1. In this question, we will be exploring both `InfiniteList<T>` and `Stream<T>`.

   (a) Write a method `fib(int a, int b)` that returns an `InfiniteList<Integer>` where the elements of the infinite list are the Fibonacci numbers starting from `a` and `b`.

```
fib(1, 1).head(); // returns 1
fib(1, 1).tail().head(); // returns 1
fib(1, 1).tail().tail().head(); // returns 2
fib(1, 1).tail().tail().tail().head(); // returns 3
```

Suggested Guide:

```
InfiniteList<Integer> fib(int a, int b) {
  return new InfiniteList<Integer>(() -> a, () -> fib(b, a+b
   ));
}
```

   (b) Netx, write another method that returns the n-th Fibonacci number using `fib` method.

Suggested Guide:

```
int fibonacci(int n) {
  InfiniteList<Integer> il = fib(1, 1);
  for (int i = 0; i < n; i++) {
    il = il.tail();
  }
  return il.head();
}

fibonacci(5); // returns 8
fibonacci(9); // returns 55
```

   (c) Lastly, write a method that returns the first n Fibonacci numbers as an instance of `Stream<Integer>`. For instance, the first 10 Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, and 55.

   Hint: Write an additional `Pair` class that keeps two items around in the stream.

Suggested Guide:

```
class Pair<T> {
  T first;
  T second;
  Pair(T first, T second) {
    this.first = first;
    this.second = second;
  }
}
```

```
9
10  Stream<Integer> fibonacci(int n) {
11    return Stream.iterate(
12                      new Pair<>(1, 1),
13                      pr -> new Pair<>(pr.second,
14                                       pr.first+pr.second))
15                  .map(pr -> pr.first).limit(n);
16  }
17
18  fibonacci(10).forEach(System.out::println);
```

2. `IntStream` is the `int` primitive version of `Stream`. Write a method `omega` with method descriptor `IntStream omega(int n)` that takes in an `int n` and returns a `LongStream` containing the first n.

   The i-th omega number is the number of distinct prime factors of the number i for $i \geq 1$. The first 10 omega numbers are 0, 1, 1, 1, 1, 2, 1, 1, 1, and 2. Note that the first omega number is 0 because $i = 1$ and it has no prime factor since it is only divisible by 1 (*and 1 is not a prime number*).

   The `isPrime` method is given below:

```
1  boolean isPrime(int n) {
2    return IntStream
3             .range(2, n)
4             .noneMatch(x -> n%x == 0);
5  }
```

Suggested Guide:

We use `LongStream` in order to work with large integer values.

```
1  import java.util.stream.IntStream;
2  import java.util.stream.LongStream;
3
4  boolean isPrime(int n) {
5    return IntStream
6             .range(2, n)
7             .noneMatch(x -> n%x == 0);
8  }
9
10 IntStream primeFactorsOf(int x) {
11   return factors(x)
12            .filter(d -> isPrime(d));
13   }
14
15 IntStream factors(int x) {
16   return IntStream
17            .rangeClosed(2, x)
18            .filter(d -> x % d == 0);
19 }
20
21
```

```
LongStream omega(int n) {
  return IntStream
            .range(1, n + 1)
            .mapToLong(x -> primeFactorsOf(x).count());
}

omega(10).forEach(System.out::println)
```

3. Write a method product that takes in two `List` objects `list1` and `list2` to produce a `Stream` containing elements combining each element from `list1` from `list2` using `BiFunction`. This operation is similar to a Cartesian product.

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

For example, the following program fragment:

```
List<Integer> list1 = List.of(1, 2, 3, 4);
List<String> list2 = List.of("A", "B");
product(list1, list2, (str1, str2) -> str1 + str2)
  .reduce("", (x, y) -> x + y + " ");
```

gives the output:

```
1A 1B 2A 2B 3A 3B 4A 4B
```

Suggested Guide:

```
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)));
}
```