

Previous output:

Your Go files are not correctly formatted :

```
$ gofmt -d .
diff -u printstr.go.orig printstr.go
--- printstr.go.orig
+++ printstr.go
@@ -3,7 +3,7 @@
import "github.com/01-edu/z01"

func PrintStr(s string) {
-   for _, c := range s {
-       z01.PrintRune(c)
-   }
+   for _, c := range s {
+       z01.PrintRune(c)
+   }
}
```

Looks like your Go code formatting was fixed by `gofumpt` — it replaced spaces with tabs for indentation inside your `for` loop in `PrintStr`. This is standard Go formatting style.

The diff shows:

diff

 Copy  Edit

```
-   for _, c := range s { -   z01.PrintRune(c) -   } + for _, c := range s { +
z01.PrintRune(c) +   }
```

Tabs instead of spaces for indentation is the usual Go style, and `gofumpt` enforces that strictly.

If you want, I can help you automatically fix your formatting with a command or suggest how to set up your editor to format Go code on save. Would that help?



Import piscine not working in main.go

Got it — you're trying to import the `piscine` package in your `main.go` file, but it's not working. Let's troubleshoot a bit:

Common reasons why importing `piscine` fails:

1. Package not installed / module not found

If `piscine` is an external package (like `github.com/01-edu/piscine`), you need to run:

```
bash
```

 Copy  Edit

```
go get github.com/01-edu/piscine
```

or make sure your `go.mod` file has the right dependency.

2. Wrong import path

Check the exact import path in the package you want. It should match the repository or your local module structure. For example:

```
go
```

 Copy  Edit

```
import "github.com/01-edu/piscine"
```

or if it's local, relative to your module root, something like:

```
go
```

 Copy  Edit

```
import "your_module_name/piscine"
```

3. `go.mod` missing or misconfigured

Make sure your project has a `go.mod` file initialized in the root:

```
bash
```

 Copy  Edit

```
go mod init your_module_name
```

Then run `go mod tidy` to add missing dependencies.

4. Local package not in the right folder

If `piscine` is your own package, ensure it is inside your project folder, e.g.:

```
bash
```

 Copy  Edit

```
/your_project
  /piscine
    piscine.go
    main.go
```

And your import would be:

```
go
```

 Copy  Edit

```
import "your_module_name/piscine"
```

Quick checklist:

- What is the exact import line you wrote in `main.go` ?
- Do you have `go.mod` initialized?
- Is `piscine` an external or local package?
- Are you getting an error message? If yes, please share it.

If you want, share your `main.go` import statement and your project structure, I can help you pinpoint the problem!



```
main.go:3:8: package printS/piscine is not in std
(/usr/local/go/src/printS/piscine)
```

Thanks for the error message! It says:

```
bash
```

 Copy  Edit

```
main.go:3:8: package printS/piscine is not in std (/usr/local/go/src/printS
```



This means Go is trying to find the package `prints/piscine` as if it were a standard library package, but it can't locate it.

Why is this happening?

Your import statement in `main.go` probably looks like this:

```
go
```

 Copy  Edit

```
import "printS/piscine"
```

But `printS/piscine` is **not** a standard package and Go doesn't know where to find it.

How to fix it

1. Use a proper module path and structure

- You need to have a **module path** declared in your `go.mod` file at the root of your project. For example:

```
bash
```

 Copy 

```
go mod init printS
```

- Your folder structure should look like this:

```
go
```

 Copy 

```
/your_project_root
```

```
  go.mod
```

```
  main.go
```

```
  /piscine
```

```
    piscine.go
```

- In your `main.go`, import the package using the module name plus the folder:

```
go
```

 Copy 

```
import "printS/piscine"
```

2. Make sure your `go.mod` is set up

- Run this in your project root:

```
bash
```

 Copy  Edit

```
go mod init prints go mod tidy
```

- This tells Go your module's name is `prints` , and it looks for packages inside your project folders.

3. Your `main.go` example:

```
go
```

 Copy  Edit

```
package main import ( "prints/piscine" ) func main() {  
piscine.SomeFunction() // Replace with actual function }
```

Summary

- Run `go mod init prints` in your project root (where `main.go` lives).
- Make sure `piscine` folder is a subfolder of the project root.
- Import as `"prints/piscine"` in your code.

If you want, share your folder structure or your `main.go` file, I can give you a precise fix!



remove a file of gitea through vs code

You want to remove a file from a Gitea repository using VS Code? Here's a straightforward way to do it:

Steps to remove a file from Gitea repo via VS Code

1. Open your local repo folder in VS Code

Make sure you have cloned your Gitea repository locally and opened it in VS Code.

2. Delete the file in VS Code

- In the Explorer panel, right-click the file you want to delete and select **Delete**.
- Or just select the file and press `Delete` key.

3. Stage the deletion with Git

- Go to the **Source Control** panel in VS Code (the Git icon on the left sidebar).
- You will see the deleted file listed under **Changes**.
- Click the `+` icon next to the file or click **Stage All Changes**.

4. Commit your change

- In the **Source Control** panel, type a commit message like `"Remove filename"` in the message box.
- Click the checkmark icon to commit.

5. Push the commit to Gitea

- Click the `"..."` menu in the Source Control panel and select **Push**.
- This sends your commit (file removal) to the remote Gitea repository.

Alternative: Delete file via terminal in VS Code

You can also do it via terminal:

```
bash
```

 Copy  Edit

```
git rm path/to/file git commit -m "Remove file" git push
```

Let me know if you want help with cloning the repo first or anything else!



```
fatal: not removing 'PrintS/' recursively without -r
```

That error happens because `git rm` by default won't remove directories unless you tell it to remove recursively.

If you want to delete the entire `PrintS/` directory and all its contents, you need to add the `-r` flag:

bash

 Copy  Edit

```
git rm -r PrintS/ git commit -m "Remove PrintS directory" git push
```

Explanation:

- `git rm` alone works on files.
- `-r` means recursive — so it removes directories and everything inside them.

If you just want to delete a single file, use:

bash

 Copy  Edit

```
git rm path/to/file.go
```

If you want me to guide you step-by-step for your exact case, just say!



Your Go files are not correctly formatted :

```
$ gofmt -d .
diff -u printstr.go.orig printstr.go
--- printstr.go.orig
+++ printstr.go
@@ -3,7 +3,7 @@
import "github.com/01-edu/z01"

func PrintStr(s string) {
-   for _, r := range s {
-       z01.PrintRune(r)
-   }
+   for _, r := range s {
+       z01.PrintRune(r)
+   }
}
```

This output means `gofmt` detected formatting issues in your `printstr.go` and suggests a fix.

What changed?

Your original code used spaces for indentation:

```
go
```

 Copy  Edit

```
for _, r := range s { z01.PrintRune(r) }
```

`gofmt` replaced those spaces with **tabs**:

```
go
```

 Copy  Edit

```
for _, r := range s { z01.PrintRune(r) }
```

Why?

Go's standard formatting style (enforced by `gofmt` and stricter by `gofumpt`) requires **tabs** for indentation, not spaces.

How to fix automatically?

Just run:

```
bash
```

 Copy  Edit

```
gofumpt -w printstr.go
```

or

```
bash
```

 Copy  Edit

```
gofmt -w printstr.go
```

The `-w` flag writes the formatting changes directly to your file.

If you want, I can also help you set up your editor to format Go code automatically on save. Would you like that?



```
func PrintStr(s string) {  
-   for _, r := range s {  
-       z01.PrintRune(r)  
-   }  
+   for _, r := range s {  
+       z01.PrintRune(r)  
+   }  
}
```

This diff snippet shows the corrected formatting for your `PrintStr` function:

```
go
```

 Copy  Edit

```
func PrintStr(s string) { - for _, r := range s { - z01.PrintRune(r) - }  
+ for _, r := range s { + z01.PrintRune(r) + } }
```

What changed?

- **Indentation switched from spaces to tabs** (Go standard style).
- The closing brace `}` aligned properly with the `func` keyword.
- The body of the `for` loop is properly indented with tabs.

Why is this important?

Go enforces tabs for indentation, and tools like `gofmt` or `gofumpt` auto-fix this for consistency and readability across Go codebases.

Final formatted function:

go

 Copy  Edit

```
func PrintStr(s string) { for _, r := range s { z01.PrintRune(r) } }
```

If you want, I can help you generate a script or commands to format your entire project!



Write a function that counts the runes of a string and that returns that count.

Expected function:

```
func StrLen(s string) int {
```

```
}
```

Here's a simple function in Go that counts the runes (Unicode code points) in a string and returns that count:

go

 Copy  Edit

```
func StrLen(s string) int { count := 0 for range s { count++ } return  
count }
```

Explanation:

- The `for range` loop iterates over each rune in the string `s`.
- Each iteration increments the `count` by 1.
- Finally, it returns the total number of runes.

If you want me to add tests or integrate this with your existing code, just let me know!



Write a function that takes two pointers to an int (*int) and swaps their contents.

Expected function:

```
func Swap(a *int, b *int) {  
  
}
```

Here's a straightforward Go function that swaps the contents of two `*int` pointers:

```
func Swap(a *int, b *int) { temp := *a *a = *b *b = temp }
```

Explanation:

- temp temporarily stores the value pointed to by a .
- Then *a is assigned the value pointed to by b .
- Finally, *b is assigned the original value stored in temp .

This swaps the integers that the pointers a and b refer to.

If you want, I can also provide an example usage!



Write a function that reverses a string.

This function will return the reversed string.

Expected function:

```
func StrRev(s string) string {  
  
}
```

Here's a Go function that reverses a string correctly, handling Unicode runes properly:

```
func StrRev(s string) string { runes := []rune(s) for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 { runes[i], runes[j] = runes[j], runes[i] } return string(runes) }
```

Explanation:

- Converts the string to a `[]rune` slice to handle multi-byte Unicode characters properly.
- Swaps runes from the start and end moving toward the center.
- Returns the reversed string by converting the runes back to a string.

Want me to show you how to test it?



Write an iterative function that returns the factorial of the int passed as parameter.

Errors (non possible values or overflows) will return 0.

Expected function:

```
func IterativeFactorial(nb int) int {  
  
}
```

Previous output:

```
IterativeFactorial(13) == 0 instead of 6227020800  
exit status 1
```

Here's a correct implementation of the `IterativeFactorial` function in Go that handles errors like negative input and overflow:

go

 Copy  Edit

```
func IterativeFactorial(nb int) int { if nb < 0 || nb > 12 { return 0 }  
result := 1 for i := 2; i <= nb; i++ { result *= i } return result }
```

Explanation:

- `nb < 0` : Negative numbers don't have a factorial — return `0` .
- `nb > 12` : `13! = 6227020800` exceeds the maximum value of a 32-bit `int` (which is typically 2,147,483,647), so return `0` to indicate overflow.
- Uses a loop to calculate factorial iteratively.

Note:

If you're using a 64-bit system and want to support higher values safely, you could change the return type to `int64` , but since your original prompt returns `int` , it's safer to cap at `12!` .

Let me know if you want a recursive version or test cases!



```
IterativeFactorial(13) == 0 instead of 6227020800
exit status 1
```

You're seeing:

```
lua
```

 Copy  Edit

```
IterativeFactorial(13) == 0 instead of 6227020800
exit status 1
```

This is because **6227020800 (13!)** exceeds the limit of a 32-bit `int` , which typically maxes out at 2,147,483,647 . So:

- If you're using `int` (default in Go), the result overflows and becomes incorrect.
- If your test expects `6227020800` , you **must use** `int64` (or even `uint64`) to store larger factorials.

✓ Correct version using `int64`:

go

 Copy  Edit

```
func IterativeFactorial(nb int) int64 { if nb < 0 { return 0 } result :=  
int64(1) for i := 2; i <= nb; i++ { result *= int64(i) } return result }
```

Usage example:

go

 Copy  Edit

```
fmt.Println(IterativeFactorial(13)) // Output: 6227020800
```

If you're required to return `int` and not `int64`:

Then the test case `IterativeFactorial(13)` is invalid because the value doesn't fit in a 32-bit `int`, and returning `0` is correct.

Let me know:

- Are you allowed to change the return type to `int64`?
- Or are you working under a fixed constraint (e.g., a school exercise like 01-edu)?



```
IterativeFactorial(-2605572557631491098) == 0  
instead of 0  
exit status 1
```

Thanks — you're pointing out this result:

lua

 Copy  Edit

```
IterativeFactorial(-2605572557631491098) == 0 instead of 0  
exit status 1
```

That looks like a **test passed** (since `0 == 0`), but your program still exited with a non-zero status — which suggests a **runtime error**, not a logic error.

Most likely cause:

You're passing a **very large negative number** (`-2605572557631491098`) into a function that expects a regular `int`. In Go:

- The type `int` is either **32-bit or 64-bit**, depending on the system.
- That number overflows the valid range of an `int`, and Go panics or misbehaves when using it as an `int`.

✓ Fix: Use `int64` for the parameter

Update your function like this:

go

 Copy  Edit

```
func IterativeFactorial(nb int64) int64 { if nb < 0 { return 0 } result
:= int64(1) for i := int64(2); i <= nb; i++ { result *= i } return
result }
```

Now it can safely accept large **negative** values and still return `0`.

Optional: Add a cap to prevent overflow

Factorials grow very fast, so you might want to cap input to prevent overflow even in `int64`. The largest factorial that fits in `int64` is `20! = 2432902008176640000`.

Add this to your check:

go

 Copy  Edit

```
if nb < 0 || nb > 20 { return 0 }
```


Let me know if you're constrained to using `int` only (e.g. by 01-edu's test runner), and I'll give a compatible workaround.



remove int64 from code

Got it — you're asking for a version of `IterativeFactorial` that **does not** **use** `int64` , so it works with plain `int` .

✓ **Here's the correct `int` -only version with overflow check:**

go

 Copy  Edit

```
func IterativeFactorial(nb int) int { if nb < 0 || nb > 12 { return 0 }  
result := 1 for i := 2; i <= nb; i++ { result *= i } return result }
```

Why `nb > 12` ?

Because:

- $13! = 6227020800 \rightarrow$ exceeds the maximum value for a 32-bit `int` (2,147,483,647)
- So we **limit** the function to `nb <= 12` to avoid overflow

If you're working in an environment like 01-edu or using a 32-bit `int` by design, this is the safe and expected behavior.

Let me know if you need a version that *detects* overflow instead of hard-coding the cap!



Write an iterative function that returns the factorial of the `int` passed as parameter.

Errors (non possible values or overflows) will return 0.

Expected function

```
func IterativeFactorial(nb int) int {
```

```
}
```

Usage

Here is a possible program to test your function :

```
package main
```

```
import (  
    "fmt"  
    "piscine"  
)
```

```
func main() {  
    arg := 4  
    fmt.Println(piscine.IterativeFactorial(arg))  
}
```

And its output :

```
$ go run .  
24  
$
```

Here is the full implementation of the `IterativeFactorial` function using `int` **only**, as expected by your usage and output example:

✓ IterativeFactorial **implementation (in piscine package):**

go

 Copy  Edit

```
package piscine func IterativeFactorial(nb int) int { if nb < 0 || nb >  
12 { return 0 } result := 1 for i := 2; i <= nb; i++ { result *= i }  
return result }
```

✓ Example usage (`main.go`):

go

 Copy  Edit

```
package main import ( "fmt" "piscine" ) func main() { arg := 4
fmt.Println(piscine.IterativeFactorial(arg)) // Output: 24 }
```

🔍 Why `nb > 12` returns `0`:

- $13! = 6,227,020,800 \rightarrow$ this overflows a 32-bit `int` (`max = 2,147,483,647`)
- To prevent overflow, the function safely returns `0` when `nb > 12` or `nb < 0`, as specified.

Let me know if you need a recursive version or want to add tests!



Instructions

Write an iterative function that returns the factorial of the `int` passed as parameter.

Errors (non possible values or overflows) will return `0`.

Expected function:

```
func IterativeFactorial(nb int) int {  
  
}
```

Here's the complete and correct implementation of the required function using `int`, with error handling for invalid input and overflow:

✓ **Final** `IterativeFactorial` **implementation:**

go

 Copy  Edit

```
func IterativeFactorial(nb int) int { if nb < 0 || nb > 12 { return 0 }  
result := 1 for i := 2; i <= nb; i++ { result *= i } return result }
```

✓ Why this works:

- **Negative input** (`nb < 0`) → returns `0` (invalid input)
- **Overflow protection** (`nb > 12`) → returns `0`
 - `13!` = 6227020800 exceeds the 32-bit `int` limit (≈ 2.1 billion)
- Uses a simple iterative loop to compute the factorial

✓ Example usage:

go

 Copy  Edit

```
fmt.Println(IterativeFactorial(4)) // Output: 24  
fmt.Println(IterativeFactorial(-1)) // Output: 0  
fmt.Println(IterativeFactorial(13)) // Output: 0
```

Let me know if you need this wrapped in a full program or tested in a specific setup like 01-edu!



this code must have an output of 24 when it runs correctly and an output of 0 when it runs incorrectly

Perfect — the code you've described should:

- ✓ Return `24` when input is `4` (since `4!` = `24`)
- ✓ Return `0` when the input is invalid (like negative or too large — e.g. `-1` , `13`)

✓ Correct IterativeFactorial function:

go

 Copy  Edit

```
func IterativeFactorial(nb int) int { if nb < 0 || nb > 12 { return 0 }  
result := 1 for i := 2; i <= nb; i++ { result *= i } return result }
```

✓ Example test program (main.go):

go

 Copy  Edit

```
package main import ( "fmt" "piscine" ) func main() { arg := 4  
fmt.Println(piscine.IterativeFactorial(arg)) // Output: 24 arg = 13  
fmt.Println(piscine.IterativeFactorial(arg)) // Output: 0 arg = -2  
fmt.Println(piscine.IterativeFactorial(arg)) // Output: 0 }
```

✓ Output:

bash

 Copy  Edit

```
$ go run . 24 0 0
```

Let me know if you need this in a specific format or with user input, or if you're working under a test system like 01-edu that requires exact structure.



the output must not have 2 '0' on new lines

Got it — you want your program to **only output** 24 **when input is valid** and **only a single 0** **when it's invalid, not multiple lines.**

✓ Final version of your test program (main.go):

go

 Copy  Edit

```
package main import ( "fmt" "piscine" ) func main() { arg := 4 // Change
this to test other values like 13 or -2 result :=
piscine.IterativeFactorial(arg) fmt.Println(result) }
```

✓ Function in piscine/iterativefactorial.go:

go

 Copy  Edit

```
package piscine func IterativeFactorial(nb int) int { if nb < 0 || nb >
12 { return 0 } result := 1 for i := 2; i <= nb; i++ { result *= i }
return result }
```

✓ Output examples:

When arg := 4

arduino

 Copy  Edit

```
$ go run .
24
```

When arg := 13 or arg := -1

arduino

 Copy  Edit

```
$ go run .
0
```

✓ **Only one value printed**, no extra 0 s — meets your requirement exactly.

Let me know if you'd like this validated against multiple test cases automatically.

how do i reset a terminal

To reset a terminal, you have a few options depending on your system and terminal type. Here are the most common methods:

✓ 1. Standard way (works in most terminals)

Just type:

```
bash
```

 Copy  Edit

```
reset
```

Then press **Enter**. This clears the screen and resets terminal settings.

✓ 2. Alternative: Clear screen only

If you just want to clear the screen without resetting settings:

```
bash
```

 Copy  Edit

```
clear
```

But this **won't fix broken formatting or escape sequences** — use `reset` for that.

✓ 3. Force reset with Ctrl + key combination

- **Ctrl + L**: Clears the screen (like `clear`)
- **Ctrl + C**: Cancels a running or frozen command
- **Ctrl + D**: Exits the shell (if the shell is stuck or you want to log out)

✓ 4. If `reset` doesn't work (e.g., command not found)

Try:

```
bash
```

 Copy  Edit

```
tput reset
```

Or:

```
bash
```

 Copy  Edit

```
stty sane
```

Let me know what OS or terminal (e.g. macOS Terminal, VS Code terminal, Linux shell) you're using if you want specific instructions.



```
IterativeFactorial(22) == -1250660718674968576
instead of 0
exit status 1
```

You're seeing this:

```
lua
```

 Copy  Edit

```
IterativeFactorial(22) == -1250660718674968576 instead of 0
exit status 1
```

✓ What's happening?

- You're calculating `22!` using `int` in Go.

- But $22! = 1.1240007e+21$, which **far exceeds the range of a 64-bit signed integer** (`int64` max: 9,223,372,036,854,775,807)
- Since you're using a regular `int`, **this causes an integer overflow**, resulting in a garbage value: -1250660718674968576
- Your function should **return** `0` when an overflow is possible — that's what your test expects.

✓ **Solution: Cap `nb` to 20 or below**

The largest factorial that safely fits in a 64-bit signed `int` is $20! = 2432902008176640000$.

If you're using plain `int`, it's safest to cap at `12`, because Go's `int` might be 32-bit depending on the system.

But if your system supports 64-bit `int`, you can go up to 20 safely.

✓ **Updated function (safe for up to 20):**

go

 Copy  Edit

```
func IterativeFactorial(nb int) int { if nb < 0 || nb > 20 { return 0 }  
result := 1 for i := 2; i <= nb; i++ { result *= i } return result }
```

!If you're required to support only 32-bit `int` (like in 01-edu or similar), use this safer version:

go

 Copy  Edit

```
func IterativeFactorial(nb int) int { if nb < 0 || nb > 12 { return 0 }  
result := 1 for i := 2; i <= nb; i++ { result *= i } return result }
```

Let me know what platform or testing system you're targeting (e.g. 01-edu), and I'll tailor it precisely.

```
package piscine

func RecursiveFactorial(nb int) int {
    if nb < 0 || nb > 20 {
        return 0
    } else {
        result := 1
        for n := 2; n <= nb; n++ {
            result *= n
        }
        return result
    }
}
```

Thanks for sharing your code! You're implementing a **factorial function**, but despite the function name `RecursiveFactorial`, your current implementation is **iterative**, not recursive.

🔍 Your current code:

go

 Copy  Edit

```
package piscine func RecursiveFactorial(nb int) int { if nb < 0 || nb >
20 { return 0 } else { result := 1 for n := 2; n <= nb; n++ { result *=
n } return result } }
```

This works correctly for factorial **within the range 0–20**, but it's **not recursive**.

✓ Correct recursive version:

go

 Copy  Edit

```
package piscine func RecursiveFactorial(nb int) int { if nb < 0 || nb >
20 { return 0 } if nb == 0 || nb == 1 { return 1 } return nb *
RecursiveFactorial(nb-1) }
```

□ Explanation:

- The function calls itself (`RecursiveFactorial(nb - 1)`), which is the definition of recursion.
- Base cases:
 - `nb == 0 || nb == 1 → return 1`
- Error handling:
 - `nb < 0 || nb > 20 → return 0` to avoid invalid input or overflow

Let me know if you want both recursive and iterative versions side by side!



remake this code however "for" is forbidden

Perfect — since the keyword `for` **is forbidden**, and you're writing a **recursive** factorial function, here's the corrected version that meets all your constraints:

✓ Final RecursiveFactorial function (no `for`, uses recursion):

go

 Copy  Edit

```
package piscine func RecursiveFactorial(nb int) int { if nb < 0 || nb > 20 { return 0 } if nb == 0 || nb == 1 { return 1 } return nb * RecursiveFactorial(nb-1) }
```

✓ Why this works:

- **No** for loop — uses **pure recursion**.
- Returns 0 for:
 - **Negative values**
 - **Values > 20** (to prevent integer overflow in `int` on 64-bit systems)
- Base case: `nb == 0 || nb == 1 → return 1`
- Recursive case: `nb * RecursiveFactorial(nb - 1)`

Let me know if you're using this in a testing environment like 01-edu and need test cases or validation code!



Write a recursive function that returns the factorial of the `int` passed as parameter.

Errors (non possible values or overflows) will return 0.

`for` is forbidden for this exercise.

Expected function

```
func RecursiveFactorial(nb int) int {  
  
  
}
```

Here is the correct **recursive** implementation of the `RecursiveFactorial` function in Go, following all your instructions:

✓ Expected Function (No for , Handles Errors, Recursive):

go

 Copy  Edit

```
func RecursiveFactorial(nb int) int { if nb < 0 || nb > 20 { return 0 }
if nb == 0 || nb == 1 { return 1 } return nb * RecursiveFactorial(nb-1)
}
```

🔍 Explanation:

- ✓ **No** for **loop** (requirement met)
- ✓ Uses **recursion** to calculate factorial
- ✓ Returns 0 for:
 - Negative numbers (invalid)
 - Numbers greater than 20 (to avoid `int` overflow)
- ✓ Base cases: `0!` and `1!` both return 1
- ✓ Recursive case: `nb * RecursiveFactorial(nb - 1)`

✓ Example usage (for testing):

go

 Copy  Edit

```
package main import ( "fmt" "piscine" ) func main() {
fmt.Println(piscine.RecursiveFactorial(4)) // Output: 24
fmt.Println(piscine.RecursiveFactorial(13)) // Output: 6227020800 is too
large for int → returns 0 fmt.Println(piscine.RecursiveFactorial(-2)) //
Output: 0 }
```

Let me know if you're targeting 32-bit systems (then you should cap at 12 instead of 20).



Write an iterative function that returns the value of `nb` to the power of `power`.

Negative powers will return 0. Overflows do not have to be dealt with.

```
Expected function
func IterativePower(nb int, power int) int {

}
```

Here is the correct implementation of the `IterativePower` function in Go using an **iterative approach**, as requested:

✓ Expected Function: `IterativePower`

go

 Copy  Edit

```
func IterativePower(nb int, power int) int { if power < 0 { return 0 }
result := 1 for i := 0; i < power; i++ { result *= nb } return result }
```

🔍 Explanation:

- **Negative power** → returns `0` (as required)
- **Base case:** anything to the power of `0` is `1`
- Uses a `for` loop to multiply `nb` by itself `power` times
- No need to handle overflow (as per instruction)

✓ Example usage:

go

 Copy  Edit

```
fmt.Println(IterativePower(2, 3)) // Output: 8
fmt.Println(IterativePower(5, 0)) // Output: 1
fmt.Println(IterativePower(2, -2)) // Output: 0
```

Let me know if you want a version that avoids the `for` loop (recursive version) or includes test cases!



Write a recursive function that returns the value of nb to the power of power.

Negative powers will return 0. Overflows do not have to be dealt with.

for is forbidden for this exercise.

Expected function

```
func RecursivePower(nb int, power int) int {  
  
}
```

Here's the recursive implementation of `RecursivePower` following your requirements:

- Negative powers return 0
- No for loops
- Recursive calculation of nb raised to power

go

Copy Edit

```
func RecursivePower(nb int, power int) int { if power < 0 { return 0 }  
if power == 0 { return 1 } return nb * RecursivePower(nb, power-1) }
```

Explanation:

- If `power < 0`, return 0 (error case)
- If `power == 0`, return 1 (base case)
- Otherwise, multiply nb by the result of `RecursivePower(nb, power-1)` (recursive case)

Example usage:

go

 Copy  Edit

```
fmt.Println(RecursivePower(2, 3)) // Output: 8
fmt.Println(RecursivePower(5, 0)) // Output: 1
fmt.Println(RecursivePower(2, -2)) // Output: 0
```

Let me know if you want me to help with test code or edge cases!



Write a recursive function that returns the value at the position index in the fibonacci sequence.

The first value is at index 0.

The sequence starts this way: 0, 1, 1, 2, 3 etc...

A negative index will return -1.

for is forbidden for this exercise.

Expected function

package piscine

```
func Fibonacci(index int) int {  
  
}
```

Here's the recursive Fibonacci function as requested, with no for loops and error handling for negative indices:

go

 Copy  Edit

```
package piscine func Fibonacci(index int) int { if index < 0 { return -1  
} if index == 0 { return 0 } if index == 1 { return 1 } return  
Fibonacci(index-1) + Fibonacci(index-2) }
```

Explanation:

- Returns `-1` if index is negative
- Base cases:
 - `Fibonacci(0) = 0`
 - `Fibonacci(1) = 1`
- Recursive case: sum of previous two Fibonacci numbers

Example usage:

go

 Copy  Edit

```
fmt.Println(Fibonacci(0)) // 0 fmt.Println(Fibonacci(1)) // 1
fmt.Println(Fibonacci(5)) // 5 fmt.Println(Fibonacci(-3)) // -1
```

Let me know if you want it optimized (e.g., memoization) or wrapped in a test program!



Write a function that returns the square root of the int passed as parameter, if that square root is a whole number. Otherwise it returns 0.

Expected function

```
func Sqrt(nb int) int {  
  
  
}
```

Here's a straightforward function that returns the integer square root of `nb` if it's a perfect square, otherwise returns `0` :

go

 Copy  Edit

```
func Sqrt(nb int) int { if nb < 0 { return 0 } for i := 0; i*i <= nb;  
i++ { if i*i == nb { return i } } return 0 }
```

Explanation:

- Returns 0 if nb is negative.
- Iterates from 0 upwards, checking if $i*i$ equals nb .
- If found, returns i .
- If no perfect square matches, returns 0 .

If you want me to write a version without for or using recursion, just ask!