Validate Rush00

Created @November 1, 2021 10:55 AM

Introduction

평가 진행 시 다음과 같은 사항들을 지켜주십시오.

- 상호 간 존중하는 마음으로 예의를 지키며 평가를 진행합니다.
- 평가 진행 중 발생한 기능 장애들은 평가자와 함께 문제점을 밝혀내도록 하십시오. 이 과정에서 토론과 논의가 오갈 수 있습니다.
- 동료가 이해한 것과 다르게 이해할 수도 있다는 점을 배려하십시오. 열린 마음으로 평가에 임하는 것을 권장합니다.

Guidelines

반드시 제출 시 이용되었던 Git 레포지토리를 기준으로 평가를 진행해야 합니다.

평가 진행 시 악의가 있는 별칭 등으로 결과를 조작하지는 않았는지 꼼꼼히 확인하십시오. 그리고 평가 진행 시 이용될 수 있는 스크립트에 대해서도 크로스 체크를 통해 별 문제가 없다는 것을 확인하여 당황스러운 상황을 피할 수 있도록 하십시오.

빈 레포지토리, (서브젝트에서 의도되지 않은) 기능 상에 문제가 있는 프로그램, 설명할 수 없는 코드가 포함된 치팅 등의 상황에서는 **FAIL**을 부여합니다. 특히 치팅에 대해선 경위와 사유를 듣고, 다음에도 같은 상황이 벌어지지 않도록 장려해주십시오.

General

- 제출된 레포지토리가 **gofmt**와 **golangci-lint**를 만족하는지 확인하세요.

```
gofmt -l [filename]
golanci-lint run [filename]
```

- 제출된 레포지토리에 go.sum이 없거나 변조되어 모듈이 정상 작동하지 않을 수 있습니다. **go mod tidy**를 이용하여 모듈 정리를 진행 했음에도 문제가 있다면, **FAIL**입니다.
- 모든 기능들은 서브젝트에 명시된 *_test.go 파일을 이용하여 제출 이전에 검증되어 있어야 합니다. 테스트 파일이 잘 작성되어 있는지 확인하십시오. 테스트 케이스가 빈약하다면 평가자가 테스트 케이스를 요구할 수 있으며, 제시된 테스트 케이스가 정상적으로 작동하지 않는다면 **FAIL**입니다.

✅ **YES**

✅ **No**

Validation

1) ex00

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex00
├── go.mod
└── prime
    ├── prime.c
    ├── prime.h
    ├── prime.go
    └── prime_test.go
```

✅ **YES**

✅ **No**

- .c 파일에 //# 표기된 빈 칸을 적절히 채워져야 합니다. num = 1 경우에 0을 리턴해야 합니다.

✅ YES

✅ No

- prime.go 파일에 다음과 같은 코드가 있는지 확인하십시오. .go 파일에서 math.h를 include 하지 않았다면 타당한 이유가 있는지 물어보십시오.

```
// #include <math.h>
// #include "prime.h"
import "C"
```

✅ YES

✅ No

- prime_test.go 파일에서 import "C"를 하여 cgo를 사용할 수 없습니다. 따라서 prime.go 파일에서 적절한 wrapping 함수를 만들어야 합니다. 다음 처럼 prime.go 파일에 cgo를 이용해서 c 파일에 있는 함수를 호출할 수 있는 wrapping 함수가 있는지 확인하십시오. 비슷한 형태이기만 하면 됩니다.
- isPrimeFast 함수와 isPrimeSlow 함수 둘을 wrappring하는 함수가 있는지 확인하십시오.

```
func CallIsPrimeFast(num int) bool {
  if C.isPrimeFast(C.int(num)) == 1 {
    return true
  } else {
    return false
  }
}
```

✅ YES

✅ No

- **go test**로 실행 하십시오. 테스트 케이스가 부족하다고 생각되면 테스트 케이스를 더 요구할 수 있습니다.

```
1 -> 소수x
2 -> 소수
3 -> 소수
5 -> 소수
7 -> 소수
1009 -> 소수
```

```
1011 -> 소수x
2147483647 -> 소수
```

✅ YES

✅ No

- prime/prime_test.go 파일에 Benchmark라는 이름으로 시작하는 함수가 있는지 확인하십시오. 함수가 없다면, testing.Benchmark는 함수 리터럴을 아래와 같이 실행해야 합니다. 둘 다 아니면 Benchmark 테스트를 하지 않았으므로 **Fail** 입니다.

```
testing.Benchmark(func(b *testing.B) {
  for i := 0; i < b.N; i++ {
    CallIsPrimeFast(num)
  }
})
```

- Benchmark라는 이름으로 시작하는 함수가 있다면 **go test -bench=.** 로, 없다면 **go test**로 Benchmark 테스트를 수행하십시오.
- isPrimeFast와 isPrimeSlow 두 함수에 대해 각각 Benchmark를 실행하여 시간을 비교할 수 있게 했는지 확인하십시오.

✅ YES

✅ No

- 제시된 항목은 평가되지 않습니다. 하지만 피평가자는 이에 대해 명확히 답할 수 있어야 합니다.
- isPrimeFast와 isPrimeSlow 두 함수의 실행 시간이 어느 정도 차이 나는 num 값을 어떻게 하면 찾을 수 있을지 답하십시오. (예를 들면 operation 당 100ms 이상 차이가 나기 시작하는 num 값을 찾는다든가...)
- **go test**의 timeout은 기본값이 10분으로 정해져 있습니다. 피평가자가 답변을 했다면, panic: test timed out after 10m0s 이 발생하지 않을지 물어보십시오.
- 위 사항에 대해 답하지 못한다면, 피평가자는 아래 코드를 기반으로 답하여야 합니다.

```
func TestStress(t *testing.T) {
  num := float64(1)
  diff := float64(1)
  wantDiff := float64(100000000)
  for {
    if diff < wantDiff*0.9 {
```

```
      num *= wantDiff / diff
    } else if wantDiff*1.1 < diff {
      num /= diff / wantDiff
    } else {
      break
    }
    if math.IsInf(num, 1) {
      panic("Inf out of range float64")
    }
    resFast := testing.Benchmark(func(b *testing.B) {
      for i := 0; i < b.N; i++ {
        CallIsPrimeFast(int(num))
      }
    })
    resSlow := testing.Benchmark(func(b *testing.B) {
      for i := 0; i < b.N; i++ {
        CallIsPrimeSlow(int(num))
      }
    })
    diff = math.Abs(float64(resSlow.NsPerOp()) - float64(resFast.NsPerOp()))
    fmt.Println("Abs(Fast - Slow):", time.Duration(diff), ", num:", int(num))
  }
```

2) ex01

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex01
├── go.mod
└── gopher
    ├── gohper.go
    └── gopher_test.go
```

✅ **YES** ✅ **No**

- gopher_test.go 파일에 다음과 같은 이름의 함수가 있는지 확인하십시오.
- 피평가자는 Example이란 이름으로 시작하면 어떤 특징이 있는지 답할 수 있어야 합니다.

```
func ExamplePrintGopher()
```

✅ YES ✅ No

- gopher.go의 패키지가 gopher로 되어있고, gopher_test.go의 패키지가 gopher_test로 되어있다면 패키지를 분리한 이유에 대해서 설명할 수 있어야 합니다. 반대로 둘의 패키지가 gopher로만 되어있다면 왜 패키지를 분리하지 않았는지 설명할 수 있어야 합니다. 트레이드 오프 관계지만 타당한 이유가 있어야 합니다.

✅ YES ✅ No

- **go test -v**를 실행했을 때 아래처럼 나오면 안 됩니다.

```
 > go test -v
testing: warning: no tests to run
PASS
ok      ex01/lcs        0.003s
 >
```

- 올바른 결과는 다음과 같습니다.

```
 > go test -v
=== RUN   ExampleGopher
--- PASS: ExampleGopher (0.00s)
PASS
ok      ex01/gopher     0.003s
 >
```

✅ YES ✅ No

3) ex02

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.
- star.h 같은 헤더 파일이 있다면 괜찮습니다. 문제를 제대로 안 읽은 것이 분명하지만, 서브젝트에는 별도의 제한 사항으로 두지 않았기 때문에 문제는 없습니다.
- 별 찍기 문제를 풀기위해 c파일을 따로 만들어도 괜찮습니다. 다만, 파일 이름이 ex02/c/star.c 혹은 ex02/c/Makefile이면 안 됩니다.

```
ex01
├── go.mod
└── star
    ├── star.go
    └── star_test.go
```

✅ **YES** ✅ **No**

- star.go 파일에 다음과 같은 함수가 존재해야 합니다.

```
func Run(given ...string) (output string, err error)
```

- Run 함수 안에서 os.exec로 컴파일된 c 바이너리 파일을 실행하는지 확인하십시오.
- c 바이너리 파일을 실행할 때 다음 처럼 argc, argv로 모든 가변길이 매개변수를 넘겨주는지 확인하십시오. 타당한 이유가 있다면 상관 없습니다.

```
exec.Command("./c/result_star", given...)
```

- "os/exec" 모듈 외에 다른 모듈을 import 했다면 이유를 물어보십시오.

✅ **YES** ✅ **No**

- star_test.go 파일에 다음과 같은 함수들이 있는지 확인하십시오.

```
func TestMain(m *testing.M)
func TestStar(t *testing.T)
func TestInputBehavior(t *testing.T)
```

☑️ **YES**

☑️ **No**

TestMain

- TestMain 함수에 아래처럼 파일이 존재하는지 확인하는 로직이 작성되어 있어야 합니다. 내용이 정확히 일치하지 않아도 됩니다.
- Makefile과 star.c가 존재하는지 확인하는 로직이 있어야 합니다.

```
if _, err := os.Stat("c/Makefile"); os.IsNotExist(err) {
    log.Print("Makefile does not exist in c/")
    log.Fatal(err)
  }
```

- TestMain 함수에 아래처럼 Makefile을 실행하는 로직이 있는지 확인하십시오. 내용이 정확히 일치하지 않아도 됩니다.

```
if err := exec.Command("make", "-C", "c/").Run(); err != nil {
    log.Print("make fail")
    log.Fatal(err)
  }
```

- TestMain 함수에 아래처럼 테스트를 실행하는 m.Run함수가 호출되었는지 확인하십시오. os.Exit을 사용하지 않았다면 그 이유에 대해서 물어보십시오.

```
res := m.Run()
os.Exit(res)
```

- 다음과 같은 에러 메세지를 상황에 맞게 보여줄 수 있는지 확인하십시오.

```
star.c does not exist in c/
Makefile does not exist in c/
make fail
make clean fail
```

```
> go test
2021/11/03 11:39:57 Makefile does not exist in c/
2021/11/03 11:39:57 stat c/Makefile: no such file or directory
exit status 1
FAIL    ex02/star       0.205s
>
```

✅ **YES** ✅ **No**

TestInputBehavior

- 인자가 여러 개인 경우를 테스트 하는지 확인하십시오.
- 인자가 없는 경우를 테스트 하는지 확인하십시오.

✅ **YES** ✅ **No**

TestStar

- TestStar 함수 내부에서 여러 하위 항목 테스트를 실행하는지 확인하십시오.
- 모든 테스트 케이스를 일일이 작성하면 안 됩니다. 코드를 통해 추상화되어 있어야 합니다. (보통 Golang으로 문제를 풀고, 그 코드를 호출하여 c 바이너리 파일의 실행 결과를 비교하게 됩니다.)
- TestStar 함수가 테스트 통과한 60이하의 숫자에 대해서 passed를 출력하는지 확인하십시오.
- 숫자 0과 음수의 테스트 케이스가 포함 되는지 확인하십시오.
- TestStar 함수가 **FAIL**하는 테스트 케이스를 실행했을 때, 이후 나머지 숫자에 대해서 테스트 케이스를 실행하지 않는지 확인하십시오. **FAIL**하는 테스트 케이스에서 멈추지 않고, 계속 로직이 수행되어선 안 됩니다.

✅ **YES** ✅ **No**

4) bonus

- 다음과 같이 Moulinette을 실행해보세요.

```
go build
./bonus -p=42424
```

- 깃허브에 임시 퍼블릭 레포지토리를 만들어 코드를 제출하십시오. 웹 브라우저 URL에 다음과 같이 QueryString으로 깃허브 레포지토리 주소를 입력해야 합니다. 그 결과로 테스트 케이스에 따라 채점 결과가 **PASS** 혹은 **FAIL**이 나와야 합니다.

```
localhost:42424/ex00/prime&LINK=GITHUB_URL
```

Ratings

✅ **OK**

⭐ **Outstanding project**

🗂️ **Empty work**

💾 **Invalid program**

👀 **Cheat**

💣 **Crash**