Exercise 9.1 (Digital clock) Chronos and Hora look at a digital clock with a seven-segment display:

```
 _       _   _       _   _   _   _   _ 
| |   |  _|  _| |_| |_  |_    | |_| |_|
|_|   | |_   _|   |  _| |_|   | |_|  _|
```

Chronos sees only the four upper segments, Hora sees only the four lower segments. We want to investigate the states of knowledge at different time points from 00:00 to 23:59.

A philosophical remark up front. This task seems simple enough that one might be tempted to try to solve it manually. Even a randomly chosen pedestrian who has no idea of such concepts as “definition” or “logic” (let alone “Aumann model of incomplete information” and “knowledge operator”), would probably attempt to solve it, and would possibly even come up with an answer that resembles the correct solution. But this answer would be based on intuition and hand-waving, and it would not explain how to deal with such problems in general. Instead of relying on bare intuition, we shall write a program that computes the correct answer. We will use Scala for scripting.

The first thing one would like to do is to generate a description of the partitions that describe the events observable by Choronos and Hora. For this, we explicitly write down the states of LEDs as strings and perform some substring manipulations in order to group the numbers by their visible parts:

```scala
// States of leds on a digital display
val ledStrings = Array(
  "1110111","0010010","1011101","1011011","0111010",
  "1101011","1101111","1010010","1111111","1111011"
)

// This method generates the knowledge partition
// of the set {start, ..., end} for a player, it
// requires a function that tells us what the player
// can see. Therefore, it transforms the geometrical
// information into a simple partition of integers.
def partition(
  start: Int, end: Int,
  visibility: String => String
) = {
  // determine number of digits (1 or 2)
  var nd = numDigits(end)

  // group numbers by the visible parts
  val p = (start to end).groupBy { k =>
    val ds = digits(nd, k)
    ds.map { digit => visibility(ledStrings(digit)) }
  }

  // ‘group‘ returns maps,
  // we need only the right hand side
  p.values.map{_.toSet}.toSet
}
```

Here is what the string manipulations look like. The bits are arranged in a way such that first four correspond to upper part, and last four correspond to the lower part:

```
def upperPartition(start: Int, end: Int) =
  partition(start, end, _.take(4))
def lowerPartition(start: Int, end: Int) =
  partition(start, end, _.drop(3))
```

Now we generate two lists (one for Chronos, one for Hora) with three partitions in each list. The first partition is of the range $\{0 \dots 23\}$, the second of the range $\{0 \dots 5\}$ and the third of the range $\{0 \dots 9\}$. Notice that we cannot treat the first two digits independently, because for example Hora cannot tell 13 and 19 apart, but can recognize 23, because there are no 25 or 29.

```
// lists of partitions for Chronos and Hora
val upperPartitions = List(23, 5, 9).map{ n =>
  upperPartition(0, n)
}

val lowerPartitions = List(23, 5, 9).map{ n =>
  lowerPartition(0, n)
}
```

This is what the partitions look like: for Chronos:

$$\begin{aligned}
&\{\\
&\ \{00\}, \{01\}, \{02, 03\}, \{04\}, \{05, 06\},\\
&\ \{07\}, \{08, 09\}, \{10\}, \{11\}, \{12, 13\},\\
&\ \{14\}, \{15, 16\}, \{17\}, \{18, 19\}, \{20\},\\
&\ \{21\}, \{22, 23\}\\
&\}\\
&\{\{0\}, \{1\}, \{2, 3\}, \{4\}, \{5\}\}\\
&\{\{0\}, \{1\}, \{2, 3\}, \{4\}, \{5, 6\}, \{7\}, \{8, 9\}\}
\end{aligned}$$

for Hora:

$$\begin{aligned}
&\{\\
&\ \{00\}, \{01, 07\}, \{02\}, \{03, 05, 09\}, \{04\},\\
&\ \{06, 08\}, \{10\}, \{11, 17\}, \{12\}, \{13, 15, 19\},\\
&\ \{14\}, \{16, 18\}, \{20\}, \{21\}, \{22\}, \{23\}\\
&\}\\
&\{\{0\}, \{1\}, \{2\}, \{3, 5\}, \{4\}\}\\
&\{\{0\}, \{1, 7\}, \{2\}, \{3, 5, 9\}, \{4\}, \{6, 8\}\}
\end{aligned}$$

The event “Chronos knows time exactly” is simply the product of unions of sets with exactly one element. Same holds for Hora. Here are the both events:

$$\begin{aligned} chronosKnowsTime &= \{00, 01, 04, 07, 10, 11, 14, 17, 20, 21\} \times \{0, 1, 4, 5\} \times \{0, 1, 4, 7\} \\ horaKnowsTime &= \{00, 02, 04, 10, 12, 14, 20, 21, 22, 23\} \times \{0, 1, 2, 4\} \times \{0, 2, 4\} \end{aligned}$$

So far we did not need the knowledge operators. One important observation about the knowledge operators is that they respect the product structure of the involved sets. That is, if $\mathcal{F}_i$ are partitions of Ω_i and $\mathcal{F} = \prod_i \mathcal{F}_i$ is a partition of $\Omega = \prod_i \Omega_i$, then for the knowledge operator $K_{\mathcal{F}}$ it holds:

$$K_{\mathcal{F}}(A) = \prod_i K_{\mathcal{F}_i}(A_i).$$

This property is important for the speed of the computation and for a nice structure of the finite result: we still get a representation as a product of sets, and not a giant set of tuples. Here is the formula expressed as code:

```
// The knowledge operator for a single \Omega_i
def knowledge(partition: Set[Set[Int]])(a: Set[Int]) = {
  val subsetsOfA = partition.filter{x => x subsetOf a}
  subsetsOfA.flatten
}

// Knowledge operator for the whole product space
// \Omega = \prod_i \Omega_i
// implemented in terms of the previous knowledge
// operator for a single slice
def knowledge(
  partitions: List[Set[Set[Int]]]
)(a: List[Set[Int]]): List[Set[Int]] = {
  for ((p, aComponent) <- partitions zip a) yield {
    knowledge(p)(aComponent)
  }
}
```

Now we can use the definition of the general knowledge operator and the partitions generated previously to instantiate the specific knowledge operators for Chronos and Hora:

```
val k_u = knowledge(upperPartitions)(_)
val k_l = knowledge(lowerPartitions)(_)
```

Here are the results of $k_u(lowerKnowsTime)$ and $k_l(upperKnowsTime)$:

$$\begin{aligned} K_C(horaKnowsTime) &= \{00, 04, 10, 14, 20, 21, 22, 23\} \times \{0, 1, 4\} \times \{0, 4\} \\ K_H(chronosKnowsTime) &= \{00, 01, 04, 07, 10, 11, 14, 17, 20, 21\} \times \{0, 1, 4\} \times \{0, 1, 4, 7\} \end{aligned}$$

Notice that there are situations where for example Hora does not know the time herself, but knows that Chronos knows it (e.g. at 11:41).

Finally, we want to get the set of all events where the knowledge of time becomes *common knowledge*. For this, observe that if there is a sequence of player indices $i_1, \dots, i_n$ and for the event

$$C := K_{i_n} \dots K_{i_1}(A)$$

it holds that $K_pC = C$ for all players p, then C is exactly the set of events where A is common knowledge. Furthermore, notice that $K_p(A)$ is always contained in A. That means that we can start with an event $C_0 = A$ and then apply K_i in some rather arbitrary order, until $K_pC = C$ for all players p holds. Since we work with finite sets, the sequence

$$A \supseteq K_{i_1}(A) \supseteq K_{i_2}K_{i_1}(A) \supseteq \dots$$

is noetherian, so the algorithm terminates. This is exactly what we do in the code:

```
val ks = List(k_u, k_l)
var commonKnowledge = bothKnowTime
while (!ks.forall{k => k(commonKnowledge) == commonKnowledge}){
  println(prodToString(commonKnowledge))
  for (k <- ks) commonKnowledge = k(commonKnowledge)
}
```

This gives the set of all time-points where the time is common knowledge:

$$\{00, 04, 10, 14, 20, 21\} \times \{0, 1, 4\}x\{0, 4\}$$

See code for more details, if necessary.