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

```
1  import java.util.function.Function;
2
3  class Maybe<T> {
4    private static final Maybe<?> NULL = (Maybe<?>) new Maybe<>();
5
6    private final T val;
7
8    private Maybe() {
9      this.val = null;
10   }
11   private Maybe(T val) {
12     this.val = val;
13   }
14
15   public static <T> Maybe<T> some(T val) {
16     if (val == null) {
17       @SuppressWarnings("unchecked")
18       Maybe<T> res = (Maybe<T>) NULL;
19       return res;
20     } else {
21       return new Maybe<T>(val);
22     }
23   }
24
25   public <R> Maybe<? extends R> flatMap(Function<? super T, ?
      extends Maybe<? extends R>> f) {
26     return f.apply(this.val);
27   }
28
29   @Override
30   public String toString() {
31     if (this.val == null) {
32       return "<>";
33     } else {
34       return "<" + this.val + ">";
35     }
36   }
37 }
```

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

```
1  Optional<Integer> foo(Integer x) {
2    if (x == null) {
3      return null;
4    }
5    return Optional.ofNullable(x*2);
6  }
7
8  Maybe<Integer> bar(Integer x) {
9    if (x == null) {
10     return null;
11   }
12   return Maybe.some(x*2);
13 }
```

(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)`

Suggested Guide:

```
1 Optional.<Integer>ofNullable(null).flatMap(x -> foo(x))
2 $.. ==> Optional.empty
3 foo(null)
4 $.. ==> null
5
6 Maybe.<Integer>some(null).flatMap(x -> bar(x))
7 $.. ==> null
8 bar(null)
9 $.. ==> null
```

`Optional` breaks the Left Identity Law but `Maybe` does not.

ii. **Right Identity Law**

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

Suggested Guide:

```
1 Optional<Integer> monad = Optional.<Integer>ofNullable(
     null)
2 monad ==> Optional.empty
3 monad.flatMap(y -> Optional.<Integer>ofNullable(y))
4 $.. ==> Optional.empty
5
6 Maybe<Integer> monad = Maybe.<Integer>some(null)
7 monad ==> [null]
8 monad.flatMap(y -> Maybe.<Integer>some(y))
9 $.. ==> [null]
```

`Optional` and `Maybe` obey the Right Identity Law.

iii. **Associative Law**

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

Suggested Guide:

```
Optional<Integer> monad = Optional.<Integer>ofNullable
    (5)
monad ==> Optional[5]
monad.flatMap(x -> foo(x)).flatMap(y -> foo(y))
$.. ==> Optional[20]
monad.flatMap(x -> foo(x).flatMap(y -> foo(y)))
$.. ==> Optional[20]

Maybe<Integer> monad = Maybe.<Integer>some(5)
monad ==> [5]
monad.flatMap(x -> bar(x)).flatMap(y -> bar(y))
$.. ==> [20]
monad.flatMap(x -> bar(x).flatMap(y -> bar(y)))
$.. ==> [20]
```

`Optional` and `Maybe` obey the Associative Law.

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

i. **Preserving Identity**

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

Suggested Guide:

```
Optional<Integer> functor = Optional.<Integer>ofNullable
    (5)
functor ==> Optional[5]
functor.map(y -> y)
$.. ==> Optional[5]

Maybe<Integer> functor = Maybe.<Integer>some(5)
functor ==> [5]
functor.map(x -> x)
$.. ==> [5]
```

`Optional` and `Maybe` preserve Identity.

ii. **Preserving Composition**

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

Suggested Guide:

```
1  Integer baz(Integer x) {
2    if (x == null) {
3      return 0;
4    }
5    return null;
6  }
7
8  Optional.ofNullable(0).map(x -> baz(x)).map(x -> baz(x))
9  $.. ==> Optional.empty
10 Optional.ofNullable(0).map(x -> baz(foo(x)))
11 $.. ==> Optional[0]
12
13 Maybe.some(0).map(x -> baz(x)).map(x -> baz(x))
14 $.. ==> [0]
15 Maybe.some(0).map(x -> baz(baz(x)))
16 $.. ==> [0]
```

`Optional` does not preserve Composition but `Maybe` does.

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

Suggested Guide:

`Maybe` is both a Monad and a Function but `Optional` is neither.

Suggested Guide:

FURTHER NOTE (adi):

I gave it further thought and I would like to point and `null` values is a **really** bad practice here. I would even consider this two functions `foo` and `bar` to be *improper function*. The reason is that `null` is of any type and it is not really a proper value for `Optional<Integer>` or `Maybe<Integer>`. A more proper function will be:

```
1  Optional<Integer> foo(Integer x) {
2    if (x == null) {
3      return Optional.ofNullable(null);
4    }
5    return Optional.ofNullable(x*2);
6  }
7
8  Maybe<Integer> bar(Integer x) {
9    if (x == null) {
10     return Maybe.some(null);
```

```
  }
  return Maybe.some(x*2);
}
```

Now, if the modified function is used, then **BOTH** `Optional` and `Maybe` are actually a monad. Similarly, the added method `baz` for composition are also *improper function* due to the same reason of `null`. Using a modified function:

```
Integer baz(Integer x) {
  if (x == null) {
    return 0;
  }
  return x;
}
```

we again found that **BOTH** `Optional` and `Maybe` are actually a functor. However, this answer comes from past semester and I keep it for posterity.

You are **guaranteed** that we will not be testing `null` on PE2 and/or final assessment.

2. Consider the following stream pipeline:

```
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?

Suggested Guide:
Running the stream sequentially gives 26 since the pipeline evaluates as `((((0 * 2 + 1) * 2 + 2) * 2 + 3) * 2 + 4)`

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

Suggested Guide:
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*).

The code to produce the above output is as follows:

```
1 System.out.println(Stream.of(1,2,3,4).parallel().reduce(0, (
    result, x) -> {
2   int res = result * 2 + x;
3   System.out.println("(" + result + " * 2 + " + x + ") = " +
      (res)  + ": " + Thread.currentThread().getName());
4   return res;
5 }));
```

The most general form of `reduce` is `reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`. A few important notes:

- If `identity` is not given then *the first element* of the stream is going to be used in place of `identity`
  - The `identity` should still be the identity element of the operation to ensure that the behaviour of `reduce` is the same between *sequential* and *parallel*
- If `combiner` is not given then `accumulator` is used in place of `combiner`
  - As a consequence, without a `combiner`, the type of the output must be the same as the type of the stream *(e.g., if we have a string of `Integer`, the a `reduce` with `combiner` will produce an `Integer`)*
  - The corollary is that with a `combiner`, we can change the type
  - This should give more insight into what is meant by "`accumulator` and `combiner` must be compatible" *(but they need not be the same)*

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:

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

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:

```
1  static BigInteger findFibTerm(int n) {
2    List<BigInteger> fibList = new ArrayList<>();
3    fibList.add(BigInteger.ONE);
4    fibList.add(BigInteger.ONE);
5
6    while (fibList.size() < n) {
7      generateFib(fibList);
8    }
9    return fibList.get(n-1);
10 }
```

(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 $n = 50000$.

Suggested Guide:

```
1  import java.time.Instant;
2  import java.time.Duration;
3  import java.math.BigInteger;
4  import java.util.stream.Stream;
5  import java.util.stream.Collectors;
6  import java.util.List;
7  import java.util.ArrayList;
8
9  class Fib {
10   static void generateFib(List<BigInteger> fibs) {
11     int k = fibs.size();
12     fibs.addAll(
13       Stream
14         .iterate(0, i -> i < k - 1, i -> i + 1)
15         .parallel()
16         .map(i -> fibs.get(k-2).multiply(fibs.get(i)).add(
17                   fibs.get(k-1).multiply(fibs.get(i+1))))
18         .collect(Collectors.toList())
19     );
20   }
21
22   static BigInteger findFibTerm(int n) {
23     List<BigInteger> fibList = new ArrayList<>();
24     fibList.add(BigInteger.ONE);
25     fibList.add(BigInteger.ONE);
26
```

```
27      Instant start = Instant.now();
28      while (fibList.size() < n) {
29        generateFib(fibList);
30      }
31      Instant stop = Instant.now();
32      System.out.println(Duration.between(start,stop).toMillis
        () + "ms");
33      return fibList.get(n-1);
34    }
35
36    public static void main(String[] args) {
37      BigInteger result =
38          findFibTerm(
39            new java.util.Scanner(System.in).nextInt()
40          );
41      System.out.println(result);
42    }
43  }
```