it in subsequent rounds. The result of merging the distribution of some text is the original text.

Join To join a sequence of blocks you just string them together one after the other. If you join the result of chopping some text you get the original text.

Recombine To recombine a sequence of blocks you reverse the steps that you took to split it. So recombining the split performed above would first bring together the single words "jumps" and "lazy" and then insert "brown" and "dog" three positions apart giving:
`brown jumps lazy dog`
then inserting "quick" and "over" with a count of three gives:
`quick brown jumps over lazy dog`
then inserting "The", "fox", and "the" with a count of three gives:
`The quick brown fox jumps over the lazy dog`

The task

You will be provided with a file **`WordSalad.java`** which includes an implementation of a singly linked list of **`String`**. There are four methods that you are required to implement: `distribute`, `merge`, `chop`, and `join`. In addition to this there are two bonus methods that you should attempt to implement: `split` and `recombine`. This means that it is (just) possible to get full marks for the assignment if you do a good enough job implementing the first four methods. However, if you implement the last two methods as well you could get some bonus marks to offset any places where you might have previously lost marks. The signatures of the methods that are to be implemented are:

Required

- **`public WordSalad[] distribute(int k)`** Distributes **`this`** into an array of k instances of **`WordSalad`** objects.

- **`public WordSalad[] chop(int k)`** Chops **`this`** into an array of k instances of **`WordSalad`** objects.
- **`public static WordSalad merge(WordSalad[] blocks)`** Merges the given blocks into a single **`WordSalad`** object.
- **`public static WordSalad join(WordSalad[] blocks)`** Joins the given blocks into a single **`WordSalad`** object.

Highly recommended but not essential

- **`public WordSalad[] split(int k)`** Splits **`this`** into an array of **`WordSalad`** objects.
- **`public static WordSalad recombine(WordSalad[] blocks, int k)`** Recombines the given blocks into a single **`WordSalad`** object.

You can write as many additional helper methods as you would like to in order to complete your `WordSalad` class.

In addition to the file `WordSalad.java` we have also provide an application class `SaladApp.java` that you should use to test and run your `WordSalad` class. There are also two small files `words.txt` and `letters.txt` containing some sample test input.

Copy the provided files from the /home/cshome/coursework/241/pickup/asgn directory like this

```
cp -r /home/cshome/coursework/241/pickup/asgn ~/241/09
```

You should compile your program from within the `~/241/09` directory like this:

```
javac -d .  -Xlint *.java
```

and you can run your program like this:

```
java week09.SaladApp
```

Group work and Submission

For this assignment we require you to work in groups of three people. You may select your own group and inform us of your choice by 4pm Wednesday April 11^{th}.

Send an email to ihewson@cs.otago.ac.nz letting us know the names and University user codes of all students in your group. Any students who don't select their own group will be assigned to one by us and informed via email. Groups will be assigned in a

pseudo-random manner. Students will be teamed up with others who have completed a similar amount of internal assessment.

Do *not* work with any other student who is not in your group, or ask the demonstrators for help with the assignment. However, feel free to come and discuss any questions you may have with Iain Hewson.

By week 7 of the semester, you will be able to check your assignment against some basic tests by running the command:

```
WEEK=09 241-check
```

Your assignment code should be in the package **`week09`**. You will be able to submit your assignment using the command:

```
asgn-submit
```

Any assignments submitted after the due date and time will lose marks at a rate of 10% per day late.

All submissions will be checked for similarity.

Each student in a group must submit their work (which should be identical to the other group members) using **`asgn-submit`**. When submitting your assignment you will be asked to rate the contribution of you and your group members using the scale:

1. I did none of the work. My group members did it all.
2. I did a little. My group members did most of the work.
3. We contributed evenly.
4. I did most of the work. My group members did a little.
5. I did it all. My group members did none of the work.

If you have serious concerns about unequal contributions within your group, please let Iain know.

Marking

This assignment is worth 10% 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.

Marks are awarded for your program based on both implementation (7%) and style (3%). It should be noted however that it is very bad to style to have an implementation

that doesn't work. Partial marks will be given for a partially complete implementation.

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).
- 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.

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.