

# Programmier-Einführung mit Go

## Rekursion

Reiner Hüchting

18. Februar 2026

# Rekursion – Überblick

Rekursion

Einleitung

Beispiele

Türme von Hanoi

# Einleitung

Was gibt diese Funktion für  $n = 3$  aus?

```
1 func CountDown(n int) {  
2     if n <= 0 {  
3         return  
4     }  
5     fmt.Println(n)  
6     CountDown(n - 1)  
7 }
```

# Einleitung

Beispiel: Addition als Gleichungen spezifiziert

$$\textcolor{blue}{x} + 0 = x$$

$$\textcolor{blue}{x} + s(\textcolor{orange}{y}) = s(x + y)$$

Anwendung der Gleichungen:

$$\begin{array}{rcl} s(s(0)) + s(\textcolor{orange}{s}(s(0))) & & \\ s( & \textcolor{blue}{s}(s(0)) + \textcolor{orange}{s}(s(0)) & ) \\ s(s( & \textcolor{blue}{s}(s(0)) + \textcolor{orange}{s}(0) & )) \\ s(s(s( & \textcolor{blue}{s}(s(0)) + \textcolor{orange}{0} & ))) \\ s(s(s( & \textcolor{blue}{s}(s(0)) & ))) \end{array}$$

# Einleitung

## Rekursive Addition als Go -Programm:

```
1 func Add1(x, y int) int {  
2  
3     // Gleichungen für die Addition:  
4     // x + 0 = x  
5     // x + (y+1) = (x+y) + 1  
6  
7     if y == 0 {  
8         return x  
9     }  
10    return Add1(x, y-1) + 1  
11 }
```

# Einleitung

## Alternative Version (**Tail-Recursion**):

```
1 func Add2(x, y int) int {
2
3     // Gleichungen für die Addition:
4     // x + 0 = x
5     // x + y = (x+1) + (y-1)
6
7     if y == 0 {
8         return x
9     }
10    return Add2(x+1, y-1)
11 }
```

# Einleitung

## Wozu Rekursion?

- ▶ Manches lässt sich kürzer und eleganter schreiben.
- ▶ Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^n i \quad \text{oder} \quad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n - 1) \end{aligned}$$

- ▶ Als iteratives Go -Programm:

```
1 func FactorialIter(n int) int {
2     result := 1
3     for i := 2; i <= n; i++ {
4         result *= i
5     }
6     return result
7 }
```

# Einleitung

## Wozu Rekursion?

- ▶ Manches lässt sich kürzer und eleganter schreiben.
- ▶ Beispiel Fakultät:

$$fac(n) = \prod_{i=1}^n i \quad \text{oder} \quad \begin{aligned} fac(0) &= 1 \\ fac(n) &= n \cdot fac(n-1) \end{aligned}$$

- ▶ Als rekursives Go -Programm:

```
1 func Factorial(n int) int {
2     if n <= 1 {
3         return 1
4     }
5     return n * Factorial(n-1)
6 }
```

# Einleitung

## Schema für rekursive Definitionen

- ▶ Ein oder mehrere Basisfälle (Rekursionsanfang, Anker).
- ▶ Ein oder mehrere rekursive Aufrufe (Rekursionsschritt).

## Vergleich mit while -Schleifen

- ▶ Abbruchbedingung entspricht Rekursionsanfang.
- ▶ Schleifenrumpf entspricht Rekursionsschritt.

# Beispiele

## Beispiel: Fibonacci-Folge

$$fib(1) = fib(2) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2)$$

die ersten 10 Fibonacci-Zahlen:

$n :$	1	2	3	4	5	6	7	8	9	10
$fib(n) :$	1	1	2	3	5	8	13	21	34	55

# Beispiele

## Beispiel: Hailstone-Folge

- ▶ Beginne mit einer *natürlichen Zahl*  $n$ .
- ▶ Ist  $n$  gerade, so nimm als nächstes  $n/2$ .
- ▶ Ist  $n$  ungerade, so nimm als nächstes  $3n + 1$ .
- ▶ Wiederhole, bis der Zyklus 4, 2, 1 erreicht ist.

## Beispiele:

$n = 1$ : 1, 4, 2, 1

$n = 2$ : 2, 1, 4, 2, 1

$n = 3$ : 3, 10, 5, 16, 8, 4, 2, 1

$n = 4$ : 4, 2, 1

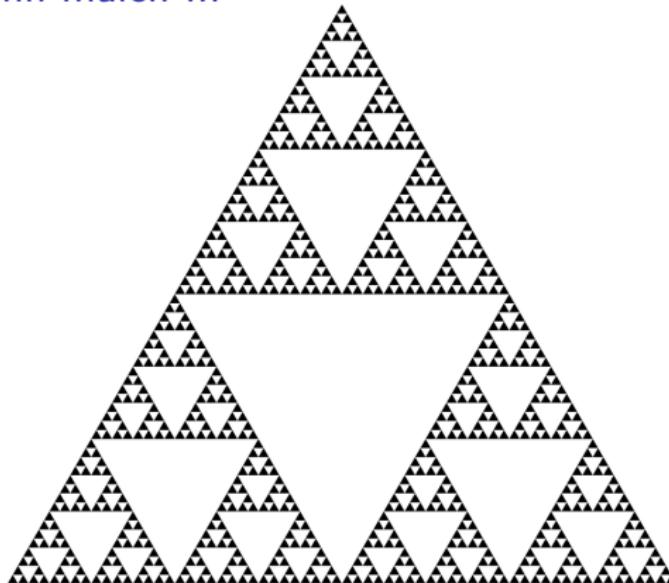
$n = 5$ : 5, 16, 8, 4, 2, 1

$n = 6$ : 6, 3, 10, 5, 16, 8, 4, 2, 1

$n = 7$ : 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

## Beispiele

Rekursion kann malen ...



Dieses Bild wird **Sierpinski-Dreieck** genannt.

# Beispiele

## Beispiel: Ackermann-Funktion

$$A(m, n) = \begin{cases} n + 1 & \text{falls } m = 0 \\ A(m - 1, 1) & \text{falls } m > 0 \text{ und } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{falls } m > 0 \text{ und } n > 0 \end{cases}$$

## Hintergrund

- ▶ Die Werte dieser Funktion wachsen extrem schnell!
- ▶ Die Funktion wurde erdacht, um zu beweisen, dass Schleifen ohne Laufzeitschranke beim Programmieren notwendig sind.
- ▶ Der Beweis hat die Wachstumsgeschwindigkeit der Ackermann-Funktion verwendet.

# Türme von Hanoi

Aufgabe: Bewege einen Turm aus Spielsteinen von A nach C

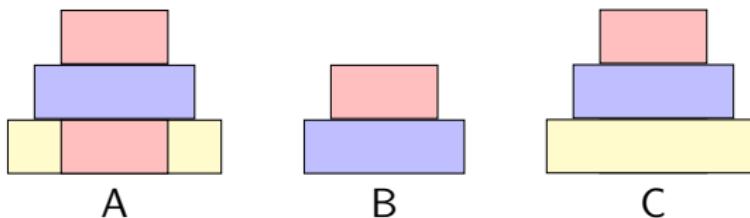
Gegeben:

- ▶ Spielsteine unterschiedlicher Größe.
- ▶ Drei Stellen A, B und C, an denen Spielsteine liegen können.

Spielregeln:

1. Die Steine werden einzeln bewegt.
2. Es wird niemals ein größerer Stein auf einen kleineren gelegt.

Beispiel mit 3 Steinen:



# Türme von Hanoi

Frage: Wie bewegt man einen Turm der Höhe  $h$  von A nach C?

Naive Antwort:

1. Bewege alle bis auf die letzte Platte von A nach B
2. Bewege die letzte Platte von A nach C
3. Bewege den Turm von B nach C

Überraschung: So naiv ist das gar nicht!

- Wir konstruieren einen Algorithmus auf Basis dieser Vorgehensweise.

# Türme von Hanoi

Wir definieren stückweise eine Funktion, die das Problem löst.

- ▶ Bewegen einer einzelnen Platte von *Start* ( s ) über *Mitte* ( m ) nach *Ziel* ( z ):

```
1 func Move(s, z string) {  
2     fmt.Printf("Bewege Scheibe von %s nach %s.\n",  
3 }
```

# Türme von Hanoi

Wir definieren stückweise eine Funktion, die das Problem löst.

- ▶ Bewegen eines Turms der Höhe 1:

```
1 func Hanoi1(s, m, z string) {  
2     Move(s, z)  
3 }
```

# Türme von Hanoi

## Konstruktion der Hanoi-Lösung (Fortsetzung)

- ▶ Bewegen eines Turms der Höhe 2:

```
1 func Hanoi2(s, m, z string) {  
2     Hanoi1(s, z, m)  
3     Move(s, z)  
4     Hanoi1(m, s, z)  
5 }
```

# Türme von Hanoi

## Konstruktion der Hanoi-Lösung (Fortsetzung)

- ▶ Bewegen eines Turms der Höhe 3:

```
1 func Hanoi3(s, m, z string) {  
2     Hanoi2(s, z, m)  
3     Move(s, z)  
4     Hanoi2(m, s, z)  
5 }
```

# Türme von Hanoi

## Konstruktion der Hanoi-Lösung (Fortsetzung)

- ▶ Bewegen eines Turms der Höhe 4:

```
1 func Hanoi4(s, m, z string) {  
2     Hanoi3(s, z, m)  
3     Move(s, z)  
4     Hanoi3(m, s, z)  
5 }
```

Laaaaaaaaaa...

# Türme von Hanoi

## Konstruktion der Hanoi-Lösung (Fortsetzung)

- ▶ Bewegen eines Turms der Höhe 5:

```
1 func Hanoi5(s, m, z string) {  
2     Hanoi4(s, z, m)  
3     Move(s, z)  
4     Hanoi4(m, s, z)  
5 }
```

...aaaaaaang...

# Türme von Hanoi

## Konstruktion der Hanoi-Lösung (Fortsetzung)

- ▶ Bewegen eines Turms der Höhe 6:

```
1 func Hanoi6(s, m, z string) {  
2     Hanoi5(s, z, m)  
3     Move(s, z)  
4     Hanoi5(m, s, z)  
5 }
```

...weeeeilig

# Türme von Hanoi

## Beobachtungen:

- ▶ Die Funktionen `hanoi2` , `hanoi3` , `hanoi4` , ...sind alle gleich.
- ▶ Beim Aufruf wird nur die Zahl reduziert und dann wieder das Gleiche gemacht.
- ▶ Nur bei `hanoi1` wird kein `hanoi0` aufgerufen.

Schlussfolgerung: Wenn die Höhe als Argument übergeben wird, können wir alles in eine Funktion schreiben.

# Türme von Hanoi

## Rekursive Hanoi-Lösung

```
1 func Hanoi(h int, s, m, z string) {
2     if h == 1 {
3         Move(s, z)
4     } else {
5         Hanoi(h-1, s, z, m)
6         Move(s, z)
7         Hanoi(h-1, m, s, z)
8     }
9 }
```

