Using the Java Concurrency Framework

In this recitation you will use the Java concurrency framework to parallelize, and hopefully improve the performance of, a numeric computation. Specifically, in `SequentialMersennePrimes` we have provided a sequential algorithm that finds Mersenne primes. A *Mersenne prime* is a prime number of the form $2^p - 1$, where p is itself a prime number.

The sequential algorithm to find Mersenne primes is straightforward. It iterates over all prime numbers p and, for each prime p, computes and checks the primality of the Mersenne number $2^p - 1$. In practice, checking the primality of the Mersenne numbers is a very expensive operation because, even for relatively small prime numbers p, the Mersenne numbers $2^p - 1$ are very large, with thousands of digits.

You should parallelize the Mersenne prime-finding algorithm by checking several candidate Mersenne numbers in parallel. In its most basic form this goal is simple, but it can be subtly tricky to create a parallel implementation that is functionally equivalent to our sequential implementation. The subtle challenge is to output results in increasing order, which might be different from the order the Mersenne primes are found (due to the unpredictable scheduling of concurrent threads).

This recitation consists of two tasks: you will first parallelize the Mersenne prime-finding algorithm to produce results in any order, and (time permitting) then attempt to functionally match our sequential implementation by producing Mersenne primes in increasing order.

Recitation tasks

1. In `ParallelMersennePrimes1`, parallelize our Mersenne prime-finding algorithm to find 20 Mersenne primes in any order. To do this:

   (a) Create a separate task (a `Runnable` or `Callable`) to check the primality of a candidate Mersenne number.

   (b) Use the `Executors` utility class to get an `ExecutorService`.

   (c) Create and submit your tasks (one task per Mersenne number) to the `ExecutorService`.

   The biggest challenge of this sub-part is to approximately synchronize the generation of the primality-checking tasks to their execution; you don't want to generate and submit millions of tasks to an `ExecutorService` if only a few dozen (or few hundred) tasks are needed to find 20 Mersenne primes.

2. In `ParallelMersennePrimes2`, parallelize our Mersenne prime-finding algorithm to output Mersenne primes in sequential order, exactly matching the functionality of our sequential implementation. To guarantee that Mersenne primes are output in sequential order (rather than the order they are found) you will need to implement some producer/consumer workflow, where some thread or threads produce results to be output and another thread sequentially outputs the results in increasing order. In addition to the basic tools of the Java concurrency framework above, you might need concurrent data structures such as the Java `ArrayBlockingQueue`.