# 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: finding Mersenne primes. A *Mersenne prime* is a prime number of the form $2^p - 1$, where $p$ is itself a prime number.

To sequentially find Mersenne primes, you can iterate over all prime numbers $p$ and, for each prime $p$, compute and check the primality of the Mersenne number $2^p - 1$. In practice, checking the primality of the Mersenne numbers is an expensive operation because, even for relatively small prime numbers $p$, the Mersenne numbers $2^p - 1$ are very large, with thousands of digits.

In `SequentialMersennePrimes` we provide an implementation of the sequential algorithm described above. You should inspect and understand that implementation and then parallelize the algorithm by checking several candidate Mersenne numbers in parallel, in `ParallelMersennePrimes`. This goal is simple to describe, but it is subtly tricky to create a parallel implementation that is equivalent to the sequential implementation. We recommend:

1. Given a single candidate Mersenne number, create a task to check its primality. We recommend that this task implements the `Callable` interface, with the `call` method returning the Mersenne number if it is prime and `null` otherwise.
2. Use the `Executors` utility class to get an appropriate `ExecutorService` to execute the tasks you define above.
3. Use a bounded blocking queue (such as an `ArrayBlockingQueue`) to store results for Mersenne candidates yet-to-be-checked.
4. Create and execute a task to produce Mersenne candidates. It should essentially be an infinite loop, with each execution of the loop being:
   (a) Generate the next candidate Mersenne number and create a task for it.
   (b) Submit the task to the `ExecutorService`.
   (c) Insert each task's result (probably some `Future` object from the `ExecutorService`) into the bounded blocking queue.
5. In the main thread, remove results from the bounded blocking queue, counting and printing any Mersenne primes that have been found.

The above approach is designed to overcome two key challenges. First, generating candidate Mersenne numbers is fast, but checking the primality of a candidate number is slow. The bounded blocking queue approximately synchronizes the generation of candidates to primality-checking, which avoids creating millions of tasks if only a few dozen (or few hundred) tasks are needed to find 20 Mersenne primes. The second challenge is that, due to unpredictable scheduling of concurrent threads, Mersenne primes might be found in a different order than the candidates are generated; this could cause the results to be printed in non-sequential order. The above approach solves this problem by using a single thread to generate candidates and a single thread (the main thread) to print results. The bounded blocking queue stores results in increasing order even though the results may be found in non-increasing order.