Golang questions

1.

Fix the following code to get result sum = 21.

```go
package main

func main() {
    var resultChan chan int
    go func() {
        resultChan <- 1 + 2 + 3
    }()
    go func() {
        resultChan <- 4 + 5 + 6
    }()
    sum := 0
    sum += <-resultChan
    sum += <-resultChan
    println("sum:", sum)
}
```

2.

Fix the following code to get result on stdout

```go
package main

func main() {
    go func() {
        sum := 0
        for i := 0; i < 5; i++ {
            sum += i
        }
        println("sum:", sum)
    }()
}

// result: empty stdout
```

3.

What is the result (updated balance) in the following code? Fix it.

```go
package main
```

```go
type Account struct {
    Name    string
    Balance int
}

func (a Account) AddBalance(amount int) {
    a.Balance = a.Balance + amount
}

func main() {
    a := Account{Name: "Name0", Balance: 1000}
    a.AddBalance(200)
    println("updated balance:", a.Balance)
}
```

4.

What is the result if you run the following code? Fix it.

```go
package main

func main() {
    m := make(map[int]bool)
    done := make(chan bool)
    go func() {
        for i := 0; i < 1000; i++ {
            m[i] = true
        }
        done <- true
    }()
    for k := 3000; k < 4000; k++ {
        m[k] = true
    }
    <-done
    println(len(m))
}
```

5.

Fix the following code to get result sum = 10.

```go
package main

import "sync"

func main() {
    sum := 0
    wg := sync.WaitGroup{}
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func() {
            defer wg.Add(-1)
            sum += i
        }()
    }
    wg.Wait()
    println("sum:", sum)
}
```

6.

What's wrong if you run the following code for a long duration?

```
package main

import (
    "math/rand"
    "time"
)

func callExternal(resultChan chan string) {
    time.Sleep(time.Duration(500+rand.Intn(1000)) * time.Millisecond)
    resultChan <- "ok"
}

func main() {
    for true {
        resultChan := make(chan string)
        go callExternal(resultChan)
        select {
        case result := <-resultChan:
            println("result callExternal:", result)
        case <-time.After(1 * time.Second):
            println("callExternal timed out")
        }
    }
}
```

7.

Will the following code print "the number is positive"? Explain and fix the problem.

```
package main

type CustomizedError struct {
    Code    string
    Message string
}
```

```
func (e CustomizedError) Error() string { return e.Message }

func CheckIsPositive(n int) error {
    var myErr *CustomizedError = nil
    if n <= 0 {
        myErr = &CustomizedError{Code: "400", Message: "the number is
negative"}
    }
    return myErr
}

func main() {
    err := CheckIsPositive(5)
    if err != nil {
        println(err.Error())
        return
    }
    println("the number is positive")
}
```

8.

Will the following code print "cannot Sqrt negative number: -9"?

```
package main

import (
    "fmt"
    "math"
)

type ErrNegativeSqrt float64

func (e ErrNegativeSqrt) Error() string {
    return fmt.Sprintf("cannot Sqrt negative number: %v", e)
}

func Sqrt(x float64) (float64, error) {
    if x < 0 {
        return 0, ErrNegativeSqrt(x)
    }
    return math.Sqrt(x), nil
}

func main() {
    squareRoot, err := Sqrt(-9)
    if err != nil {
        fmt.Println(err)
        return
    }
```

```
        fmt.Println("result:", squareRoot)
    }
```

9.

Difference between goroutines vs operating system threads?

10.

What do you do if your program always uses 100% CPU or consumes more RAM day by day?

Hint

1. Sending or receiving from a nil channel

2. App can exit with running goroutines

3. Value vs pointer variable

4. Concurrent write

5. Loop variable in goroutine

6. Leaking goroutines

7. Check nil interface

8. Calling fmt.Sprintf in method Error or String

9. Difference between goroutines vs operating system threads?

Goroutines are much lighter than OS threads. You can run hundreds of thousands of goroutines. A new goroutine consumes 2KiB of memory but the default stack size for a new thread on Linux x64 is 8 MiBs. Go runtime multiplex executing functions onto a set of OS threads, when a function blocks, the runtime automatically moves other functions on the same OS thread to a different, runnable OS thread so they won't be blocked. Programmers do not need to care about these complexities of thread creation and management. Goroutines not only consume less memory but also setup, teardown and switch faster than OS threads.

- Go FAQ goroutine
- StackOverflow
- Go runtime memory _StackMin

10. What do you do if your program always uses 100% CPU or consumes more RAM day by day?

Short answer: use package net/http/pprof to detect where is the bad code.

Basic pprof commands:

- `pdf`: draw the top N functions that use the most amount of resource to file `profile*.pdf`
- `list {functionName}`: show resource usage for each line of code in a function
- `top{N}`: show the top N functions that use the most amount of resource