Solution Review: Merge Sort

Let's go over the solution of solving MergeSort through concurrency.

```
package main
import "fmt"

func Merge(left, right [] int) [] int{
  merged := make([] int, 0, len(left) + len(right))
  for len(left) > 0 || len(right) > 0{
    if len(left) == 0 {
      return append(merged,right...)
    }else if len(right) == 0 {
      return append(merged,left...)
    }else if left[0] < right[0] {
      merged = append(merged, left[0])
      left = left[1:]
    }else{
      merged = append(merged, right [0])
      right = right[1:]
    }
  }
  return merged
}

func MergeSort(data [] int) [] int {
  if len(data) <= 1 {
    return data
  }
  done := make(chan bool)
  mid := len(data)/2
  var left [] int

  go func(){
    left = MergeSort(data[:mid])
    done <- true
  }()
  right := MergeSort(data[mid:])
  <-done
  return Merge(left,right)

}

func main(){
  data := [] int{9,4,3,6,1,2,10,5,7,8}
  fmt.Printf("%v\n%v\n", data, MergeSort(data))

}
```

Let's go over the changes we made to the `MergeSort` function.

Firstly, in merge sort, we keep dividing our array recursively into the `right` side and the `left` side and call the `MergeSort` function on both sides from **line 30** to **line 34**. You will note that these two calls can be made independently and we can execute one of them in a goroutine:

```
func MergeSort(data [] int) [] int {
  if len(data) <= 1 {
    return data
  }
  done := make(chan bool)
  mid := len(data)/2
  var left [] int

  go func(){
    left = MergeSort(data[:mid])
    done <- true
  }()
  right := MergeSort(data[mid:])
  <-done
  return Merge(left,right)

}
```

Adding code to goroutine

Now we have to make sure that `Merge(left,right)` is executed only once we get the return values from both the recursive calls, i.e. both the `left` and `right` have been updated before `Merge(left,right)` has to execute. Hence, we introduce a channel of type `bool` on **line 26** and send `true` on it as soon as `left = MergeSort(data[:mid])` is executed (**line 32**). The `<-done` operation blocks the code on **line 35** before the statement `Merge(left,right)` so that it does not proceed until our goroutine has finished. After the goroutine has finished and we receive `true` on the `done` channel, the code proceeds forward to `Merge(left,right)` statement on **line 36**.

```
func MergeSort(data [] int) [] int {
  if len(data) <= 1 {
    return data
  }
  done := make(chan bool)
  mid := len(data)/2
  var left [] int

  go func(){
```

```
    left = MergeSort(data[:mid])
    done <- true
  }()
  right := MergeSort(data[mid:])
  <-done
  return Merge(left,right)

}
```

Adding `done` Channel

Easy Peasy, right?

You will be able to appreciate how simply we divided our algorithm into independent chunks of code to execute using goroutines and synchronize the concurrent operations using channels.

See you in the next lesson!