1. Which of the following are pure functions?

   (a) Problem #A

```
1  int f(int i) {
2    if (i < 0) {
3      throw new IllegalArgumentException();
4    } else {
5      return i + 1;
6    }
7  }
```

> **Suggested Guide:**
> **No**. This function can throw an exception.

   (b) Problem #B

```
1  int g(int i) {
2    System.out.println(i);
3    return i + 1;
4  }
```

> **Suggested Guide:**
> **No**. This function prints out `i`.

   (c) Problem #C

```
1  int h(int i) {
2    return new Random().nextInt() + i;
3  }
```

> **Suggested Guide:**
> **No**. This function is not deterministict (*i.e.*, *calling* `h(1)` *will give a different answer every time*).

   (d) Problem #A

```
1  int k(int i) {
2    return Math.abs(i);
3  }
```

> **Suggested Guide:**
> **Yes**. This function has no side effect.

2. Consider the following lambda expression:

   `x -> y -> z -> f(x, y z)`

   where `x`, `y`, and `z` are of some type T and `f` returns a value of type R.

(a) What kind of lambda expression is this?

> **Suggested Guide:**
> This is called a **curried function**. A chain of unary function (*i.e.*, *function that takes is one argument*).

(b) Suppose that:

- T and R are of type `Integer`
- `f(x,y,z)` is given by `x + y + z`
- The above lambda expression implements the `Immutator` functional interface

Initialize the appropriate lambda expression and assign it to a variable `trisum`. Given three inputs `x`, `y`, and `z`, show how you can evaluate the lambda expression with `x`, `y`, and `z` to obtain `f(x,y,z)`.

> **Suggested Guide:**
> Let us use `Int` to indicate `Integer`. Then, our variable `trisum` is of type:
>
> `Immutator<Immutator<Immutator<Int, Int>, Int>, Int>`
>
> ```
> 1 Immutator<Immutator<Immutator<Integer, Integer>, Integer>,
>     Integer> trisum
> 2         = x -> y -> z -> (x + y + z);
> ```
>
> You evaluate the curried function by evaluating the arguments one by one, from the outer to the inner function. Consider `x = 3`, `y = 1`, and `z = 2`.
>
> ```
> 1 trisum.invoke(3).invoke(1).invoke(2);
> ```
>
> will return the result of `f(3,1,2)` or simply `6`.

3. The following depicts a classic tail-recursive implementation for finding the sum of values of n (*given by* $\sum_{i=0}^{n} i$) for $n \geq 0$.

```
1 static long sum(long n, long result) {
2   if (n == 0) {
3     return result;
4   } else {
5     return sum(n - 1, n + result);
6   }
7 }
```

In particular, the implementation above is considered **tail-recursive** because the recursive function is at the tail end of the method (*i.e.*, *no computation is done **after** the recursive call returns*). As an example, `sum(100, 0)` gives `5050`.

Although the tail-recursive implementation can be simply rewritten in an iterative form using loops, we desire to capture the original intent of the tail-recursive implementation using delayed evaluation via the `Producer` functional interface.

We present each recursive computation as a `Compute<T>` object. A `Compute<T>` object can be either:

- **Recursive Case:** Represented by a `Recursive<T>` object, that can be recursed, or
- **Base Case:** Represented by a `Base<T>` object, that can be evaluated to a value of type `T`.

As such, we can rewrite the `sum` method as:

```
1  static Compute<Long> sum(long n, long s) {
2    if (n == 0) {
3      return new Base<>(() -> s);
4    } else {
5      return new Recursive<>(() -> sum(n - 1, n + s));
6    }
7  }
```

Then we can evaluate the sum of n terms via the `summer` method below:

```
1  static long summer(long n) {
2    Compute<Long> result = sum(n, 0);
3
4    while (result.isRecursive()) {
5      result = result.recurse();
6    }
7
8    return result.evaluate();
9  }
```

(a) Complete the program by writing the `Compute`, `Base`, and `Recursive` classes.

Suggested Guide:

```
1  public interface Compute<T> {
2    public boolean isRecursive();
3
4    public Compute<T> recurse();
5
6    public T evaluate();
7  }
8
9  public class Base<T> implements Compute<T> {
10   private Producer<T> producer;
11
12   public Base(Producer<T> producer) {
13     this.producer = producer;
14   }
15
16   public boolean isRecursive() {
17     return false;
18   }
19
20   public T evaluate() {
21     return producer.produce();
22   }
23
24   public Compute<T> recurse() {
25     throw new IllegalStateException(
26         "Invalid recursive call in base case"
27     );
28   }
29 }
30
31 public class Recursive<T> implements Compute<T> {
32   private Producer<Compute<T>> producer;
33
34   public Recursive(Producer<Compute<T>> producer) {
35     this.producer = producer;
36   }
37
38   public boolean isRecursive() {
39     return true;
40   }
41
42   public Compute<T> recurse() {
43     return producer.produce();
44   }
45
46   public T evaluate() {
47     throw new IllegalStateException(
48         "Invalid evaluation in recursive case"
49     );
50   }
51 }
```

(b) By making use of a suitable client class `Main`, show how the "tail-recursive" implementation is invoked.

Suggested Guide:

```
1  import java.util.Scanner;
2
3  class Main {
4    static long summer(long n) {
5      Compute<Long> result = sum(n, 0);
6      while (result.isRecursive()) {
7        result = result.recurse();
8      }
9      return result.evaluate();
10   }
11
12   static Compute<Long> sum(long n, long s) {
13     if (n == 0) {
14       return new Base<>(() -> s);
15     } else {
16       return new Recursive<>(() -> sum(n - 1, n + s));
17     }
18   }
19
20   public static void main(String[] args) {
21     System.out.println(summer(new Scanner(System.in).
       nextLong()));
22   }
23 }
```

(c) Redefine the `Main` class so that it now computes the factorial of n recursively.

Suggested Guide:

```
1  import java.util.Scanner;
2  class Main {
3    static Compute<Long> fact(long n, long s) {
4      if (n == 0) {
5        return new Base<>(() -> s);
6      } else {
7        return new Recursive<>(() -> fact(n - 1, n * s));
8      }
9    }
10
11   public static void main(String[] args) {
12     Compute<Long> result = fact(new Scanner(System.in).
       nextLong(), 1);
13     while (result.isRecursive()) {
14       result = result.recurse();
15     }
16     System.out.println(result.evaluate());
17   }
18 }
```