JDK 8 Lambdas and Streams MOOC: Lesson 3 Homework

The homework this week is in three parts. The goal of the exercise is to give you a better understanding of the differences in performance of sequential and parallel stream processing.

Part 1: A file called "words" is provided containing a large collection of words. A template file, RandomWords.java, is also provided. In the constructor you need to read all the words (which are one per line) from the source file into a list (remember to use a stream to do this).

You also need to write the body of the createList() method. This generates a list of the size specified as a parameter selecting words at random from the list read in the constructor. HINT: You can use the ints() method of the Random class, which returns a stream of random integers. You can specify the size of the stream using a parameter.

Part 2: In order to provide a relatively compute-intensive task we will calculate the Levenshtein distance between two strings. This distance is a measure of how many changes need to be made to the first string to convert it to the second string (see https://en.wikipedia.org/wiki/Levenshtein_distance for a more detailed description of what this involves). Since we're focusing on streams programming, a source file, Levenshtein.java containing a lev() function is provided that will calculate the distance for you.

A second template file, Lesson3.java, is provided. This contains the code necessary to measure the time taken to execute the code of the get() method of a Supplier (as you will see in the main() method this is simple to do with a lambda expression).

Your task is to write the necessary code in the computeLevenshtein method to calculate the distances between each pair of strings in the wordList using the streams API. You will need to process this sequentially or in parallel based on the flag passed as a parameter.

HINT: Although the aim of this exercise is to use Lambdas and Streams in place of explicit loops, there are cases when for loops still make perfect sense. Use IntStream to create a stream of indices for one dimension of the array so this can be left sequential or converted to parallel, as required. However, feel free to use a for loop when processing each value to generate the distance from the other dimension of the array.

Try modifying the size of the wordList to see what impact this has on the performance, and in particular, how the difference between sequential and parallel performance is affected by the input size.

Part 3: As another demonstration of the differences in sequential and parallel stream processing there is a second method, processWords() for you to implement. Take the list of strings passed and process these using a sequential or parallel stream as required to generate a new list. Start by simply sorting the strings then experiment with adding things like mapping to lower or upper case, filtering out certain words (such as those beginning with a certain letter). See what impact adding distinct() to the stream has. For each of these vary the size of the input list to see how this affects the performance. Find the threshold below which sequential will give a faster answer than parallel.

NOTE: Because this is a micro-benchmark the results will not give you a totally accurate measure of the difference between sequential and parallel stream operations. It is good enough, however, to illustrate the differences between the two forms of stream processing in this case.