**Exercises: Parallel Programming with Fork/Join**

This is a tough exercise! To simplify it slightly, start by copying the Primes class from the fork-join-

exercises project (*not* the fork-join project).

**1.** Make an ordinary (serial) method that, given a boolean[], will loop down the array and mark each

entry as true or false, depending on whether that index is prime. For example, array[0] and array[1]

should be false (0 and 1 are not prime), array[2] and array[3] should be true (2 and 3 are prime),

array[4] should be false (4 is not prime), array[5] should be true (5 is prime), and so forth.

Notes:

• Use Primes.isPrime(number) to test whether a number is prime. This method is already built in to

the Primes class that you copied from the fork-join-exercises project.

• To simplify the later parallel version, break your code into two methods, one that takes the whole

array and one that takes the array and two indices.

public static void markPrimesSerial(boolean[] primeFlags,

int lowerIndex, int upperIndex) {

// One simple line of code that uses Primes.isPrime

}

public static void markPrimesSerial(boolean[] primeFlags) {

markPrimesSerial(primeFlags, 0, primeFlags.length-1);

}

**2.** Test your code on boolean arrays of different sizes. For a 1,000 element array, it should find 168

primes, with 997 as the largest. For a 10,000 element array, it should find 1,229 primes, with 9,973

as the largest. For a really large test case, for a 10,000,000 element array, it should find 664,579

primes, with 9,999,991 as the largest. Your testing might be easier if you make a method that, given

a boolean[] with the entries marked, produces a List<Integer> of the primes.

**3.** Make a ParallelPrimeMarker class that extends RecursiveTask. Note that, since you have no com-

bining operation (you just mark each entry separately), you will extend RecursiveTask<Void>, the

return type of compute will be Void, and you will just return null at the bottom of compute. Since

the prime-testing operation is moderately expensive, use a small value like 10 as the parallel cutoff.

Even so, your code will be*very* similar to that of the ParallelArraySummer (the first example in the

lecture), so be sure to have that code in front of you when writing ParallelPrimeMarker.

**4.** Make a parallel version of markPrimes that uses the ParallelMarker class.

**5.** Verify that the parallel and serial versions give the same results. If you made the method suggested

in problem 2 (producing a List<Integer> from the marked boolean[]), you can produce two Lists

and check that the number of entries and the value of the last entry are the same.

**6.** Compare the timing of the two approaches. On my 4-core machine, the serial version takes 3 to 4

times longer for almost any problem size from 1,000 to 10,000,000.