```
## 🧪 Project Title: Demonstrating a Race Condition Between Goroutines in Go

### 📄 Explanation

A **race condition** occurs when two or more goroutines access and manipulate the same shared resource concurrently without proper

In Go, race conditions can arise when multiple goroutines read and write to the same variable simultaneously. Since goroutines are

**How the Race Condition Occurs:**

Consider a shared integer variable `counter` initialized to 0. Two goroutines are launched, each incrementing `counter` 1000 times
```

💻 Go Code Demonstrating the Race Condition

```
    "fmt"
    "sync"
)

var counter int

func increment(wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < 1000; i++ {
        counter++
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go increment(&wg)
    go increment(&wg)

    wg.Wait()
    fmt.Println("Final Counter:", counter)
}
```

Expected Output:

```
Final Counter: 2000
```

Possible Output Due to Race Condition:

```
Final Counter: 1837
```

The actual output may vary with each execution, often resulting in a value less than 2000 due to the race condition.

🛠️ Detecting Race Conditions in Go

Go provides a built-in race detector to help identify race conditions. You can use it by running your program with the `-race` flag: (Introducing the Go Race Detector - The Go Programming Language)

```
go run -race main.go
```

If a race condition is present, the race detector will output warnings indicating the conflicting accesses. (Data Race Detector - The Go Programming Language)

✅ Resolving the Race Condition

To fix the race condition, you can use a `sync.Mutex` to ensure that only one goroutine accesses the shared variable at a time: (Race Condition:Goroutines - Medium)

```
import (
    "fmt"
    "sync"
)

var (
    counter int
    mu      sync.Mutex
)

func increment(wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < 1000; i++ {
        mu.Lock()
        counter++
        mu.Unlock()
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)
```

With the mutex in place, the final counter value will consistently be 2000, as the critical section is now protected from concurrent access. (Race Condition:Goroutines - Medium)