Exercise 1 : Introduction to Concurrency

Freshly graduated from EPFL, you all have been hired as contractors for a successful and rapidly growing bank. The bank has recently been experiencing problems with their money management system, coded in Scala, and so they hired the best and brightest young engineers they could find: you! The system has been working perfectly fine so far, they tell you. In the past days, due to an increased number of customers, they had to switch from a single threaded sequential execution environment to a multithreaded concurrent environment, in which multiple threads may perform transactions concurrently. That's when problems started, your manager says…

Below is the code responsible to withdraw money from the account `from` and transfer it to the account `to`, within the same bank.

```
def transfer(from: Account, to: Account, amount: BigInt) {
  val balanceFrom = from.balance
  if (balanceFrom >= amount) {
    from.balance = balanceFrom - amount
    val balanceTo = to.balance
    to.balance = balanceTo + amount
  }
}
```

For the bank, it is very important that the following two properties hold in any sequence of transfer transactions:

1. The balance of an account never goes below 0.
2. The total sum of money held by the bank is constant.

Question 1

Does the above `transfer` method respect the two properties in a **sequential** execution environment ?

Question 2

What can go wrong in a setting where multiple threads can execute the `transfer` method concurrently ? For each of the two desired properties of the system, check if its holds in this concurrent environment. If not, come up with an example execution which exhibits a violation of the property.

Question 3

For each of the proposed implementation of `transfer` below, check which of the properties hold. Additionally, check if the system is vulnerable to *deadlocks*.

Variant 1

```
def transfer(from: Account, to: Account, amount: Long) {
  val balanceFrom = from.balance
  if (balanceFrom >= amount) {
    from.synchronized {
      from.balance = balanceFrom - amount
    }
    to.synchronized {
      val balanceTo = to.balance
      to.balance = balanceTo + amount
    }
  }
}
```

Variant 2

```
def transfer(from: Account, to: Account, amount: Long) {
  from.synchronized {
    val balanceFrom = from.balance
    if (balanceFrom >= amount) {
      from.balance = balanceFrom - amount
      to.synchronized {
        val balanceTo = to.balance
        to.balance = balanceTo + amount
      }
    }
  }
}
```

Variant 3

```
object lock // Global object.
def transfer(from: Account, to: Account, amount: Long) {
  lock.synchronized {
    val balanceFrom = from.balance
    if (balanceFrom >= amount) {
      from.balance = balanceFrom - amount
      val balanceTo = to.balance
      to.balance = balanceTo + amount
    }
  }
}
```

Exercise 2 : Depth

As a group, review the notion of *depth* seen in the video lectures. What does it represent ?

Below is a formula for the depth of a *divide and conquer* algorithm working on an array segment of size *L*, as a function of *L*. The values *c, d* and *T* are constants. We assume that *L>0* and *T>0*.

$$D(L) = \begin{cases} c \cdot L & \text{if } L \leq T \\ max(D(\lfloor \frac{L}{2} \rfloor), D(L - \lfloor \frac{L}{2} \rfloor)) + d & \text{otherwise} \end{cases}$$

Below the threshold *T*, the algorithm proceeds sequentially and takes time *c* to process each single element. Above the threshold, the algorithm is applied recursively over the two halves of the array. The results are then merged using an operation that takes *d* units of time.

Question 1

Is it the case that for all $1 \leq L_1 \leq L_2$ we have $D(L_1) \leq D(L_2)$?

If it is the case, prove the property by induction on L. If it is not the case, give a counterexample showing values of L_1, L_2, *T*, *c*, and *d* for which the property does not hold.

Question 2

Prove a logarithmic upper bound on *D(L)*. That is, prove that *D(L)* is in *O(log L)* by finding specific constants *a*,*b* such that $D(L) \leq a \log_2 L + b$.

Exercise 3 : Parallel Reductions

Question 1

As a group, write a function called `minMax`, which should take a non-empty array as input and return a pair containing the smallest and the largest element of the array.

```
def minMax(a: Array[Int]): (Int, Int) = ???
```

Now write a parallel version of the function. You may use the construct `task` seen in the lectures.

Question 2

Imagine that the data structure you are given, instead of an `Array[A]`, is one called `ParSeq[A]`. This class offers the two following methods, which work in parallel:

```
def map[B](f: A => B): ParSeq[B]
def reduce(f: (A, A) => A): A
```

Can you write the following `minMax` function in terms of `map` and/or `reduce` operations ?

```
def minMax(data: ParSeq[Int]): (Int, Int) = ???
```

Question 3

What property does the function `f` need to satisfy in order to have the same result regardless on how `reduce` groups the applications of the operation `f` to the elements of the data structure? Prove that your function `f` indeed satisfies that property.

Exercise 4 : 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 signature:

```
def rle(data: ParSeq[Char]): Buffer[(Char, Int)]
```

The `Buffer` class is already given to you. A buffer of type `Buffer[A]` represents sequences of elements of type `A`, with a notable feature that supports efficient concatenation. It supports the following methods, all of which run in **constant** time:

```
def isEmpty: Boolean
def head: A
def last: A
def ++(that: Buffer[A]): Buffer[A]

Buffer.singleton(element: A): Buffer[A]
```

As in the previous exercise, you can assume that instances of `ParSeq[A]` offer parallel implementation of the following methods:

```
def map[B](f: A => B): ParSeq[B]
def reduce(f: (A, A) => A): A
```