Produkt aller ungeraden Zahlen

```
func ProductOdd(numbers []int) int {
  result := 1
  for _, i := range numbers {
    if i%2 != 0 {
      result = result * i}}
return result
```

Produkt aller ungeraden Zahlen

```
func ProductOdd(numbers []int) int {
if len(numbers) == 0{
  return 1
  head := numbers[0]
  tail := numbers[1:]
if head%2 != 0 {
return head * ProductOdd(tail)
 }return ProductOdd(tail)}
```

Summe aller ungeraden Zahlen

```
func SumOdd(numbers []int) int {
  if len(numbers) == 0 {
    return 0}
head := numbers[0]
tail := numbers[1:]
if head%2 != 0 {
  return head + SumOdd(tail)}
return SumOdd(tail)}
```

alle Elemente zwischen first und last zurückgeben

```
func IncludeStngBetween(list []string, start, end string) []string
{ firstpos := 0 lastpos := 0
  for pos , s := range list{
    if s == start { firstpos = pos}
    if s == end {
      lastpos = pos}}
  if firstpos >= lastpos {
    return []string{} }
return list[firstpos +1 : lastpos] }
```

Ob gültige Dominokette

```
func ValidDominoChain(dominos
[]Domino) bool {
  for i:= 0; i < len(dominos)-1; i++{
  if dominos[i].right != domi-
nos[i+1].left {
    return false
  }return true}
type Domino struct{
  left int
  right int }
```

func SumEven(numbers []int) int {

```
  if len(numbers) == 0 {
    return 0}
  oddnumbers := numbers[0]
  tail := numbers[1:]
  if oddnumbers%2 == 0 {
    return oddnumbers + SumE-
ven(tail)
  } else {
```

Summe geraderx Zahlen in der Liste

```
func SumEven(numbers []int) int {
  if len(numbers) == 0 {
    return 0}
oddnumbers := numbers[0]
 tail := numbers[1:]
 if oddnumbers%2 == 0 {
return oddnumbers + SumEven(tail)  }
else { return SumEven(tail) }}
```

Schnittmenge/Dupli der beiden Listen zurückgibt

```
func Intersection(a, b []int) []int {
  result := []int{}
    if len(a) == 0 {
      return b}
    if len(a) == 0 {
      return b}
    for _, el1 := range a {
      if Contains(el1,b){
        result = append(result, el1)}}
  return result}
func Contains (e int , l []int) bool {
  for _, el := range l {
    if el == e {
      return true}}
return false}
```

alle Elemente zurückgibt, die mit "abc"

```
func AllAbc(list []string) []string {
result := []string{}
  for _ , val := range list {
    if len(val) >= 3 && val[:3] == "abc" {
      result = append(result, val)}}
  return result}
```

Funktion LongestAbc, die das längste E zurückgibt, das mit "abc" beginnt

```
func LongestAbc(list []string) string {
  longestpos := -1
  longestlen := -1
  for pos , val := range list {
    currentlen := len(val)
    if currentlen >= 3 && val[:3] == "abc" && currentlen > longestlen{
      longestlen = currentlen
      longestpos = pos}}
if longestpos != -1 {
  return list[longestpos]
}  return "" }
```

Längestes E mit ende „def“

```
func LongestDef(list []string) string {
  longestpos := -1
  longestlen := -1
  for pos, val := range list {
    currentlen := len(val)
 if currentlen >= 3 && val[currentlen-3:] == "def" &&
    currentlen > longestlen {
      longestlen = currentlen
      longestpos = pos}  }
  if longestpos != -1 {
    return list[longestpos] }
  return ""}
```

Slice umdrehen

```
func Reverse(slice []int) []int {
  for i, j := 0, len(slice)-1; i < j; i, j = i+1,
j-1 {
 slice[i], slice[j] = slice[j], slice[i] }
 return slice }
```

jeder Position das kleinere der beiden E

```
func MinElements(a, b []int) []int {
if len(a) == 0{
  return b}
if len(b) == 0{
  return a}
  less := a[0]
  if less >= b[0]{
    less = b[0] }
  return append([]int{less}, MinEle-
ments(a[1:], b[1:])...)}
```

Modulo n%m rekursiv*

```
func abc(n, m int) int{
  if n >= m{
    return Foo1(n-m, m)}
  if n < 0 {
   return Foo1(n+m, m) }}
```

Vereinigung der beiden Listen

```
func Union(a, b []int) []int {
  result := []int{}
    if len(a) == 0 {
      return b }
    if len(a) == 0 {
      return b}
    for _, el1 := range a {
      if !Contains(el1, result) {
        result = append(result, el1)}}
    for _, el2 := range b {
      if !Contains(el2, result) {
        result = append(result, el2)}}
  return result}
func Contains(e int , l []int) bool {
  for _, el := range l {
    if el == e {
      return true}}
  return false}
```

Anzahl der Elemente zählt, die mit "abc"

```
func CountAbc(list []string) int {
  result := 0
  for _, val := range list {
if len(val) >= 3 && val[:3] == "abc" {
    result++}}
return result}
```

func ExcludeBetween(list []int, m, n int) []int {

```
result := []int{}
for i := 0; i < len (list); i++ {
 if list[i] > m && list[i] < n{
  result = append(result, list[i])}}
return result}
```

m geteilt n durch rekursiv*

```
func abc(m, n int) int{
  if m < n {
    return 0 }
  return 1+ abc(m-n, n) }
```

jeder Position die Summe der beiden E

```
func SumElements(a, b []int) []int {
  maxlenght := len(a)
  if maxlenght < len(b) {
    maxlenght = len(b)}
  result := make([]int, maxlenght)
  for i := 0; i < maxlenght; i++ {
    if i < len(a) {
      result[i] += a[i]}
    if i < len(b) {
      result[i] += b[i]}}
  return result}
```

m geteilt durch rekursiv*

```
func abc(m, n int) int{
  if m < n {
    return 0 }
```

Elemente außerhalb von first und last zurückgeben

```
func ExcludeStringsOutside(list []string, start string, end
string) []string {
  firstpos := -1 lastpos := -1
  for pos, s := range list{
    if s == start {
      firstpos = pos}
    if s == end {
      lastpos = pos} }
    if firstpos >= lastpos {
      return []string{}}
  return append(list[:firstpos], list[lastpos +1:]...)}
```

func (c Card) GreaterThan(other Card) bool {

```
  return c.Suit==other.Rank &&
c.Rank > other.Rank}

type Card struct {
  Suit int
  Rank int}
```

Potenz n^m rekursiv berechnen*

```
func abc(n, m int) int {
  if m == 0 {
    return 1}
  return n * abc(n, m-1)}
```

Stelle n, Summe der E aus list

```
func ArraySums(list []int) []int {
result := []int{}
 for i := 0; i < len(list); i++ {
      result1 := 0
      for j := 0; j <= i; j++ {
        result1 += list[j]}
 result = append(result, result1)}
 return result}
```

Primzahl ja/nein

```
func IsPrime(n int) bool {
return n > 1 && !DivisibleByAny(n,
n/2)
}
func DivisibleByAny(n, c int) bool {
if c <= 1 {
return false }
if n%c == 0 {
          return true}
```

Gemeinsame Vielfache

```
func CommonMultiples(m, n, max int)
[]int {
  result := []int{}
  for i := 1 ; i <= max; i ++{
    if i%m == 0 && i%n == 0 {
      result = append(result,i)}}
  return result}
```

func ElementProducts(l1, l2 []int) []int {

```
  if len(l1) == 0 {
    return l2 }
    if len(l2) == 0 {
      return l1 }
return append([]int{l1[0] * l2[0]}, ElementProducts(l1[1:],
l2[1:])...)
```

func Power2(x int) float64

```
  if x == 0 {
 return 1}
  if x < 0 {
return Power2(x+1) * 0.5}
return Power2(x-1) *2}
```

```
x1 := Foo1("Hallo", 15)
x2 := Foo2(x1)
x3 := Foo3(127, x2)
if x2 {
  x3 = append(x3, Foo1("Welt", x1))
}
x1 += Foo4(x2, true)
Foo5(x2 && x1 != Foo4(x2, x2))
return x2 && !(x1 > x3[0])
```

x_3 : int

Foo1 (string, int) int

Foo2 (int) bool

Foo3 (int, bool) [] int {} | wegen x_3 append

Foo4 (bool, bool) int | weil Typen beim rechnen gleich sein müssen

Foo5 (bool)

return: bool | wenn if oder && = bool

```
x := 42
var int y 55
int z = 42
s := string([]byte{'a', 'b', 'c'})
b := []byte{'a', 'b', 'c'}
var l1 []int := make([]int, 0)
string := hallo
```

- Zeile 2: Typ und Name der Variablen sind vertauscht und es fehlt ein Gleichheitszeichen.
- Zeile 3: Typ und Name der Variablen sind vertauscht und es fehlt das var oder statt der Typangabe ein :=
- Zeile 6: Die var-Form kann nicht mit der :=-Form kombiniert werden.
- Zeile 7: Bei `hallo` fehlen Anführungszeichen. (*Anmerkung:* Das

```
func IsPrimeCorrect(n int) bool {
    //* Fehler 1: Spezialfall hat gefehlt, 0 und 1
       sind keine Primzahlen
    if n <= 1 {
        return false
    }
    // Im
    for i := 2; i < n-1; i++ {
        if n%i == 0 {
            return false
        }
        //* Fehler 2: Der else-Zweig war falsch: Wenn
           n%i != 0,
        //* dann ist n nicht automatisch prim, denn
           es kann immer
        //*  noch ein anderer Teiler gefunden werden.
    }
    return true
}
```

Behauptung	wahr	falsch
`int` ist ein Datentyp in Go.	X	
`float` ist ein Datentyp in Go.		X
Jede Go-Quelldatei muss eine `main`-Funktion enthalten.		X
Bei Variablen muss der Datentyp immer explizit angegeben werden.		X
`var x1 int`	X	
`var x2 = 42`	X	
`var x3 string = "42"`	X	
`int x4`		X
`x5 int := 42`		X

`int16` ist ein Datentyp in Go.	X	
`float64` ist ein Datentyp in Go.	X	
Eine Funktionssignatur sagt nichts über den Rückgabetyp der Funktion aus.		X
Bei der Definition von Variablen muss der Datentyp immer feststehen.	X	
`range`-Schleifen haben keinen Schleifenzähler.		X

```
x1 := Foo1(1, 2)
x2 := Foo2(x1, "Hallo")
if Foo3(x2) {
  Foo4(x1 < 42)
}
return Foo5(Foo2(x1, fmt.Sprintf("%t", x2)) && x2)
```

- Foo1: `func (int, int) int`
- Foo2: `func (int, string) bool`
- Foo3: `func (bool) bool`
- Foo4: `func (bool)`
- Foo5: `func (bool) bool`

```
func Foo(n int) bool {
  for i := 1; i < n; i++ {
    for j := 1; j < n; j++ {
      if i*1+j*j == n {
        return true
      }
    }
  }
  return false
}
```

Lösung

Die Funktion prüft, ob `n` die Summe zweier Quadrate ist. Falls ja, liefert sie `true`, ansonsten `false`.

Behauptung	wahr	falsch
`void` ist ein Datentyp in Go.		X
Bei der Definition von Variablen muss der Datentyp immer angegeben werden.		X
Eine Funktionssignatur enthält immer den Namen der Funktion.		X
Eigene Datentypen werden immer mit `type` definiert.	X	
Jede Schleife muss einen Zähler haben.		X

```
package fehlersuche1

import "fmt"

func Foo(x int) string {
  return fmt.Sprint(x) // Muss string zurückgeben
}

func Bar(x, y int) string {
  for i := 0; i < x; i++ {
    y += i
  }
  return Foo(y) + fmt.Sprint(x) // Zwei Werte zurückgegeben.
}

func FooBar() { // Eckige statt runder Klammern.
  s := "Huhu"
  for x := range []int{10, 20, 30, 40, 50} {
    s += Bar(x, x+len(s)) // x und s können nicht addiert werden.
  }
  s += "y"
  fmt.Println(s) // println klein geschrieben.
}
```

```
func Foo(s1, s2 string) int {
  if len(s1) == 0 || len(s2) < len(s1) {
    return 0
  }
  if s2 == s1+s2[len(s1):] {
    return 1 + Foo(s1, s2[1:])
  }
  return Foo(s1, s2[1:])

}
```

Die Funktion berechnet, wie oft `s1` in `s2` vorkommt.

Beispielrechnungen:

$$\begin{aligned}\texttt{Foo("ab", "abab")} &= 1 + \texttt{Foo("ab", "bab")}\\ &= 1 + \texttt{Foo("ab", "ab")}\\ &= 1 + 1 + \texttt{Foo("ab", "b")}\\ &= 1 + 1 + \texttt{Foo("ab", "")}\\ &= 1 + 1 + 0\\ &= 2\end{aligned}$$

```
// Sorted liefert true, falls die Liste aufsteigend sortiert ist.
func Sorted(list []int) bool {
  // Sonderfall für leere Liste hat gefehlt:
  if len(list) == 0 {
    return true
  }
  // i ist der Index, list[i] der Wert.
  // Es muss auf beiden Seiten der Ungleichung list[...] stehen.
  for i := range list[1:] {
    // Bei der Vergleichs-Richtung in der Fehler-Version
    // wird geprüft, ob die Liste ab- und nicht aufsteigend
    // sortiert ist.
    if list[i] < list[i+1] {
      return false
    }
```

```
func Foo() {
    x := 5                  // Output:
    y := 38                 // 5
    y = Bar(x) + y          // 0X1X2X3X4X5
    fmt.Println(x)          // 85
    fmt.Println(y)
}

func Bar(x int) int {
    fmt.Println(x)
    for i := 0; i < x; i++ {
        fmt.Printf("%vX", i)
    }
    x += 42
    return x
}
```

```
func Sorted(list []int) bool {
  for i, el := range list[1:] {
    if el < list[el] {
      return false
    }
  }
  return true
}
```

+ Sonderfall für leere Liste fehlt