Assignment 1: Practice with 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 `a1` (i.e. the first line of your Go program files should be `package a1`).

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 correctness.

  Also, to get this second mark, we want to see that you use Go features in appropriate way, and have not done something extremely inefficient, or that is bad in some significant way. The marker will run your code by typing `go test` at the command line, so please be sure your program works correctly.

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 clearly cite the source of any ideas you used in this work.

Please submit your final files on the CS Submission Server as a zipped folder called `a1.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(5)` should return 3, `countPrimes(10000)` should return 1229, and `countPrimes(-6)` should return 0.
2. (2 marks) Implement a function called `countStrings(filename)` that reads the contents of the file named `filename`, and returns a `map[string]int` whose keys are all the different strings in the file, and the corresponding values are the number of times the key occurs in the file.

   For example, suppose you have this text file named `sample.txt`:

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

   Then `countStrings("sample.txt")` 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 and methods:

1. The function `equalsTime24(a, b)` returns `true` if `a` and `b` are exactly the same time, and `false` otherwise.
2. The function `lessThanTime24(a, b)` returns `true`, if time `a` comes strictly before `b`, and `false` otherwise.
3. The method (not a function!) `t.String()` that converts a `Time24` to a human-readable string. It has this signature:

```
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 string will always be the same length.

Also, notice that you *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.

4. The method (not a function!) `t.validTime24()` 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.
5. The function `minTime24`, which returns the earliest 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) Write a function called `linearSearch(x, lst)` that uses linear search to return the first index location of `x` in the slice `lst`. It must work with (at least) both strings and ints as in these examples:
    1. `linearSearch(5, []int{4, 2, -1, 5, 0})` returns 3 and a `nil` error.
    2. `linearSearch(3, []int{4, 2, -1, 5, 0})` returns any integer and and an error object with a helpful message (because 3 is not in the slice).
    3. `linearSearch("egg", []string{"cat", "nose", "egg"})` returns 2 and a `nil` error.
    4. `linearSearch("up", []string{"cat", "nose", "egg"})` returns any integer and and an error object with a helpful message.

    You can use helper functions, but you must have only one function named `linearSearch`. If the type of `x` is not the same as the type of the elements in `lst`, then `linearSearch` should call `panic` with a helpful message.

5. (2 marks) Implement a function the returns all the genetic sequences of length n using the bases A, C, G, and T. It should have this signature:

    ```
    func allGeneSeqs(n int) []string
    ```

    For example:

    1. `allGeneSeqs(1)` returns `[A C T G]`.
    2. `allGeneSeqs(2)` returns `[AA AC AT AG CA CC CT CG TA TC TT TG GA GC GT GG ]`.
    3. `allGeneSeqs(3)` returns `[AAA AAC AAT AAG ACA ACC ACT ACG ATA ATC ATT ATG AGA AGC AGT AGG CAA CAC CAT CAG CCA CCC CCT CCG CTA CTC CTT CTG CGA CGC CGT CGG TAA TAC TAT TAG TCA TCC TCT TCG TTA TTC TTT TTG TGA TGC TGT TGG GAA GAC GAT GAG GCA GCC GCT GCG GTA GTC GTT GTG GGA GGC GGT GGG]`.

    The exact order of the strings in the returned `[]string` doesn't matter. So, for example, `allGeneSeqs(2)` could instead return `[AC AT CA CC CT GT CG AG TA TC TT AA TG GA GC GG ].`

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

    Of course, for large values of `n`, `allGeneSeqs` will take too much time and memory to actually run. Nonetheless, the method you use to calculate the genetic sequences should work with any `int` greater than 0, if given enough time and memory.