Problem Set 1: Go

The following questions are meant as practice for learning some of the basic features of Go. Write all your solution code in a package named `ps1` (i.e. the first line of your Go program files should be `package ps1`).

Each question will be marked as follows:

- 1 mark for correctness of the function
- 1 mark for writing a Go-style test function that will run when the command "go test" is called and automatically run test cases on the function to help verify its corrrectness.

  Also, to get this mark, we want to see that you have used Go features in appropriate way, and have not done something extremely inefficient, or that is bad in some significant way.

Your test functions must be completely automated, i.e. no human intervention should be needed to check that the output is correct. Use if- statements to ensure that your functions are returning the correct results.

You may import any standard Go packages you need in this program. Write all other code yourself: don' t use any packages or code from outside of the Go standard library.

Please submit your final files on Canvas in a zip folder called `ps1.zip`. You will need at least two Go files: one for the functions, and one for the testing code that will be run when "go test" is called.

Hint: Begin by learning how to use "go test" , and write the test functions as you go.

1. (2 marks) Implement a function called `countPrimes(n)` that returns the number of primes less than, or equal to, the `int n`. For example, `countPrimes(4)` should return 2, `countPrimes(10000)` should return 1229, and `countPrimes(-6)` should return 0.

2. (2 marks) Implement a function called `countStrings(filename)` that takes the name of a text file as input, and returns a `map[string]int` whose keys are all the different strings in the file, and the corresponding values are the number of occurrences of the key in the file.

   For example, suppose you give your function this text file:

   ```
   The big big dog
   ate the big apple
   ```

   Then it should return this `map[string]int`:

   ```
   {"The":1, "the":1, "big":3, "dog":1, "ate":1, "apple":1}
   ```

   Note that *case matters*: strings like `the` and `The`, which differ only in the case of some letters, are considered different strings.

3. (2 marks) Here’ s a simple way to represent a moment in time:

```
type Time24 struct {
    hour, minute, second uint8
}
// 0 <= hour < 24
// 0 <= minute < 60
// 0 <= second < 60
```

Implement the following functions:

- `validTime24(t)` returns `true` if `t` is a valid `Time24` object (i.e. it meets the constraints listed in the comments below the `struct`), and `false` otherwise.

- `equalsTime24(a, b)` returns `true` if `a` and `b` are exactly the same time, and `false` otherwise.

- `lessThanTime24(a, b)` returns `true`, if time `a` comes strictly before `b`, and `false` otherwise.

- `minTime24`, which returns the smallest time in a slice of `Time24` objects. It has this signature:

  ```
  func minTime24(times []Time24) (Time24, error)
  ```

  If `times` is empty, then `Time24{0, 0, 0}` is returned, along with an error object with a helpful message.

  If `times` has one, or more, items, then the smallest time is returned, and the error is `nil`.

4. (2 marks) Continuing the previous question, implement the following two methods (not functions!) on `Time24`:

- A method that converts a `Time24` to human-readable string:

  ```
  func (t Time24) String() string {
      // ...
  }
  ```

  The returned string should have the form “hh:mm:ss” . For example:

  ```
  t := Time24{hour: 5, minute: 39, second: 8}
  fmt.Println(t)  // "05:39:08"
  ```

  Notice that each number is written as 2-digits, possibly with a leading 0. Thus the returned strings will always have the same length.

  Also, notice that we *don’ t* need to call `t.String()` inside `fmt.Println`. That’ s because the signature for `String` implements the fmt.Stringer interface, and functions like `fmt.Print` know to call `String()` on objects that implement it.

- A method called `AddOneHour()` that adds one hour to a time. It should work with code like this:

```
t := Time24{20, 15, 0}
fmt.Println(t)
for i := 0; i < 5; i++ {
    t.AddOneHour()
    fmt.Println(t)
}
```

This prints:

```
20:15:00
21:15:00
22:15:00
23:15:00
00:15:00
01:15:00
```

5. (2 marks) Implement a function the returns all the bit sequences of length n. It should have this signature:

```
func allBitSeqs(n int) [][]int
```

For example:

- `allBitSeqs(1)` returns `[[0] [1]]`.
- `allBitSeqs(2)` returns `[[0 0] [0 1] [1 0] [1 1]]`.
- `allBitSeqs(3)` returns `[[0 0 0] [0 0 1] [0 1 0] [0 1 1] [1 0 0] [1 0 1] [1 1 0] [1 1 1]]`.

The exact order of the slices in the returned `[][]int` doesn’ t matter. So, for example, `allBitSeqs(2)` could instead return `[[1 1] [1 0] [0 0] [0 1]]`.

If `n <= 0`, then return an empty `[][]int`.

Here’ s the definition of a bit sequence:

```
func isBitSeq(seq []int) bool {
    for _, b := range seq {
        if !(b == 0 || b == 1) {
            return false
        }
    }
    return true
}
```

You don’ t necessarily need to use this function in your implementation of `allBitSeqs`. It’ s just a way to define what is meant by a bit sequence for this question.