Ficha 3

Exercicio 1

1.1a)

```
def factorialA(n: Int) : Int =
  n match {
  case 0 => 1
 case x => x * factorialA(n-1)
}
```

1.1b)

```
def factorialB(n: Int): Int =
  if(n == 0)
    1
 else
  n * factorialB(n-1)
```

1.1c)

```
import scala.annotation.tailrec
def factorial(n: Int): Int = {
  @tailrec
 def loop(acc: Int, n: Int): Int =
    if(n == 0)
      acc
    else
   loop(acc * n, n-1)
  loop(1, n)
}
```

1.2a)

```
def compressRecursive[A](ls: List[A]): List[A] =
  ls match {
    case Nil => Nil
  case h :: tail => h :: compressRecursive(tail.dropWhile(_ ==
h))
  }
```

1.2b)

```
def compressTailRecursive[A](ls: List[A]): List[A] = {
  def compressR(result: List[A], curList: List[A]): List[A] =
    curList match
  {
      case Nil => result.reverse
      case h :: tail => compressR(h :: result, tail.dropWhile(_
== h))
    }
```

```
  compressR(Nil, ls)
}
```

OU

```
def compressTailRecursive[A](ls: List[A]): List[A] = {
  def compressR(result: List[A], curList: List[A]): List[A] =
    curList match
  {
      case Nil => result
      case h :: tail => compressR(result:+h, tail.dropWhile(_ ==
h))
    }
  compressR(Nil, ls)
}
```

Exercicio 2

```
> lazyListRange(1,100).take(3).toList
1
2
3
val res8: List[Int] = List(1, 2, 3)

> listRange(1,100).take(3).toList
1
.
.
.
100
val res11: List[Int] = List(1, 2, 3)
Because the the elements of the LazyList are only create when they are evaluated
```

Exercicio 3

3a)

```
def addPairwise(a: List[Int], b: List[Int]): List[Int] =
   (a,b) match
 {
     case (Nil, _) => Nil
  case (_, Nil) => Nil
  case ((h1::t1), (h2::t2)) => (h1+h2::addPairwise(t1,t2))
   }
```

3b)

```
def zipWith[A,B,C](a: List[A], b: List[B], f: (A,B) => C): List[C] =
   (a,b) match
 {
     case (Nil, _) => Nil
  case (_, Nil) => Nil
  case ((h1::t1), (h2::t2)) => (f(h1,h2)::zipWith(t1,t2,f))
   }
```

OR

```
def zipWith[A,B,C](a: List[A], b: List[B])(f: (A,B) => C): List[C] =
   (a,b) match
  {
      case (Nil, _) => Nil
   case (_, Nil) => Nil
   case ((h1::t1), (h2::t2)) => (f(h1,h2)::zipWith(t1,t2)(f))
   }
```

3c)

```
def isSorted[A](lst: List[A], ordered: (A,A) => Boolean): Boolean = {
   lst match {
      case Nil => true
   case List(x) => true
   case (x::y::ys) => ordered(x,y) && isSorted(y::ys,ordered)
   }
}
```

3d)

```
def getLargest(data: List[Int], f: (Int, Int) => Boolean): (Int, List[Int]) =
   data match {
      case Nil => (0, Nil)
      case head :: Nil => (head, Nil)
      case head :: tail => val (large, remaining) = getLargest(tail,
f)
        if ( f(large,head))
           (large, head :: remaining)
        else
     (head, large :: remaining)
   }

def bubbleSort(data: List[Int], f: (Int, Int) => Boolean ): List[Int] =
   data match {
      case Nil => Nil
   case _=> val (greatest, tail) = getLargest(data, f)
        bubbleSort(tail, f) ::: List(greatest)
}
```

OR

```
def bubbleSort(data: List[Int], f: (Int, Int) => Boolean): List[Int] = {
  def aux(data1: List[Int]):List[Int] = {
    data1 match {
      case Nil => Nil
      case List(x) => List(x)
      case a :: b :: xs => {
        if (f(a, b)) a::aux(b::xs) else b::aux(a::xs)
      }
```

```
  }
 }
 val res=aux(data)
 if(isSorted(res,f))
  res
 else
  bubbleSort(res.init,f):+res.last
}
```

Exercicio 4

a)

recursive function patterns: Filtering

```
def paresord(lst: List[(Int,Int)]): List[(Int,Int)] =
{
  lst match {
    case Nil => Nil
  case x ::xs => {
      if( x._1 < x._2)
        x :: paresord(xs)
      else
    paresord(xs)
     }
  }
}
```

OR

```
def paresord(lst: List[(Int,Int)]): List[(Int,Int)] = lst filter (x =>
x._1 < x._2)
```

b)

recursive function patterns: folding

```
def myconcat(lst:List[String]) : String =
{
  lst match {
    case Nil => ""
  case x ::xs => x ++ myconcat(xs)
  }
}
```

OR

```
def myconcat(xs: List[String]) = (xs foldLeft "") (_ ++ _)
```

c)

recursive function patterns: mapping

```
def maximum(lst:List[(Double, Double)]): List[Double] =
{
```

```
  lst match {
    case Nil => Nil
  case x :: xs =>
      if(x._1 > x._2)
        x._1 :: maximum(xs)
      else
    x._2 :: maximum(xs)
  }
}
```

OR

```
def maximum(lst:List[(Double, Double)]): List[Double] = lst map (x
=> if(x._1 > x._2) x._1 else x._2)
```

Exercicio 6

```
def indicative(ind:String, telefs:List[String]): List[String] =
{
  telefs match
 {
    case Nil => List()
    case x :: xs =>
      if(x. substring(0,ind.length).equals(ind))
        x :: indicative(ind, xs)
      else
   indicative(ind, xs)
  }
}
```

OR

```
def indicative(ind: List[Char], telefs:List[List[Char]]): List[List[Char]]
=
{
  def concorda(s:List[Char], s1:List[Char]):Boolean =
  {
     (s,s1) match
  {
      case (Nil,_) => true
   case (x::xs, y::ys) => (x==y) && concorda(xs,ys)
      case (x::xs, Nil) => false
  }
  }

  telefs match
 {
    case Nil => List()
    case x::xs =>
      if(concorda(ind, x))
        x :: indicative(ind, xs)
      else
    indicative(ind, xs)
```

```
  }
}
```

Exercicio 7

```
def abrev(l:List[String]): List[String] = l map (x => s"${x.head}.
${x.split(" ").last}")
```

OR

```
def abrev(l:List[String]): List[String] =
{
  def conv(s1: String):String =
  {
    val ns = s1.split(" ")
    if(ns.length > 1)
      s"${ns.head.head}. ${ns.last}"
  else
   s1
  }
  l map conv
}
```

OR

```
def abrev(l:List[String]): List[String] =
{
  def conv(s: String):String =
  {
    val ns = s.split(" ")
    if(ns.length > 1)
      ns.head.head+". "+ns.last
    else
   s
  }

  l map conv
}
```