Exercises week 7

Last update: 2022/10/10

The goals of this week are to enable you to apply Java streams and parallelize Java streams and recognize possible applications of functional programming and lazy evaluation.

The goals are:

- Apply and examine a number of Java stream concepts including: stream sources, intermedialte operators and terminal operators
- Apply lambda expressions in Java.
- Apply benchmarking to the students own algorithms/methods written in Java

Not mandatory
If you are already comfortable with Java lambdas, you may skip this exercises. If you are not, please try to solve it and turn in your solution.

Exercise 7.1 You may use this Java skeleton as a starting point of the exercise.

```
import java.util.function.Function;
class LambdaExample {
  public static void main(String[] args) { new LambdaExample(); }
  public LambdaExample() {
    System.out.println("I: "+increment(f));
    //To be filled in
  }
  Function<Integer, Integer> f = (x) -> x+1;
}
```

You can find the code above in `Week07/code-exercises ...  /LambdaExample.java`.

1. Write the (missing) code for the `increment` function to make the output of the `LambdaExample`: I: 9
2. Change the code in `LambdaExample` so that the function `f` multiplies with 5 (instead of incrementing).
3. These code snippets are from `Benchmark.java` and `Benchmarkable.java` in `Week05/exercises-code ...  /...`:

```
---- Benchmark.java
import java.util.function.IntToDoubleFunction;
...
public Benchmark() {
...
    Mark6("multiply", i -> multiply(i));
    Mark6("multiply", Benchmark::multiply);
...
}
public static double Mark6(String msg, IntToDoubleFunction f) {
    ...
    dummy += f.applyAsDouble(i);
}
---- Benchmarkable.java
import java.util.function.IntToDoubleFunction;
public abstract class Benchmarkable implements IntToDoubleFunction {
  public void setup() { }
  public abstract double applyAsDouble(int i);
}
```

Write a short explanation of what happens in the two lines (emphasize explaining the two lambda expressions):

```
Mark6("multiply", i -> multiply(i));
Mark6("multiply", Benchmark::multiply);
```

4. Write a new version of `Mark6` called `Mark6int` that will *only* accept measuring functions that takes an integer argument and delivers an integer result (e.g. `intcountSequential` in Exercise 7.2). Like `Mark6`, `Mark6int` should measure the running time of the function given as the second argument.

```
public static double Mark6int(String msg, ???) {
  //To be filled in
}
```

Exercise 7.2 Consider this program that computes prime numbers using a while loop:

```
class PrimeCountingPerf {
  public static void main(String[] args) { new PrimeCountingPerf(); }
  static final int range= 100000;
  //Test whether n is a prime number
  private static boolean isPrime(int n) {
    int k= 2;
    while (k * k <= n && n % k != 0)
      k++;
    return n >= 2 && k * k > n;
  }
  // Sequential solution
  private static long countSequential(int range) {
    long count= 0;
    final int from = 0, to = range;
    for (int i=from; i<to; i++)
      if (isPrime(i)) count++;
    return count;
  }
  // Stream solution
  private static long countStream(int range) {
    long count= 0;
    //to be filled out
    return count;
  }
  // Parallel stream solution
  private static long countParallel(int range) {
    long count= 0;
    //to be filled out
    return count;
  }
  // --- Benchmarking infrastructure ---
  public static double Mark7(String msg, IntToDoubleFunction f) {    ...  }
  public PrimeCountingPerf() {
    Mark7("Sequential", i -> countSequential(range));

    Mark7("IntStream", i -> countIntStream(range));

    Mark7("Parallel", i -> countParallel(range));

    List<Integer> list = new ArrayList<Integer>();
    for (int i= 2; i< range; i++){ list.add(i); }
```

```
    Mark7("ParallelStream", i -> countparallelStream(list));
  }
}
```

You may find this in `Week07/code-exercises ... /PrimeCountingPerf.java`. In addition to counting the number of primes (in the range: 2..range) this program also measures the running time of the loop. Note, in your solution you may change this declaration (and initialization) `long count= 0;`

Mandatory

1. Compile and run `PrimeCountingPerf.java`. Record the result in a text file.
2. Fill in the Java code using a stream for counting the number of primes (in the range: 2..range). Record the result in a text file.
3. Add code to the stream expression that prints all the primes in the range 2..range. To test this program reduce range to a small number e.g. 1000.
4. Fill in the Java code using the intermediate operation `parallel` for counting the number of primes (in the range: 2..range). Record the result in a text file.
5. Add another prime counting method using a `parallelStream` for counting the number of primes (in the range: 2..range). Measure its performance using `Mark7` in a way similar to how we measured the performance of the other three ways of counting primes.

Exercise 7.3 This exercise is about processing a large body of English words, using streams of strings. In particular, we use the words in the file `app/src/main/resources/english-words.txt`, in the exercises project directory.

The exercises below should be solved without any explicit loops (or recursion) as far as possible (that is, use streams).

Mandatory

1. Starting from the TestWordStream.java file, complete the `readWords` method and check that you can read the file as a stream and count the number of English words in it. For the `english-words.txt` file on the course homepage the result should be 235,886.
2. Write a stream pipeline to print the first 100 words from the file.
3. Write a stream pipeline to find and print all words that have at least 22 letters.
4. Write a stream pipeline to find and print some word that has at least 22 letters.
5. Write a method `boolean isPalindrome(String s)` that tests whether a word `s` is a palindrome: a word that is the same spelled forward and backward. Write a stream pipeline to find all palindromes and print them.
6. Make a parallel version of the palindrome-printing stream pipeline. Is it possible to observe whether it is faster or slower than the sequential one?
7. Make a new version of the method `readWordStream` which can fetch the list of words from the internet. There is a (slightly modified) version of the word list at this URL: `https://staunstrups.dk/jst/english-words.txt`. Use this version of `readWordStream` to count the number of words (similarly to question 7.2.1). Note, the number of words is *not* the same in the two files !!
8. Use a stream pipeline that turns the stream of words into a stream of their lengths, to find and print the minimal, maximal and average word lengths.
   Hint: There is a simple solution using an operator examplified on p. 141 of Java Precisely (included in the readings for this week).

Challenging

9. Write a stream pipeline, using method `collect` and a `groupingBy` collector from class Collectors, to group the words by length. That is, put all 1-letter words in one group, all 2-letter words in another group, and so on, and print the groups.

   Use another overload of `groupingBy` to compute (and then print) the number of 1-letter words, the number of 2-letter words, and so on. (Hint: you may want to consult *Java Precisely*).

Challenging

Exercise 7.4 In this exercise we will use parallel array operations to experimentally investigate the assertion that the count $\pi(n)$ of prime numbers less than or equal to n is proportional to $n/\ln(n)$, where $\ln(n)$ is the natural logarithm of n. More precisely, the ratio $\pi(n)/(n/\ln(n))$ converges to 1 for large n. This is known as the prime number theorem.

1. Create an `int` array `a` of size N, for instance for $N =$ 10,000,001. Use method `parallelSetAll` from utility class `Arrays` to initialize position `a[i]` to 1 if `i` is a prime number and to 0 otherwise. You may use method `isPrime` from the other prime number related examples.
2. Use method `parallelPrefix` from utility class `Arrays` to compute the prefix sums of array `a`. After that operation, the new value of `a[`i`]` should be the sum of the old values `a[0..`i`]`. Therefore, the new value of `a[`i`]` is the count of prime numbers smaller than or equal to `i`, that is, $\pi(i)$. For instance, the value of `a[10_000_000]` should be 664,579.
3. Use a for-loop to print the ratio between `a[`i`]` and $i/\ln(i)$ for 10 values of `i` equally spaced between $N/10$ and N.