Exercice 1 : Preuve par induction

Partie 1

Preuve avec n comme variable d'induction.

- Cas $n = 0$

```
zip(drop(xs, n), drop(ys, n))
   = zip(drop(xs, 0), drop(ys, 0))
   = zip(xs, ys)                          // d1
   = drop(zip(xs, ys), 0)                 // d1
   = drop(zip(xs, ys), n)
```

- Cas $n > 0$
  - Sous-cas $xs = Nil$

```
zip(drop(xs, n), drop(ys, n))
   = zip(drop(Nil, n), drop(ys, n))
   = zip(Nil, drop(ys, n))           // d2
   = Nil                             // z1
```

  - Sous-cas $ys = Nil$

```
zip(drop(xs, n), drop(ys, n))
   = zip(drop(xs, n), drop(Nil, n))
   = zip(drop(xs, n), Nil)           // d2
   = Nil                             // z2
```

  - Sous-cas $xs = x :: xss, ys = y :: yss$. La hypothèse d'induction est la suivante:

```
zip(drop(xs, n-1), drop(ys, n-1)) == drop(zip(xs, ys), n-1) // hyp
```

    Elle est utilisée dans la preuve du dernier sous-cas:

```
zip(drop(xs, n), drop(ys, n))
   = zip(drop(x :: xss, n), drop(y :: yss, n))
   = zip(drop(xss, n-1), drop(yss, n-1))        // d0
   = drop(zip(xss, yss), n-1)                   // hyp
   = drop((x, y) :: zip(xss, yss), n)           // d0
   = drop(zip(x :: xss, y :: yss), n)           // z0
```

Partie 2

Equation en question: `zip(reverse(xs), reverse(ys)) == reverse(zip(xs, ys))`

Contre-exemple: $xs = 1 :: Nil$ et $ys = 1 :: 2 :: Nil$.

Démonstration coté gauche:

```
zip(reverse(xs), reverse(ys))
   = zip(reverse(1 :: Nil), reverse(1 :: 2 :: Nil))
   = zip(1 :: Nil, 2 :: 1 :: Nil)                    // reverse
   = (1,2) :: zip(Nil, 1 :: Nil)                     // z0
   = (1,2) :: Nil                                    // z1
```

Démonstration coté droite:

```
reverse(zip(xs, ys))
   = reverse(zip(1 :: Nil, 1 :: 2 :: Nil))
   = reverse((1,1) :: zip(Nil, 2 :: Nil))    // z0
   = reverse((1,1) :: Nil)                   // z1
   = (1,1) :: Nil                            // reverse
```

Exercice 2 : Ensembles d'entiers

Une solution possible:

```
type Interval = (Int, Int)
type IntervalSet = List[Interval]

def contains(set: IntervalSet, item: Int): Boolean = set match {
  case (a, b) :: xs => (a <= item && item <= b) || contains(xs, item)
  case Nil => false
}

def addElem(elem: Interval, set: IntervalSet): IntervalSet = set match {
  case x :: xs =>
    if (mergeable(x, elem)) {
      merge(x, elem) :: xs
    } else {
      if (elem._2 < x._1)
        elem :: x :: xs
      else
        x :: addElem(elem, xs)
    }
  case Nil =>
    elem :: Nil
}

def union(s1: IntervalSet, s2: IntervalSet): IntervalSet = s1 match {
  case x :: xs => addElem(x, union(xs, s2))
  case Nil => s2
}
```

Si vous voulez tester vôtre code, il vous faut également la partie suivante:

```
def mergeable(i1: Interval, i2: Interval) = {
  val ((a1, b1), (a2, b2)) = (i1, i2)
  if (a1 < a2)
    (b1+1 >= a2)
  else if (a2 < a1)
    (b2+1 >= a1)
  else true
}

def merge(i1: Interval, i2: Interval): Interval = {
  assert(mergeable(i1, i2))
  val ((a1, b1), (a2, b2)) = (i1, i2)
  (math.min(a1, a2), math.max(b1, b2))
}
```

Et voici un test

```
def test() {
  val s1 = addElem((6, 6), addElem((1,3), addElem((12, 14), Nil)))
  val s2 = addElem((4, 5), Nil)
  val s3 = addElem((9, 11), Nil)
  val s4 = addElem((7, 7), Nil)
```

```
  println(s1)
  println(s2)
  println(s3)
  println(s4)
  println(union(s2, s3))
  println(union(union(s1, s2), s3))
  println(union(s2, union(s3, s4)))
}
```

Exercice 3 : Images fonctionnelles

Partie 1

```
def imageToText(m: Image, tl: Point2D, br: Point2D, w: Int, h: Int) = {
  val xStep = (br.x - tl.x) / w
  val yStep = (br.y - tl.y) / h

  for (y <- (0 until h).toList) yield
    for (x <- (0 until w).toList) yield
      if (m(Point2D(tl.x + xStep*x, tl.y + yStep*y))) '%'
      else '.'
}
```

Partie 2

```
def rotate(theta: Double)(i: Image): Image =
  { p: Point2D => i(Point2D(p.x * cos(-theta) - p.y * sin(-theta),
                            p.x * sin(-theta) + p.y * cos(-theta))) }
```

Partie 3

```
val mandelbrot: Image = { p: Point2D =>
  val c = Complex(p.x, p.y)
  def loop(z: Complex, i: Int): Boolean =
    (i == 100) || (z.modulus < 2 && loop(z * z + c, i + 1))
  loop(c, 1)
}
```