ESIR SPP – TP5 & 6 (Attention: **TP noté**)

Préparations:

- Assurez vous d'avoir accès au site Moodle du cours sur votre ENT (https://ent.univ-rennes1.fr, puis onglet "La Formation", "Cours en ligne").
- Vérifiez que vous avez bien accès à la page de soumission du TP 5 & 6.

Soumission et notation:

- Il vous est demandé de travailler en **binôme** pour ce TP.
- Vous devez individuellement **soumettre le code source Java des exercices du TP** et **un court rapport indiquant vos réponses aux questions** avant la **date butoir** indiquée sur Moodle. Conseil: N'attendez pas cette date pour soumettre votre solution!
- Veuillez noter dans votre soumission **le binôme** avec lequel vous avez travaillé.
- Le TP **sera noté en classe** durant **la séance de TP 7** à partir du code source que vous aurez soumis.
- Pour recevoir une note pour devez impérativement: (i) **soumettre votre solution** avant la date butoir; et (ii) **être présent lors du TP 7**, et ce pour les deux étudiants de chaque binôme.

Note: Les étudiants d'un même binôme peuvent recevoir des notes différentes en fonction des réponses données lors de la notation.

Exercise 1: Eratosthenes' Sieve

The goal of this exercise is to implement a parallel version of Eratosthenes' Sieve.

First read about Eratosthenes' Sieve at http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

The following is a possible sequential pseudo code for Eratosthenes' Sieve.

```
Input: an integer n > 1
Let A be an array of Boolean values, indexed by integers 2 to n, initially all set to true.
for i = 2, 3, 4, ..., √n :
  if A[i] is true:
    for j = i^2, i^2+i, i^2+2i, ..., n:
      A[j] := false
Now all i such that A[i] is true are prime.
```

Task 1

First look at the above pseudo code and answer the following questions:

- Why does the first loop only runs up to √n? What should you do if √n is not an integer (i.e. n is not a square)?
- Why does the inner loop start at i^2 and not 0 or i?

Task 2

As a preparation for the rest of the lab, implement a JUnit test that checks whether an algorithm that returns all the prime number smaller that a fixed integer is correct.

Note(s):

- You will need to decide how you will present Eratosthenes' Sieve as a Java class. Ideally your choice should be applicable to any prime number algorithm, i.e. be independent from the detail of the algorithm used.
- You might want to use the list of prime numbers provided in the lab's resources.

Task 3

As a first step towards your parallel program, implement a sequential version of Eratosthenes' Sieve. Check that this sequential version works with using your JUnit test program.

Note(s):

- Do no discard this sequential implementation. We will reuse it for benchmarking.

Task 4

Building on your sequential version, you will now develop a parallel version of Eratosthenes' Sieve. The idea is to parallelise the inner loop of the pseudo code " for j = i^2, i^2+i, i^2+2i, ..., n", by distributing the exploration of the range "i^2, i^2+i, i^2+2i, ..., n" among k threads. (k is a parameter of your implementation.)

To avoid the cost of repeatedly creating threads, the k threads of your program should be created outside the *outer loop* of the algorithms, and be reused in each iteration of the outer loop.

The pseudo code of the main thread of your program should look like the following (with the same notation as above):

```
create and launch k worker threads. Each worker thread has access to the array A. The k threads start by blocking on a synchronisation object.
for i = 2, 3, 4, ..., √n :
  if A[i] is true:
    distribute work among the k worker threads (*)
    unblock the k worker threads (using the appropriate action)
    wait for all worker threads to complete their iteration (**)
terminate all workers threads
```

Notes:

- There are 2 points of synchronisation in this code: The worker threads must wait for the main thread to initialise the work to be done (*); and the main thread must then wait for the worker threads to finish their share of the work (**). In both cases, you will need a shared synchronisation object to implement the synchronisation. You might want to use CyclicBarriers. (Optional question: Why would a CountDownLatch object be problematic?)
- The array A is modified concurrently by all worker threads. Do you need to protect it using a lock or a monitor?
- To distribute work among the worker threads, you essentially need to tell each of them which range to cover in the current iteration. Do this by having each thread store the range it should cover in appropriate attributes of your Thread or Runnable subclass. Then, have the main thread set this range for each worker thread before unblocking the worker thread.
- Use the interruption mechanism to terminate all the workers threads. You will need to write an appropriate try-catch block in your implementation of worker thread.

Once you have implemented a parallel version of Eratosthenes' Sieve, check this version with your JUnit test for different value of k (k=1, 2, 3, 4, 10).

Task 5

The goal of this final task is to compare the performance of your sequential and parallel versions of Eratosthenes' Sieve.

Measure the time taken to compute all prime numbers below 500,000 with:

- Your original sequential version;
- Your parallel version with the number of threads varying from 1 to 10.

Chart the results on a graph. (You should run your experiments multiple times to obtain representative averages. Would you know how to compute a confidence interval on these values?)

Repeat the same measurement for all primes below 1,000,000; 2,000,000; 4,000,000, and draw the corresponding curves on the same chart. How do you interpret your results?

Task 6 (Bonus points)

If you have some time left you may want to analyse the impact of the following two optimisations:

- *Speeding up the inner loop:* Your inner loop uses a step of i. How could you modify it to use a step of 2i instead of i? (Hint: You will need to consider 2 as a separate case.)

- *Reducing memory overheads:* your implementation uses an array of Boolean (say isPrime[]) to remember which numbers are prime. This approach uses a lot of space: all even numbers (so almost half of the array) except 2 will be set to false. Since we already know that (i) 0 and 1 are not prime (by definition), and that (ii) all even numbers except 2 (i.e. 4,6,8, …) are not prime either, a more efficient approach consists in using an array that is half as big, and only maintaining a Boolean for odd numbers above 3. In other words, you could use isPrime[i] to indicate whether k = 2*i + 3 is prime or not.

Marking scheme:

Task 1: 3 points

Task 2: 2 points

Task 3:

- Your sequential version passes your JUnit test (provided this test is correct): 1 points
- You are able to explain how your sequential version works: 3 points

Task 4:

- Your parallel version passes your JUnit test (provided this test is correct): 2 points
- You are able to explain how your parallel version works: 4 points

Task 5: 3 points

Task 6: 2 points