Exercise 1 : Parallel Encoding

In this exercise, your group will devise a parallel algorithm to encode sequences using the run-length encoding scheme. The encoding is very simple. It transforms sequences of letters such that all subsequences of the same letter are replaced by the letter and the sequence length. For instance:

`"AAAAATTTGGGGTCCCAAC"` ⇒ `"A5T3G4T1C3A2C1"`

Your goal in this exercise is to come up with a parallel implementation of this algorithm. The function should have the following shape:

```
def rle(data: ParSeq[Char]): Buffer[(Char, Int)] =
  data.aggregate(???)(???, ???)
```

The `Buffer` class is already given to you. A buffer of type `Buffer[A]` represents sequences of elements of type `A`. It supports the following methods, all of which are efficient:

```
def isEmpty: Boolean  // Checks if the buffer is empty.
def head: A           // Returns the first element of the buffer.
def tail: Buffer[A]   // Returns the buffer minus its first element.
def last: A           // Returns the last element of the buffer.
def init: Buffer[A]   // Returns the buffer minus its last element.
def ++(that: Buffer[A]): Buffer[A] // Concatenate two buffers.
def append(elem: A): Buffer[A] // Appends a single element to the right.

Buffer.empty[A]: Buffer[A] // Returns an empty buffer.
Buffer.singleton[A](element: A): Buffer[A] // Single element buffer.
```

Exercise 2 : Parallel Two Phase Construction

In this exercise, you will implement an array `Combiner` using internally a double linked list (DLL). Below is a minimal implementation of the `DLLCombiner` class and the related `Node` class. Your goal for this exercise is to complete the implementation of the (simplified) `Combiner` interface of the `DLLCombiner` class.

```
class DLLCombiner[A] extends Combiner[A, Array[A]] {
  var head: Node[A] = null // null for empty lists.
  var last: Node[A] = null // null for empty lists.
  var size: Int = 0

  // Implement these three methods...
  override def +=(elem: A): Unit = ???
  override def combine(that: DLLCombiner[A]): DLLCombiner[A] = ???
  override def result(): Array[A] = ???
}

class Node[A](val value: A) {
  var next: Node[A]     // null for last node.
  var previous: Node[A] // null for first node.
}
```

Question 1

What computational complexity do your methods have? Are the actual complexities of your methods acceptable according to the `Combiner` requirements?

Question 2

One of the three methods you have implemented, `result`, should work in parallel according to the `Combiner` contract. Can you think of a way to implement this method efficiently using 2 parallel tasks?

Question 3

Can you, given the current internal representation of your combiner, implement `result` so that it executes efficiently using 4 parallel tasks? If not, can you think of a way to make it possible?

Hint: This is an open-ended question, there might be multiple solutions. In your solution, you may want to add extra information to the class `Node` and/or the class `DLLCombiner`.

Exercise 3: Pipelines

In this exercise, we look at pipelines of functions. A pipeline is simply a function which applies its argument successively to each function of a sequence. To illustrate this, consider the following pipeline of 4 functions:

```
val p: Int => Int = toPipeline(ParSeq(_ + 1, _ * 2, _ + 3, _ / 4))
```

The pipeline `p` is itself a function. Given a value `x`, the pipeline `p` will perform the following computations to process it. In the above example,

```
p(x) = (((x + 1)        Application of first function
        * 2)            Application of second function
        + 3)            Application of third function
        / 4             Application of fourth function
```

In this exercise, we will investigate the possibility to process such pipelines in parallel.

Question 1

Implement the following `toPipeline` function, which turns a parallel sequence of functions into a pipeline. You may use any of the parallel combinators available on `ParSeq`, such as the parallel `fold` or the parallel `reduce` methods.

```
def toPipeline(fs: ParSeq[A => A]): A => A = ???
```

Hint: Functions have a method called `andThen`, which implements function composition: it takes as argument another function and also returns a function. The returned function first applies the first function, and then applies the function passed as argument to that result. You may find it useful in your implementation of pipeline.

Question 2

Given that your `toPipeline` function works in parallel, would the pipelines it returns also work in parallel? Would you expect pipelines returned by a sequential implementation of `toPipeline` to execute any slower? If so, why?

Discuss those questions with your group and try to get a good understanding how what is happening.

Question 3

Instead of arbitrary functions, we will now consider functions that are constant everywhere except on a finite domain. We represent such functions in the following way:

```
class FiniteFun[A](mappings: immutable.Map[A, A], default: A) {
  def apply(x: A): A =
    mappings.get(x) match {
      case Some(y) => y
      case None    => default
    }

  def andThen(that: FiniteFun[A]): FiniteFun[A] = ???
}
```

Implement the `andThen` method. Can pipelines of such finite functions be efficiently constructed in parallel using the appropriately modified `toPipeline` method? Can the resulting pipelines be efficiently executed?

Question 4

Compare the *work* and *depth* of the following two functions, assuming infinite parallelism. For which kind of input would the parallel version be asymptotically faster?

```
def applyAllSeq[A](x: A, fs: Seq[FiniteFun[A]]): A = {
  // Applying each function sequentially.
  var y = x
  for (f <- fs)
    y = f(y)
  y
}

def applyAllPar[A](x: A, fs: ParSeq[FiniteFun[A]]): A = {
  if (fs.isEmpty) x
  else {
    // Computing the composition in parallel.
    val p = fs.reduce(_ andThen _)
    // Applying the pipeline.
    p(x)
  }
}
```