121.

Implement a generic `sort` function that can sort any list of objects that are comparable (meaning that they inherit from `Comparable`). Please use the quicksort as written below. Instead of comparing elements using '<', use the `compareTo` method.

```
def qsort(xs: IntList): IntList = xs match {
     case Nil => Nil
     case Cons(y, ys) =>
          val (smaller, rest) = partition(ys, t => t < y)
     concat(qsort(smaller), Cons(y, qsort(rest)))
}
```

122.

Implement the `filter` and `zip` methods of `Stream`. (Some design choices to consider: Do your methods compute the head or the tail first in each step? Can you avoid computing the stream elements multiple times?) Then run the code and check that it works as expected. Also, explain how `primes` and `fibs2` work. Please implement the code in the scalafile added to this order.

123.

The inverse of `foldRight` is `unfoldRight`. For streams, `unfoldRight` looks as follows:

```
 def unfoldRight[A, S](z: S, f: S => Option[(A, S)]): Stream[A] =
   f(z) match {
     case Some((h, s)) => SCons(() => h, () => unfoldRight(s, f))
     case None => SNil
   }
```

The `foldRight` operation traverses a given stream, whereas `unfoldRight` can *produce* streams. Use `unfoldRight` to re-implement the streams ones, nats and fibs more concisely. In each case, your solution should consist of a single call to `unfoldRight`, which then takes care of building the stream. (Hint: `unfoldRight` can build both finite and infinite streams; for this exercise you only need the latter.)