CS2030S Recitation: Problem Set #9 Wk 11, Sem 1, 2022/23

1. Consider the `Maybe` class which is a simplification of `Probably` class:

```java
import java.util.function.Function;

class Maybe<T> {
  private static final Maybe<?> NULL = (Maybe<?>) new Maybe<>();

  private final T val;

  private Maybe() {
    this.val = null;
  }
  private Maybe(T val) {
    this.val = val;
  }

  public static <T> Maybe<T> some(T val) {
    if (val == null) {
      @SuppressWarnings("unchecked")
      Maybe<T> res = (Maybe<T>) NULL;
      return res;
    } else {
      return new Maybe<T>(val);
    }
  }

  public <R> Maybe<? extends R> flatMap(Function<? super T, ?
   extends Maybe<? extends R>> f) {
    return f.apply(this.val);
  }

  @Override
  public String toString() {
    if (this.val == null) {
      return "<>";
    } else {
      return "<" + this.val + ">";
    }
  }
}
```

Further consider the following two methods `foo` and `bar`:

```java
Optional<Integer> foo(Integer x) {
  if (x == null) {
    return null;
  }
  return Optional.ofNullable(x*2);
}

Maybe<Integer> bar(Integer x) {
  if (x == null) {
    return null;
  }
  return Maybe.some(x*2);
}
```

(a) Using these methods, test if `Optional` and `Maybe` obey the three Monad laws.

i. **Left Identity Law**

`Monad.of(x).flatMap(y -> f(y))` is equivalent to `f(x)`

ii. **Right Identity Law**

`monad.flatMap(y -> Monad.of(y))` is equivalent to `monad`

iii. **Associative Law**

`monad.flatMap(x -> f(x)).flatMap(x -> g(x))`
is equivalent to
`monad.flatMap(x -> f(x).flatMap(x -> g(x)))`

(b) Now test if `Optional` and `Maybe` obey the two Functor laws.

i. **Preserving Identity**

`functor.map(x -> x)` is equivalent to `functor`

ii. **Preserving Composition**

`functor.map(x -> f(x)).map(x -> g(x))`
is equivalent to
`functor.map(x -> g(f(x))`

(c) What can we deduce about `Optional` and `Monad` from Question 1a and 1b?

2. Consider the following stream pipeline:

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

   (a) What is the outcome of the stream pipeline above when run sequentially?

   (b) What happens if we parallelize the stream? Explain.

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

   Suppose 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:

$$\begin{aligned}
f_5 &= f_3 + f_4 &&= f_3 + f_4 &&= f_1 \times f_3 + f_2 \times f_4 \quad (1)\\
f_6 &= f_4 + f_5 &&= f_3 + 2f_4 &&= f_2 \times f_3 + f_3 \times f_4 \quad (2)\\
f_7 &= f_5 + f_6 &&= 2f_3 + 3f_4 &&= f_3 \times f_3 + f_4 \times f_4 \quad (3)
\end{aligned}$$

   Notice that generating each of the terms f_5 to f_7 only depends on the terms of the given sequence. In other words, to generate f_5 to f_7 we only need to know f_1 to f_4. In particular, generating f_6 is no longer dependent on f_5. 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:

```java
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, write 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 $n = 50000$.