Exercise 1 : Depth

Note: This exercise was already given last week. If you have already done it, briefly discuss the solution as a group before moving to the next exercise.

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.

Somewhat counterintuitively, the property doesn't hold. To show this, let's take the following values for L_1, L_2, *T, c, and d.*

$L_1 = 10$, $L_2 = 12$, $T = 11$, $c = 1$, and $d = 1$.

Using those values, we get that:

$D(L_1) = 10$
$D(L_2) = max(D(6), D(6)) + 1 = 7$

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 $D(L) \leq a \log_2 L + b$.

<u>Proof sketch</u>

Define the following function D'(L).

$$D'(L) = \begin{cases} c \cdot L & \text{if } L \leq T \\ max(D'(\lfloor \frac{L}{2} \rfloor), D'(L - \lfloor \frac{L}{2} \rfloor)) + d + \underline{\underline{c \cdot T}} & \text{otherwise} \end{cases}$$

Show that $D(L) \leq D'(L)$ *for all* $1 \leq L$ *.*

Then, show that, for any $1 \leq L_1 \leq L_2$ *we have* $D'(L_1) \leq D'(L_2)$*. This property can be shown by induction on* L_2*.*

Finally, let n be such that $L \leq 2^n < 2L$*. We have that:*

$D(L) \leq D'(L)$ *Proven earlier.*
$\leq D'(2^n)$ *Also proven earlier.*
$\leq \log_2(2^n)\,(d + cT) + cT$
$< \log_2(2L)\,(d + cT) + cT$
$= \log_2(L)\,(d + cT) + \log_2(2)\,(d + cT) + cT$
$= \log_2(L)\,(d + cT) + d + 2cT$

Done.

Exercise 2 : Aggregate

In the video lectures of this week, you have been introduced to the `aggregate` method of `ParSeq[A]` (and other parallel data structures…). It has the following signature:

```
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```

Discuss, as a group, what `aggregate` does and what its arguments represent.

Question 1

Consider the parallel sequence `xs` containing the three elements `x1`, `x2` and `x3`. Also consider the following call to aggregate:

```
xs.aggregate(z)(f, g)
```

The above call might potentially result in the following computation:

```
f(f(f(z, x1), x2), x3)
```

But it might also result in other computations. Come up with at least 2 other computations that may result from the above call to `aggregate`.

Some examples:

- *g(f(z, x1), f(f(z, x2), x3))*
- *g(f(f(z, x1), x2), f(z, x3))*
- *g(g(f(z, x1), f(z, x2)), f(z, x3))*
- *g(f(z, x1), g(f(z, x2), f(z, x3)))*

Question 2

Below are other examples of calls to `aggregate`. In each case, check if the call can lead to different results depending on the strategy used by `aggregate` to aggregate all values contained in `data` down to a single value. You should assume that `data` is a parallel sequence of values of type `BigInt`.

Variant 1

```
data.aggregate(1)(_ + _, _ + _)
```

This might lead to different results.

Variant 2

```
data.aggregate(0)((acc, x) => x - acc, _ + _)
```

This might lead to different results.

Variant 3

```
data.aggregate(0)((acc, x) => acc - x, _ + _)
```

This is always leads to the same result.

Variant 4

```
data.aggregate(1)((acc, x) => x * x * acc, _ * _)
```

This is always leads to the same result.

Question 3

Under which condition(s) on `z`, `f`, and `g` does `aggregate` always lead to the same result ? Come up with a formula on `z`, `f`, and `g` that implies the correctness of `aggregate`.

Hint: You may find useful to use calls to `foldLeft(z)(f)` in your formula(s).

A property that implies the correctness is:

```
forall xs, ys.

g(xs.foldLeft(z)(f), ys.foldLeft(z)(f))
==
(xs ++ ys).foldLeft(z)(f)
```

Question 4

Implement `aggregate` using the methods `map` and/or `reduce` of the collection you are defining aggregate for.

A solution:

```
def aggregate(z: B)(f: (B, A) => B, g: (B, B) => B): B =
  if (this.isEmpty) z
  else this.map((x: A) => f(z, x)).reduce(g)
```

Question 5

Implement `aggregate` using the `task` and/or `parallel` constructs seen in the first week and the `Splitter[A]` interface seen in this week's videos. The `Splitter` interface is defined as:

```
trait Splitter[A] extends Iterator[A] {
  def split: Seq[Splitter[A]]
  def remaining: Int
}
```

You can assume that the data structure you are defining `aggregate` for already implements a `splitter` method which returns an object of type `Splitter[A]`.

Your implementation of `aggregate` should work in parallel when the number of remaining elements is above the constant `THRESHOLD` and sequentially below it.

<u>Hint:</u> `Iterator`, and thus `Splitter`, implements the `foldLeft` method.

A solution:

```
def aggregate(z: B)(f: (B, A) => B, g: (B, B) => B): B = {

  def go(s: Splitter[A]): B = {
    if (s.remaining <= THRESHOLD) {
      s.foldLeft(z)(f)
    }
    else {
      val splitted = s.split

      val subs = splitted.map((t: Splitter[A]) => task { go(t) })
      subs.map(_.join()).reduce(g)
    }
  }

  go(splitter)
}
```

Question 6

Discuss the implementations from questions 4 and 5. Which one do you think would be more efficient ?

The version from question 4 may require 2 traversals (one for map, one for reduce) and does not benefit from the (potentially faster) sequential operator f.

~~Exercise 3 : Parallel Encoding~~

This question has been moved to the third exercise session.