```scala
object ejlista2020 {
  println("Welcome to the Scala worksheet")       //> Welcome to the Scala worksheet

// You are not permitted to use these List methods:
// * length
// * map
// * filter
// * ::: (and variations such as ++)
// * flatten
// * flatMap
// * reverse (and variations i.e. reverseMap, reverse_:::)
// This also means you are not permitted to use for-comprehensions on Lists.
// You are permitted to use the functions you write yourself. For example, Exercise 2 may use Exercise 1 or Exercise 3.
// Using permitted existing methods where appropriate will attract marks for elegance.

  // Ejercicio 0: Realiza dos funciones que dado un entero devuelva el siguiente y el anterior

  def succ(n: Int) = n + 1                        //> succ: (n: Int)Int
  def pred(n: Int) = n - 1                        //> pred: (n: Int)Int

 succ(10)                                         //> res0: Int = 11
 pred(10)                                         //> res1: Int = 9


  // Ejercicio 1: Realiza una función que sume dos numeros

  def add(x: Int, y: Int): Int = x match {
    case 0 => if(y == 0) 0 else add(y, x)
    case _ => if(x > 0) succ(add(pred(x), y))
              else      pred(add(succ(x), y))
  }                                               //> add: (x: Int, y: Int)Int
 println(add(10,2))                               //> 12
 add(10,0)                                        //> res2: Int = 10
 add(0,5)                                         //> res3: Int = 5


  // Ejercicio 2: Realice una función que sume los elementos de una lista

  def sum(x: List[Int]): Int = x match {
    case head :: Nil  => add(head, 0)
    case head :: tail => add(head, sum(tail))
  }                                               //> sum: (x: List[Int])Int
sum(List(10,2,3,5,6))                            //> res4: Int = 26
```

```scala
  // Ejercicio 3: Realice una función que calcule la longitud de una lista

  def length(x: List[Int]): Int = x match {
    case Nil => 0
    case head :: tail => succ(length(tail))
  }                                               //> length: (x: List[Int])Int
length(List(10,2,3,5,6))                         //> res5: Int = 5




 // Ejercicio 4: Realice una función que multiplique los elementos de una lista por su cuadrado

  def map(x: List[Int], f: Int => Int): List[Int] = x match {
    case Nil => Nil
    case head :: tail => f(head) :: map(tail, f)
  }                                               //> map: (x: List[Int], f: Int => Int)List[Int]

  map(List(1,2,3,4,5), (x:Int) => x * x)          //> res6: List[Int] = List(1, 4, 9, 16, 25)




  // Ejercicio 5 Realice una función que obtenga los elementos mayores a un numero

  def filter(x: List[Int], f: Int => Boolean): List[Int] = x match {
    case Nil => Nil
    case head :: tail => if(f(head)) head :: filter(tail, f)
                         else        filter(tail, f)
  }                                               //> filter: (x: List[Int], f: Int => Boolean)List[Int]


 filter(List(1,2,3,4,5), (x:Int) => x > 3)        //> res7: List[Int] = List(4, 5)
```

```scala
  // Ejercicio 6 Realice una función que concadene dos listas

  def append (x: List[Int], y: List[Int]): List[Int] = x match {
    case Nil => y
    case head :: Nil  => head :: y
    case head :: tail => head :: append(tail, y)
  }                                                   //> append: (x: List[Int], y: List[Int])List[Int]
  append(List(1,9,3), List(4,5,6))                    //> res8: List[Int] = List(1, 9, 3, 4, 5, 6)




  // Ejercicio 7 : Realice una función que concadene una lista de listas

  def concat (x: List[List[Int]]): List[Int] = x match {
    case head :: Nil => head
    case head :: tail => append(head, concat(tail))
  }                                                   //> concat: (x: List[List[Int]])List[Int]

concat(List(List(1,2,3), List(4,5,6), List(7,8,9)))
                                                      //> res9: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)




  // Ejercicio 8 Realice una función que enlace tres listas, la primera con los elementos de la lista, las segunda multiplicando x2 todos
los elementos y por ultimo multiplicando x3

  def concatMap(x: List[Int], f: Int => List[Int]): List[Int] = x match {
    case head :: Nil => f(head)
    case head :: tail => append(f(head), concatMap(tail, f))
  }                                                   //> concatMap: (x: List[Int], f: Int => List[Int])List[Int]
  concatMap(List(1,2,3,4,5,6), (x:Int) => List(x, x * 2, x * 3))
                                                      //> res10: List[Int] = List(1, 2, 3, 2, 4, 6, 3, 6, 9, 4, 8, 12, 5, 10, 15, 6,
                                                      //| 12, 18)
```

```scala
  // Ejercicio 9 Realice una función que devuelva el mayor de una lista

  def maximum(x: List[Int]): Int = x match {
    case head :: Nil => head
    case head :: tail => {
      val max_tail = maximum(tail)
      if(head > max_tail) head else max_tail
    }
  }                                                 //> maximum: (x: List[Int])Int

maximum (1::2::3::Nil)                              //> res11: Int = 3



  // Ejercicio 10 Realice una función que realice el reverso de una lista

  def reverse(x: List[Int]): List[Int] = x match {
    case head :: tail => append(reverse(tail), List(head))
    case Nil => Nil
  }                                                 //> reverse: (x: List[Int])List[Int]

  reverse (1::2::3::Nil)                            //> res12: List[Int] = List(3, 2, 1)
```

```
 //Otros ejemplos


   def merge(xs: List[Int], ys: List[Int]): List[Int] = xs match {
       case Nil => ys
       case x::xs1 => ys match {
       case Nil => xs
       case y :: ys1 => if (x < y) x :: merge(xs1, ys)
                 else y :: merge(xs, ys1)
  }
  }                                                  //> merge: (xs: List[Int], ys: List[Int])List[Int]

  merge (List(1,6,7,4),List(5,2,3,8))                //> res13: List[Int] = List(1, 5, 2, 3, 6, 7, 4, 8)

  val xs = List(-5, 6, 3, 2, 7)                      //> xs  : List[Int] = List(-5, 6, 3, 2, 7)
  val fruit = List("apple", "pear", "orange", "pineapple")
                                                     //> fruit  : List[String] = List(apple, pear, orange, pineapple)
```

```
//Numero de elementos de una lista:
def numElems(lista:List[Int]):Int =
   if (lista.isEmpty) 0 else
      1 + numElems(lista.tail)                     //> numElems: (lista: List[Int])Int
```

```
//Insercion en una lista ordenada:
def insert(x: Int, lista: List[Int]) : List[Int] =
   if (lista == Nil) x :: Nil else
      if (x < lista.head) x :: lista else
         lista.head :: insert(x, lista.tail)      //> insert: (x: Int, lista: List[Int])List[Int]
```

```scala
//Ordenacion de una lista:
def sort(lista: List[Int]): List[Int] =
   if (lista.isEmpty) Nil else
      insert(lista.head, sort(lista.tail))        //> sort: (lista: List[Int])List[Int]




//Reverse-list
def reverse1(lista: List[Int]) : List[Int] =
   if (lista.isEmpty) Nil else
      reverse1(lista.tail) ::: List(lista.head)   //> reverse1: (lista: List[Int])List[Int]

val lista = List(9, 8, 7, 17, 21)                 //> lista  : List[Int] = List(9, 8, 7, 17, 21)

numElems(lista)                                   //> res14: Int = 5
insert(14,lista)                                  //> res15: List[Int] = List(9, 8, 7, 14, 17, 21)
sort(lista)                                       //> res16: List[Int] = List(7, 8, 9, 17, 21)
reverse(lista)                                    //> res17: List[Int] = List(21, 17, 7, 8, 9)


  }
```