Oracle SaaS Cloud Security CAS Scala Assignment

General Instructions

- Avoid using mutable constructs. If their use are deemed necessary please provide a justification.
- Consider and elaborate on what's needed for the proposed solutions to be considered production ready, both from quality point of view and performance characteristics. *Considerations can be either fully implemented, implemented in pseudo-code or just documented.*
- Provide unit tests that cover more than just the happy path.

Problem 1

Implement a function with the following signature, that merge two ordered lists of integers into a sorted one:

```
def mergeSortedIntLists(left: List[Int], right: List[Int]): List[Int]
```

Instructions: you can't rely on any sorting data structure or any provided sorting algorithm.

Problem 2

Implement a generalised version of the function from `Problem 1` that accept anything as the content of the list:

```
def mergeSortedLists[A](left: List[A], right: List[A]): List[A]
```

Instructions: modify the signature as required.

Problem 3

Implement a generalised version of the function from `Problem 2` that accept any `F[A]`, where `F` is the collection and `A` is the type of the items in the collection. The function needs to work for mutable and immutable collections in Scala standard library, as well as `Option`, and the following custom list definition:

```
sealed trait BackwardsList[+A]
case object BWLNil extends BackwardsList[Nothing]
case class BWLCons[A](last: A, init: BackwardsList[A])

def mergeSorted[F[_], A](left: F[A], right: F[A]): F[A]

// Usage example:
val x1 = List(1,2,3)
val x2 = List(4,5,6)
val xs = mergeSorted(x1, x2)

val v1 = Vector(1,2,3)
val v2 = Vector(4,5,6)
val vs = mergeSorted(v1, v2)

val bl1 = BWLCons(1, BWLCons(2, BWLCons(3, BWLNil)))
val bl2 = BWLCons(4, BWLCons(5, BWLCons(6, BWLNil)))
val bls = mergeSorted(bl1, bl2)
```

Instructions: modify the signature as required.

Problem 4

Given the function

```
def generateNext[A](previous: A): IO[Option[A]] = ???
```

and assuming that: - it can't fail - it will return a result almost immediately.

Following is an example of such function for producing `Longs`.

```
def generateNextLong(previous: Long): IO[Option[Long]] =
  IO(Random.nextInt().abs % 100).map(v => Option(v.toLong + previous))
```

a - implement the function below that calls it to produce an ordered list of `As` of size `takeN` consuming exactly N elements in total

```
def takeNSorted[A](
    initial: A, //Starting generation on `initial` value
    producer1: A => IO[Option[A]],
    producer2: A => IO[Option[A]],
    takeN: Int
): IO[List[A]] = ???

// Usage example:
takeNSorted[Int](0, generateNext, generateNext, 300)
```

b - considering that `generateNext` won't fail, which error scenarios have you considered?

c - assuming that `generateNext` might fail or take an arbitrary amount of time to produce, how would you handle such cases?

Instructions: for this problem it's recommended to use a streaming framework like `Fs2` or `ZIO Streams`.

Problem 5

Given the functions

```
def enqueuedA1[A]: cats.effect.std.Queue[IO, A] = ???
def enqueuedA2[A]: cats.effect.std.Queue[IO, A] = ???
```

and assuming that: - elements in each queue are ordered, however, no ordering is maintained across the two queues - they can't fail - the queues are never empty

The following function can be used to fill each queue:

```
def fillQueueForPush[A](q: cats.effect.std.Queue[IO, A], lastRef: Ref[IO, A])(next: A => IO[Option[A]]):
  for {
    _ <- IO.sleep((Random.nextInt() % 10).millis)
    last <- lastRef.get
    n <- next(last).flatMap:
      case Some(l) => q.offer(l) *> lastRef.set(l)
      case None => IO.unit

    _ <- fillQueueForPush(q, lastRef)(next)
  } yield ()
```

a - implement the function below that calls them to produce an ordered list of `As` of size `takeN` consuming from both the queues

```
def streamSorter[A](
  queue1: cats.effect.std.Queue[IO, A],
  queue2: cats.effect.std.Queue[IO, A],
  takeN: Int
): IO[List[A]] = ???

//Usage sample
```scala
{
  (
```
```

```
    Ref.of[IO, Long](0L),
    Ref.of[IO, Long](0L),
    cats.effect.std.Queue.unbounded[IO, Long],
    cats.effect.std.Queue.unbounded[IO, Long]
  ).parFlatMapN { case (ref1, ref2, queue1, queue2) =>
      IO.race(
        (fillQueueForPush(queue1, ref1)(generateNextInt), fillQueueForPush(queue2, ref2)(generateNextInt
        streamSorter[Long](queue1, queue2, 300)
      )
  }
```

b - considering that `enqueuedAX` won't fail, which error scenarios have you considered ?

Instructions: for this problem it's recommended to use a streaming framework like `Fs2` or `ZIO Streams`. Any concurrently safe queue can be used.

Problem 6

Implement `Problem 4` and `Problem 5` without using any streaming library (like `Fs2` or `ZIO Streams`) and assuming that: - the generation functions (pull functions and queues) can behave normally, that is, they may or may not return a result (the queues can be empty) - the generation functions `nextOrderedAFromX` can fail - the memory space is finite

Instructions: `Cats Effect` or `ZIO` can still be used. Make sure to explain any extra parameter added to any of the functions (e.g.: lingerTime).

Deliverables

1. the source code of the solution to one of the alternatives:
   - alternative 1: problems 1, 2, 3, 4 and 5
   - alternative 2: problems 1, 4, 5 and 6 (`Problem 4` and `Problem 5` without using any streaming library (like `Fs2` or `ZIO Streams`)
2. the unit tests that you consider sufficient for a production-grade functionality
3. a `README.md` file containing instructions on how to build, run, and test the code

Format

Compressed archive by email.