Task 1 a)

```
object Task1A extends App {
    var arr:Array[Int] = new Array[Int](50)
    for (i <- 1 to 50) {
        arr(i-1) = i
    }
}
```

Task 1 b)

```
// Create a function that sums the elements in an array of integers using a for loop.
def sum(array : Array[Int]): Int = {
  var sum = 0;

  for(element <- array) {
    sum += element;
  }

  return sum
}
```

Task 1 c)

```
object Task1C extends App {

    var arr = Array(1, 2, 3, 4, 5)
    var l = arr.length

    def sum_recursively(array : Array[Int], len : Int): Int = {
        if (len == 1) {
            return array(0)
        }
        else {
            return sum_recursively(array, len-1) + array(len-1)
        }
    }
    println(sum_recursively(arr, l))
}
```

Task 1 d)

```
//Create a function to compute the nth Fibonacci number using recursion without using memoization
//(or other optimizations). Use BigInt instead of Int. What is the difference between these two data
//types?
def fib(n : BigInt) : BigInt = {
  if(n == 1 || n == 2) {
    return 1;
  }
  else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

Int type is capable of storing 32 bit signed values, ranging from -2147483648 to 2147483647.
BigInt type is capable of storing arbitrarily big signed values

Task 2 a)

```
object Task2A extends App {

    def first_foo(): Unit = println("hi")

    def second_foo(input_foo: => Unit): Thread = {
        val t = new Thread {
            ↑ run
            override def run() = input_foo
        }
        t
    }
}
```

Task 2 b)

```
object Task2B extends App {
  private var counter: Int = 0

  for(i <- 1 to 2) {
    new Thread() {
      override def run() : Unit = {
        increaseCounter();
      }
    }.start();
  }

  new Thread() {
    override def run() : Unit = {
      printCounter();
    }
  }.start();

  def increaseCounter(): Unit = {
    counter += 1
  }

  def printCounter(): Unit = {
    println("Current counter: " + counter);
  }

}
```

The printCounter function prints either 0, 1 or 2.
This phenomenon is called race condition.
It can be problematic if, for example, 2 threads are incrementing the amount of money a customer has. It might happen that only 1 of the increments gets written, and that would make customers mad.

Task 2 c)

```
object Task2C extends App {

    private var counter: Int = 0

    for(i <- 1 to 2) {
        new Thread() {
            ↑ run
            override def run() : Unit = {
                increaseCounter();
            }
        }.start();
    }

    new Thread() {
        ↑ run
        override def run() : Unit = {
            printCounter();
        }
    }.start();

    def increaseCounter(): Unit = synchronized {
        counter += 1
    }

    def printCounter(): Unit = {
        println("Current counter: " + counter);
    }
}
```

Task 2 d)

A deadlock occurs when a thread waits for a lock to be available but the lock will never be free, making the thread wait forever.
The following example has a chance to deadlock:

```scala
object Task2D extends App {
  object Foo1 {
    lazy val value = 5
    lazy val a: Int = Foo2.b
  }

  object Foo2 {
    lazy val b: Int = Foo1.value
  }

  val thread1 = new Thread() {
    override def run(): Unit = {
      println(Foo1.a)
    }
  }
  val thread2 = new Thread() {
    override def run(): Unit = {
      println(Foo2.b)
    }
  }

  thread1.start();
  thread2.start();

  thread1.join();
  thread2.join();
}
```