Exercises Exam Example

Last update: 2022/12/01

This is an example of an exercise that could have been given at the written exam. It is very similar to Exercise 7.2 from week07, but the questions are fomulated a bit differently to clarify what must be included in a solution.

After the exercise is an example of a solution. However, this is *only an example*. Most likely, acceptable/good solutions could easily look quite different. The most important is that all parts of the section **Hand-in** are addressed.

In the directory `code-exercise-solution` is an example of code accompanying the solution below. Again, there could be many other acceptable/good ways of writing the code.

All the exercises have an indication of a number of points. The points for all the exercises in the exam set will add up to 100.

Exercise 1.1 (N points)

Consider this program `PrimeCountingPerf.java` 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));
    //This will run a performance measurement of the sequential search
    //TO-DO add code to handle the three questions in the exercise
  }
}
```

You may find the bcode shown above in `PrimeCountingPerf.java` (same directory as this pdf). 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;`

1. Fill in the missing Java code in (`countStream`) using a stream for counting the number of primes (between 2 and `range`). Your code should print the result (no. of primes). Furthermore add code to measure and report the running time of countStream using `Mark7`.

   Hand in: The complete code for the method `countStream`, an explanation of how it works, the result from running it and a short explanation of what the output shows.

2. Fill in the missing Java code in (`countParallel`) using a parallel stream for counting the number of primes (between 2 and `range`). Your code should print the result (no. of primes). Furthermore add code to measure and report the running time of `countParallel` using `Mark7`.

   Hand in: The complete code for the method `countParallel`, an explanation of how it works, the result from running it and a short explanation of what the output shows.

3. Make a table with the results of the performance measurement using `Mark7` of the three different methods for counting primes. Discuss and compare the results, e.g. explain any diferences.

   Hand in: The table with the results and a short explanation (a few lines) of what the table shows.

Solution example

Question 1.1

Code for `countStream`:

```
private static long countStream(int range) {
  long count= IntStream.range(2, range)
    .filter(i -> isPrime(i))
    .count();
  return count;
}
```

This code creates a stream of integers between 2 and `range`, this stream is then filtered using the intermediate operation `filter(i -> isPrime(i))` which gives a stream containing only numbers for which `isPrime` returns true. The terminal operation `count` counts the number of elements in the stream it receives and returns the result in the variable `long count`.

This is the code used for printing the number of primes found by `countStream` and for mesuring its running time using `Mark7`:

```
  System.out.println("countStream found: "+ countStream(range) + "
     primes");
  Mark7("IntStream", i -> countStream(range));
```

and the output from running it is:

```
countStream found: 9592 primes
IntStream                        4984970.0 ns    57126.42          64
```

In addition to the number of primes found, the output shows the mean running on nanoseconds, the standard deviation and the number of times the tune-time measurement has been done (by Mark7).

Question 1.2

Code for `countParallel`:

```
private static long countParallel(int range) {
  long count= IntStream.range(2, range)
    .parallel()
    .filter(i -> isPrime(i))
    .count();
  return count;
}
```

The only difference to `countStream` is the addition of the intermediate stream operation `parallel` which enables the Java runtime system to run the intermediate operations (filter etc) in parallel. On machines with multiple cores we cannot control how many of these are used by `parallel`.

This is the code used for printing the number of primes found by `countParallel` and for mesuring its running time using `Mark7`:

```
  System.out.println("countParallel found: "+ countParallel(range) + "
     primes");
  Mark7("Parallel", i -> countParallel(range));
```

and the output from running it is:

```
countParallel found: 9592 primes
Parallel                         1363956.6 ns    11552.35         256
```

In addition to the number of primes found, the output shows the mean running on nanoseconds, the standard deviation and the number of times the tune-time measurement has been done (by Mark7).

Question 1.3

This is the complete table of results from running the three prime counting methods:

```
countSequential found: 9592 primes
Sequential                      4982763.4 ns   70028.66         64
countStream found: 9592 primes
IntStream                       4984970.0 ns   57126.42         64
countParallel found: 9592 primes
Parallel                        1363956.6 ns   11552.35        256
```

The three methods report the same number of primes (9592) which indicates that they correctly compute the required task.

The running time of `countSequential` and `countStream` are almost the same. Repeating the measurement, sometimes `countSequential` comes out (marginally) faster than `countStream`. At other times, it is the other way around. So my conclusion is that there is no performance difference between the two on my hardware.

However, `countParallel` consistently comes out around 3.6 - 3.7 times faster than `countSequential` and `countStream`. My laptop has 4 cores, which apparently are used quite effectively by the Java run-time system.