Exercise 1

```scala
def topk(xs: Array[Int], k: Int): List[Int] = {
  def mergek(as: List[Int], bs: List[Int],
             m: Int, acc: List[Int]): List[Int] =

    if (m == 0)
      acc.reverse
    else
      (as, bs) match {
        case (a :: as0, b :: bs0) =>
          if (a >= b) mergek(as0, bs, m-1, a :: acc)
          else        mergek(as, bs0, m-1, b :: acc)
        case (a :: as0, Nil) => mergek(as0, bs, m-1, a :: acc)
        case (Nil, b :: bs0) => mergek(as, bs0, m-1, b :: acc)
      }

  def recurse(from: Int, until: Int): List[Int] = {
    val len = until - from
    if (len <= k) {
      slice(xs, from, until).sorted.reverse
    } else {
      val mid = len / 2 + from
      val (as, bs) = parallel(recurse(from, mid), recurse(mid, until))
      mergek(as, bs, k, List.empty)
    }
  }

  recurse(0, xs.length)
}
// => Complexity: O(k*log k + k*log n) = O(k * log n)
```

Exercise 2

```
// Question 2a
def getCoordinates(moves: ParSeq[Direction], start: Vector2D): Vector2D =
      start + moves.aggregate(Vector2D(0, 0))(_ + _.toVector, _ + _)
```

```
// Question 2b
case class Vector2DWithBounds(x: BigInt, minX: BigInt, maxX: BigInt,
                              y: BigInt, minY: BigInt, maxY: BigInt) {
  def +(that: Vector2DWithBounds): Vector2DWithBounds =
    Vector2DWithBounds(
      this.x + that.x,
      this.minX min (this.x + that.minX),
      this.maxX max (this.x + that.maxX),
      this.y + that.y,
      this.minY min (this.y + that.minY),
      this.maxX max (this.y + that.maxY))

  def +(that: Vector2D): Vector2DWithBounds = this + addBounds(that)
}

def addBounds(vector: Vector2D) =
  Vector2DWithBounds(
    vector.x,
    vector.x min 0,
    vector.x max 0,
    vector.y,
    vector.y min 0,
    vector.y max 0)

def getCoordinates(moves: ParSeq[Direction], start: Vector2D, size: BigInt):
Option[Vector2D] = {
  val vectorWithBounds =
    moves.aggregate(addBounds(Vector2D(0, 0)))(_ + _.toVector, _ + _)

  val x = start.x + vectorWithBounds.x
  val minX = start.x + vectorWithBounds.minX
  val maxX = start.x + vectorWithBounds.maxX
  val y = start.y + vectorWithBounds.y
  val minY = start.y + vectorWithBounds.minY
  val maxY = start.y + vectorWithBounds.maxY

  if (minX < 0 || maxX > size || minY < 0 || maxY > size) None
  else Some(Vector2D(x, y))
```

Exercise 3

```
// Question 3a
object HelveticaBotAccess {
  private val maxClients = 10
  private val jobSemaphore = new Semaphore(maxClients)

  def runJob(f: () => Unit): Boolean = {
    jobSemaphore.decrement()

    val success = HelveticaBot.runJob(f)

    jobSemaphore.increment()

    success
  }
}
```

```
// Question 3b
object HelveticaBotAccess {
  private val maxClients = 10
  private val jobSemaphore = new Semaphore(maxClients)
  private val jobInitSemaphore = new Semaphore(1)
  private val exclusiveSemaphore = new Semaphore(1)

  def runJob(f: () => Unit): Boolean = {

    jobInitSemaphore.decrement()
    val count1 = jobSemaphore.decrement()
    if (count1 == maxClients - 1) {
      exclusiveSemaphore.decrement()
    }
    jobInitSemaphore.increment()

    val success = HelveticaBot.runJob(f)

    val count2 = jobSemaphore.increment()
    if (count2 == maxClients) {
      exclusiveSemaphore.increment()
    }

    success
  }

  def runExclusiveJob(f: () => Unit): Boolean = {
    exclusiveSemaphore.decrement()

    val success = HelveticaBot.runJob(f)

    exclusiveSemaphore.increment()

    success
  }
}
```

```scala
// Question 3c
class Semaphore(initialValue: Int) {
  assert(initialValue >= 0)
  private var value = initialValue

  def decrement(): Int = synchronized {
    while (value == 0)
      wait()
    value -= 1
    value
  }

  def increment(): Int = synchronized {
    value += 1
    if (value == 1)
      notifyAll()
    value
  }
}
```