Exercice10 :

1. Essayons d'utiliser des boucles dans une fonction (par exemple, une qui imprime des nombres impairs de 0 à 200).

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val listOfNumbers = List(0 to 200)
for (i<-listOfNumbers){
if(i%2!=0){
println(i)
}
else{
println("pas de nombre impaire")
}
}

// Exiting paste mode, now interpreting.
```

2. Explorons la boucle for() ou while()

- for()

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

// Scala program to illustrate while loop
object whileLoopDemo
{

    // Main method
    def main(args: Array[String])
    {
        var x = 1;

        // Exit when x becomes greater than 4
        while (x <= 4)
        {
            println("Value of x: " + x);

            // Increment the value of x for
            // next iteration
            x = x + 1;
        }
    }
}

// Exiting paste mode, now interpreting.

defined object whileLoopDemo
```

- while()

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

// Scala program to illustrate for loop
object forloopDemo {

   // Main Method
   def main(args: Array[String]) {

      var y = 0;

      // for loop execution with range
      for(y <- 1 to 7)
      {
         println("Value of y is: " + y);
      }
   }
}

// Exiting paste mode, now interpreting.

defined object forloopDemo

scala>
```

3. Essayons d'en imbriquer une boucle for dans une autre.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

object MyClass {
      def main(args: Array[String]) {
            var i , j = 0;
            for(i <- 1 to 2){
                for(j <- 1 to 2)
                    println("(" + i + "," + j + ")")
            }
        }
   }

// Exiting paste mode, now interpreting.

defined object MyClass

scala>
```

4. Essayons les bons vieux algorithmes de tri (par exemple, le tri à bulles, le tri par fusion) en utilisant for() et while() en scala.

- Bubble Sort

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def getLargest[T <% Ordered[T]](data: List[T]): (T, List[T]) =
  data match {
    case head :: Nil  => (head, Nil)
    case head :: tail =>
      val (large, remaining) = getLargest(tail)
      if (large > head)
        (large, head :: remaining)
      else
        (head, large :: remaining)
  }

def bubbleSort[T <% Ordered[T]](data: List[T]): List[T] =
  data match {
    case Nil => Nil
    case _   =>
      val (greatest, tail) = getLargest(data)
      bubbleSort(tail) ::: List(greatest)
  }

// Exiting paste mode, now interpreting.

<pastie>:12: warning: match may not be exhaustive.
It would fail on the following input: Nil
         data match {
         ^
getLargest: [T](data: List[T])(implicit evidence$1: T => Ordered[T])(T, List[T])
bubbleSort: [T](data: List[T])(implicit evidence$2: T => Ordered[T])List[T]

scala>
```

- Insertion Sort

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def insertElement[T <% Ordered[T]](elm: T, sorted: List[T]): List[T] =
  sorted match {
    case Nil => elm :: sorted
    case head :: tail if head < elm => head :: insertElement(elm, tail)
    case _ => elm :: sorted
  }

def insertionSort[T <% Ordered[T]](list: List[T]): List[T] =
  list match {
    case Nil => list
    case head :: tail =>
      val sorted = insertionSort(tail)
      insertElement(head, sorted)
  }

// Exiting paste mode, now interpreting.

insertElement: [T](elm: T, sorted: List[T])(implicit evidence$1: T => Ordered[T])List[T]
insertionSort: [T](list: List[T])(implicit evidence$2: T => Ordered[T])List[T]

scala>
```

- Merge Sort

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def split[T <% Ordered[T]](list: List[T]): (List[T], List[T]) =
  list match {
    case Nil => (Nil, Nil)
    case head :: Nil => (head :: Nil, Nil)
    case first :: second :: tail =>
      val (tl1, tl2) = split(tail)
      (first :: tl1, second :: tl2)
  }

def merge[T <% Ordered[T]](list1: List[T], list2: List[T]): List[T] =
  (list1, list2) match {
    case (x, Nil) => x
    case (Nil, y) => y
    case (flh :: flt, slh :: slt) =>
         if (flh > slh)
         slh :: merge(list1, slt)
      else
         flh :: merge(flt, list2)
  }

def mergeSort[T <% Ordered[T]](list: List[T]): List[T] =
  list match {
    case Nil | _ :: Nil =>
      list
    case _ =>
      val (part1, part2) = split(list) //list.splitAt(list.length / 2)
      val sorted1 = mergeSort(part1)
      val sorted2 = mergeSort(part2)
      merge(sorted1, sorted2)
  }

// Exiting paste mode, now interpreting.
```

- Quick Sort

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def partition[T <% Ordered[T]](elm: T, list: List[T]): (List[T], List[T]) = {
  doPartition(elm, list, Nil, Nil)

  def doPartition[T <% Ordered[T]](elm: T, list: List[T], smallerItems: List[T], largerItems: List[T]): (List[T], List[T]) =
    list match {
      case Nil => (smallerItems, largerItems)
      case head :: tail =>
        if (head < elm)
          doPartition(elm, tail, head :: smallerItems, largerItems)
        else
          doPartition(elm, tail, smallerItems, head :: largerItems)
    }
}

def quickSort[T <% Ordered[T]](data: List[T]): List[T] =
  data match {
    case Nil          => Nil
    case head :: Nil  => List(head)
    case head :: tail =>
      val (list1, list2) = partition(head, tail)
      val smallerList = quickSort(list1)
      val biggerList = quickSort(list2)
      smallerList ::: (head :: biggerList)
  }

// Exiting paste mode, now interpreting.
```