Concurrent Network Analysis: Our goal

In this project you will implement the logic behind a network analysis microservice. This particular microservice has the goal of reading IP addresses from a file and reporting the frequencies of visits by each address, the IP address of the most frequent visitor, and the number of visits. We can break this into parts:

part 0: Read IP data from a .txt file into an `[]string array`. This is mostly completed for you. You just have to create and populate the array.

part 1: Create and instantiate our `Counter` struct

part 2: Implement single-threaded counter logic

part 3: Implement concurrent counter logic & add concurrency primitives

part 0: Reading our input data

Part 0 has been completed for you and provided in the skeleton.go file. We use the `os.Open` function and `bufio.Scanner` interface to read data line by line into an `array`. You can study the given code to understand how this works more.

Understanding our data

Network visitors are stored in a `.txt` file, where each line holds one IP and represents one visit from that IP. For example,

```
user@machine $ cat data.txt | head -n 4
44.12.192.8
234.0.0.1
44.12.192.8
43.43.0.87
user@machine $
```

In this small sample, the top visitor is `44.12.192.8` with a frequency of 2

part 1: Our `Counter` struct

As of now, our counter needs to have 3 fields at minimum: a `map` to map `string` IP addresses to their `integer` frequencies, a `string` for the most frequent IP, and an `integer` for its corresponding frequency. Fill in the fields for `Counter` and write a function `NewCounter` which returns a pointer to a new `Counter` instance.

part 2: Using the `Counter` struct

Your task here is to write a method, `Counter.countIP(ip string)` which will increment the visit count for a given `string` IP adress and check if it is now the top visitor, updating those fields as needed.

part 3: Using the `Counter` struct concurrently

As of now, counter is only running on one core. We can speed this up by distributing work to multiple workers and allow them to work at the same time. Go has a ton of built-in support for concurrency, so this is not as hard as it sounds.

part 3a: Adding mutual exclusion to our `Counter`

Because multiple workers will be reading and writing to and from our `Counter`, possibly at the same time, we need to incorporate mutual exclusion to synchronize access. This is very simple using Go's `sync.Mutex` and its two functions, `Lock` and `Unlock`. This is as easy as adding a `Mutex` field to `Counter`, calling `Lock` before accessing/writing data, and calling `Unlock` when done.

part 3b: Calling and waiting on our Goroutine

To run this code concurrently, you need another method

`Counter.countIPsConcurrently(ips []string, nroutines int)`

The goal of this method is to use the `chunkArray` helper function to break the array into multiple smaller arrays, and make concurrent calls to `countIPs` on these smaller chunks. Wait on the goroutines to finish using a `sync.WaitGroup`.

part 4

Print the number of visits from top visitor and IP of top visitor. This is already done for you.

Getting started with our Go program

The first thing you need to do is set up a project directory for your new project.

```
$ cd class/projects
$ mkdir p1 && cd p1
$ pwd
/home/me/class/projects/p1
```

Next, initialize a new Go module in this project directory. This will create on our system a module named `p1` whose source code lives at `class/projects/p1`

```
$ go mod init class/projects/p1
go: creating new go.mod: module class/projects/p1
```

Next, lets create our source file, which we will call `project.go`. The name of this file is not important.

```
package main;

import "fmt"

func main() {
    fmt.Println("Hello from class/projects/p1!")
}
```

After you make changes to the source code of your project, use the `go install` command to compile our code and save the executable binary to `/go/bin/p1`

```
$ go install class/projects/p1
$ ls ~/go/bin/
p1
$ ~/go/bin/p1
Hello from class/projects/p1!
$
```