CS2030 Programming Methodology
Semester 2 2018/2019

25 October 2019
Problem Set #8 Suggested Answers
Parallel Streams

1. What is the outcome of the following stream pipeline?

```
Stream.of(1, 2, 3, 4)
    .reduce(0, (result, x) -> result * 2 + x);
```

What happens if we parallelize the stream? Explain.

Running the stream sequentially gives 26 since the pipeline evaluates

```
((((0 * 2 + 1) * 2 + 2) * 2 + 3) * 2 + 4)
```

A possible parallel run (with output from each `reduce` operation) gives 18.

```
0 * 2 + 4 = 4 : ForkJoinPool.commonPool-worker-370
0 * 2 + 3 = 3 : main
0 * 2 + 2 = 2 : ForkJoinPool.commonPool-worker-441
0 * 2 + 1 = 1 : ForkJoinPool.commonPool-worker-299
3 * 2 + 4 = 10 : main
1 * 2 + 2 = 4 : ForkJoinPool.commonPool-worker-299
4 * 2 + 10 = 18 : ForkJoinPool.commonPool-worker-299
18
```

Notice that reduction with the identity value `0` happens for all four stream elements. This is followed by reducing `(1, 2)` to give `4`, and reducing `(3, 4)` to give `10`. Finally, reducing `(4, 10)` gives `18`.

The above stream cannot be parallelized because `2 * result + x` is not associative, i.e. the order of reduction matters.

2. In this question, you are to re-solve the maximum disc coverage solution, testing each possible circle in parallel, by completing the method below using `Stream` and `Optional` without any loops or `null`s.

```
public static long solve(List<Point> points, double radius) {
    return ..
}
```

The method takes in a list of `Point` objects and a `distance` value, and returns the maximum number of points from the given list contained in any circle with radius `radius`.

Your solution should consists of a single return statement with the method calls chained together. You can assume that the `Point` class and the `Circle` class has been written as in your Lab #1.

Hint: you will also find the `product` method you wrote from Problem Set #7 and the `Pair` class that you used in Lab #7 useful.

```
import java.util.Optional;

public class Circle {
    private final Point centre;
    private final double radius;

    private Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    // change to return Optional
    public static Optional<Circle> getCircle(Point centre, double radius) {
        if (radius > 0) {
            return Optional.of(new Circle(centre, radius));
        } else {
            return Optional.empty();
        }
    }

    public boolean contains(Point p) {
        return this.centre.distanceTo(p) < radius + 1E-15;
    }

    @Override
    public String toString() {
        return "circle of radius " + radius + " centered at " + centre;
    }
}
```

```
import java.util.Scanner;
import java.util.List;
import java.util.Arrays;
import java.util.stream.Stream;
import java.util.function.BiFunction;
import java.util.Optional;

class Main {
    static class Pair<T,U> {
        T first;
        U second;

        Pair(T first, U second) {
            this.first = first;
            this.second = second;
        }

        public String toString() {
            return first +";" + second;

        }
    }
    public static <T, U, R> Stream<R> product(
            List<? extends T> list1,
            List<? extends U> list2,
            BiFunction<? super T, ? super U, ? extends R> func) {

        return list1.stream()
            .flatMap(x ->
                    list2.stream()
                    .map(y -> func.apply(x,y)));
    }

    static Point[] readPoints() {
        Scanner sc = new Scanner(System.in);
        Point[] points = new Point[sc.nextInt()];

        for (int i = 0; i < points.length; i++) {
            points[i] = new Point(sc.nextDouble(), sc.nextDouble());
        }
        return points;
    }

    // change to return Optional
    static Optional<Circle> createCircle(Point p, Point q, double radius) {
        double distPQ = p.distanceTo(q);
        if (distPQ < 2 * radius + 1E-15 && distPQ > 0) {
            Point c = p.midPoint(q);
            double cp = Math.sqrt(Math.pow(radius, 2) - Math.pow(p.distanceTo(c), 2));
```

```
            double theta = p.angleTo(q);
            System.out.println(theta + Math.PI/2 + " : " + cp);
            return Circle.getCircle(c.moveTo(theta + Math.PI / 2, cp), radius);
        } else {
            return Optional.empty();
        }
    }

    public static long solve(List<Point> points, double distance) {
        return product(points, points, (p,q) -> new Pair<>(p,q))
            .parallel()
            .map(t -> createCircle(t.first, t.second, distance))
            .flatMap(Optional::stream)
            .map(c -> points
                    .stream()
                   .filter(p -> c.contains(p))
                   .count())
            .max(Long::compare)
            .orElse(0L);
    }

    public static void main(String[] args) {
        Point[] points = readPoints();
        long maxDiscCoverage = solve(Arrays.asList(points), 1.0);
        System.out.println("Maximum Disc Coverage: " + maxDiscCoverage);
    }
}
```

3. By now you should be familiar with the Fibonacci sequence where the first two terms are defined by $f_1 = 1$ and $f_2 = 1$, and generation of each subsequent term is based upon the sum of the previous two terms. In this question, we shall attempt to parallelize the generation of the sequence.

   As an example, supose we are given the first $k = 4$ values of the sequence f_1 to f_4, i.e.

   $$1, 1, 2, 3$$

   To generate the next $k - 1$ values, we observe the following:

   $$\begin{array}{rclcrcl} f_5 & = & f_3 + f_4 & = & f_3 + f_4 & = & f_1 \cdot f_3 + f_2 \cdot f_4 \\ f_6 & = & f_4 + f_5 & = & f_3 + 2f_4 & = & f_2 \cdot f_3 + f_3 \cdot f_4 \\ f_7 & = & f_5 + f_6 & = & 2f_3 + 3f_4 & = & f_3 \cdot f_3 + f_4 \cdot f_4 \end{array}$$

   Notice that generating each of the terms f_5 to f_7 only depends on the terms of the given sequence. This actually means that generating the terms can now be done in parallel! In addition, repeated application of the above results in an exponential growth of the Fibonacci sequence.

   You are now given the following program fragment:

```
static BigInteger findFibTerm(int n) {
    List<BigInteger> fibList = new ArrayList<>();
    fibList.add(BigInteger.ONE);
    fibList.add(BigInteger.ONE);

    while (fibList.size() < n) {
        generateFib(fibList);
    }
    return fibList.get(n-1);
}
```

   (a) Using Java parallel streams, complete the `generateFib` method such that each method call takes in an initial sequence of k terms and fills it with an additional $k - 1$ terms. The `findFibTerm` method calls `generateFib` repeatedly until the n^{th} term is generated and returned.

   (b) Find out the time it takes to complete the sequential and parallel generations of the Fibonacci sequence for $n = 50000$.

```
import java.time.Instant;
import java.time.Duration;
import java.math.BigInteger;
import java.util.stream.Stream;
import java.util.stream.Collectors;
import java.util.List;
import java.util.ArrayList;

class Fib {
    static void generateFib(List<BigInteger> fibs) {
        int k = fibs.size();
        fibs.addAll(Stream
                .iterate(0, i -> i < k - 1, i -> i + 1)
                .parallel()
                .map(i ->
                    fibs.get(k-2).multiply(fibs.get(i)).add(
                        fibs.get(k-1).multiply(fibs.get(i+1))))
                .collect(Collectors.toList()));
    }

    static BigInteger findFibTerm(int n) {
        List<BigInteger> fibList = new ArrayList<>();
        fibList.add(BigInteger.ONE);
        fibList.add(BigInteger.ONE);

        Instant start = Instant.now();
        while (fibList.size() < n) {
            generateFib(fibList);
        }
        Instant stop = Instant.now();
        System.out.println(Duration.between(start,stop).toMillis() + "ms");

        return fibList.get(n-1);
    }

    public static void main(String[] args) {
        BigInteger result = findFibTerm(
                new java.util.Scanner(System.in).nextInt());
        System.out.println(result);
    }
}
```