Exercice 1 : Nombres de Hamming (15 points)

Partie 1

(*6 points*) La fonction `merge` s'écrit comme suit :

```
  def merge(xs: Stream[Int], ys: Stream[Int]): Stream[Int] = {
    val x = xs.head;
    val y = ys.head;
    if (x < y) Stream.cons(x, merge(xs.tail, ys))
    else if (x > y) Stream.cons(y, merge(xs, ys.tail))
    else Stream.cons(x, merge(xs.tail, ys.tail));
  }
```

Partie 2

(*9 points*) La valeur `hamming` se définit ainsi :

```
object Main {
  def merge(xs: Stream[Int], ys: Stream[Int]): Stream[Int] =
    // cf. partie 1

  val hamming: Stream[Int] =
    Stream.cons(1, merge(hamming map (x => 2*x),
                         merge(hamming map (x => 3*x),
                               hamming map (x => 5*x))));

  def main(args: Array[String]): Unit =
    hamming take 50 print;
}
```

Exercice 2 : Preuve par induction structurelle (10 points)

Pour démontrer l'égalité

```
at(at(t, u), v) = at(t, u::v)
```

on effectue une preuve par induction structurelle sur `u`.

Pour cela on traite tout d'abord le cas de base pour chaque côté de l'égalité avant de passer à l'étape d'induction.

1. Cas de base : `u = Nil`

   Pour le côté gauche on a :

   ```
       at(at(t, Nil), v)
   =   (selon 1ère clause de at)
       at(t, v)
   ```

   Pour le côté droit on a :

   ```
       at(t, Nil ::: v)
   =   (selon le lemme 1)
       at(t, v)
   ```

2. Etape d'induction : `u = i :: rest`

   On suppose que l'égalité est vérifiée pour `rest` et on veut montrer qu'elle l'est également pour `u` :

   $\forall$ `v`, $\forall$ `t`, `at(at(t, rest), v) = at(t, rest ::: v)`

   Pour le côté gauche on a :

   ```
       at(at(t, i :: rest), v)
   =   (selon 2ème clause de at)
       at(at(ts(i), rest), v)
   =   (selon l'hypothèse d'induction)
       at(ts(i), rest ::: v)
   ```

   Pour le côté droit on a :

   ```
       at(t, (i :: rest) ::: v)
   =   (selon lemme 2)
       at(t, i :: (rest ::: v))
   =   (selon 2ème clause de at)
       at(ts(i), rest ::: v)
   ```

   CQFD

Exercice 3 : Parcourir un arbre binaire (15 points)

Partie 1

(*7 points*) La méthode `fold` s'écrit comme suit :

```
abstract class Tree[+A] {
  def fold[B](z: B)(f: (A, B, B) => B): B = this match {
    case Empty => z
    case Node(x, left, right) =>
      f(x, left.fold(z)(f), right.fold(z)(f));
  }
}
case object Empty extends Tree[All];
case class Node[A](x: A, left: Tree[A], right: Tree[A]) extends Tree[A];
```

Partie 2

(*8 points*) La fonction `collect` utilisant `fold` s'écrit comme suit :

```
val PREORDER = -1; val INORDER = 0; val POSTORDER = 1;

def collect[A](t: Tree[A], order: Int): List[A] = {
  val f = (x: A, l: List[A], r: List[A]) =>
    if (order < 0) x :: l ::: r
    else if (order > 0) l ::: r ::: List(x)
    else l ::: List(x) ::: r;
  t.fold(List[A]())(f);
}
```

Dans la donnée de l'exercice on vous demande d'utiliser `fold` ; sans cette contrainte la fonction `collect` pourrait également s'écrire ainsi :

```
def collect[A](t: Tree[A], order: Int): List[A] = t match {
  case Empty => List()
  case Node(x, left, right) =>
    val l = collect(left, order);
    val r = collect(right, order);
    if (order < 0) x :: l ::: r
    else if (order > 0) l ::: r ::: List(x)
    else l ::: List(x) ::: r;
}
```

Exercice 4 : Lisp et Prolog (10 points)

Partie 1

(*5 points*) En Lisp la fonction `flatMap` s'écrit comme suit :

```
(define (flatMap xs f)
  (if (null? xs)
    (quote ())
    (append (f (car xs))
            (flatMap (cdr xs) f))))
```

Voici une session avec l'interpréteur Mini-Lisp écrit en Scala :

```
Mini-Lisp Interpreter (type ",?" for help)
lisp> ,ltest/lists.lisp
List()
lisp> (flatMap '(1 2 3 4) (lambda (x) (cons x (cons (* 10 x) '()))))
List(1,10,2,20,3,30,4,40)
lisp> (flatMap '((1 2) (3 4)) (lambda (xs) (map xs (lambda (x) (+ x 1)))))
List(2,3,4,5)
lisp>
```

Partie 2

(*5 points*) En Prolog le prédicat `insert` se définit ainsi :

```
leq(o, N).
leq(s(M), s(N)) :- leq(M, N).

insert(X, [], [X]).
insert(X, [Y|Ys], [X,Y|Ys]) :- leq(X, Y).
insert(X, [Y|Ys], [Y|Zs]) :- insert(X, Ys, Zs).

insert2(X, [], [X]).
insert2(X, [Y|Ys], [X|Zs]) :- leq(X, Y), append([Y], Ys, Zs).
insert2(X, [Y|Ys], [Y|Zs]) :- insert2(X, Ys, Zs).
```

Voici une session avec l'interpréteur Mini-Prolog écrit en Scala :

```
Mini-Prolog Interpreter (type ":?" for help)
prolog> :ltest/insert.pl
prolog> ?insert(s(s(o)), [], X).
List(X = [s(s(o))])
prolog> ?insert(s(s(o)), [s(s(s(o))),s(s(s(s(o))))], X).
List(X = [s(s(o)), s(s(s(o))), s(s(s(s(o))))])
prolog> ?insert(s(o),X,[o,s(o),s(s(o))]).
List(X = [o, s(s(o))])
prolog> ?insert2(s(o),X,[o,s(o),s(s(o))]).
List(X = [o, s(s(o))])
```

Exercice 5 : Neurones impulsionnels (10 points)

La classe `Neuron` pourrait par exemple être implémentée comme suit.

```
class Neuron(name: String, threshold: Int, outputIntensity: Int) {
  var outputNeurons: List[Neuron] = Nil;
  var excitationLevel: Int = 0;
  def receiveSpike(intensity: Int): Unit = {
    Console.println(name + " => received " + intensity);
    if ((excitationLevel + intensity) >= threshold) {
      excitationLevel = 0;
      outputNeurons.foreach(neuron: Neuron =>
        neuron.receiveSpike(outputIntensity));
    } else {
      excitationLevel = excitationLevel + intensity;
    }
  }
  def connect(neuron: Neuron): Unit =
    outputNeurons = neuron :: outputNeurons;
}
```