

# Lösung Aufgabe 3 „Tobis Turnier“

## Lösungsidee

Die drei Turniervarianten werden nach gegebener Spezifikation implementiert und zahlreiche Durchläufe simuliert.

Entsprechend ist die eigentliche Arbeit hauptsächlich **Umsetzung** der Spezifikation.

## Genauigkeit

Abhängig von der gleichmäßigen Streuung der Zufallszahlen des verwendeten Zufallsgenerators und der Anzahl der simulierten Durchläufe.

## Komplexität

Sei  $n$  die Anzahl der Spieler. Dann benötigen die Turniervarianten jeweils folgend viele Spiele:

- Liga:  $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
- K.O.:  $\sum_{i=1}^{\log_2(n)} \frac{n}{2^i} = n$
- K.O.x5: 5-mal so viele Spiele wie K.O.:  $5 \cdot n$

Offensichtlich ist Liga am aufwendigsten mit quadratisch vielen Spielen ( $S(n) = O(n^2)$ ). K.O. und K.O.x5 dahingegen benötigen nur linear viele ( $S(n) = O(n)$ ).

## Umsetzung

Implementierung in der modernen und performanten Programmiersprache Go.

## Kompilieren

`go build` (erzeugt `a5-Wichteln`) oder `go build main.go` (erzeugt `main`)

## Verwendung

`go run main.go <pfad>` oder `./main <pfad>`

Beispiel: `./main beispieldaten/spielstaerken1.txt`

## Ausgabe

```
Liga: <Siege Bester Spieler bei 1.000.000 Liga-Turnieren in Prozent> %  
K.O.: <Siege Bester Spieler bei 1.000.000 K.O.-Turnieren in Prozent> %  
K.O.x5: <Siege Bester Spieler bei 1.000.000 K.O.x5-Turnieren in Prozent> %  
Zeit verstrichen: <Verstrichene Zeit in Sekunden> s
```

Das Programm lässt sich nach dem EVA-Prinzip in Eingabe, Verarbeitung und Ausgabe gliedern:

## Bibliotheken

- Eingabe:
  - os: Programmargumente zum Erhalten des Pfades
  - io/ioutil: Hilfsbibliothek („utility“) zum Einlesen der Datei mit den Spielstärken.
  - strings: Unterteilen der Datei in Zeilen („split“)
  - strconv: Umwandlung von Strings in Zahlen
- Verarbeitung:
  - time: „Seeden“ des ansonsten determinierten Zufallsgenerators von Go, Zeitmessung
  - math/rand: Zufallsgenerator
- Ausgabe: fmt: Formattierung. Nötig für Ausgabe von Zahlen (implementiert Funktionalität u.a. wie C's printf)

## Eingabe

Das erste Argument wird als Dateipfad verstanden. Die Datei wird eingelesen und in Zeilen unterteilt. Die Spielstärken werden mit einer Schleife zeilenweise zu Zahlen konvertiert und in einer natürliche-Zahlen-„Slice“ fester Länge gespeichert.

## Verarbeitung

Zuerst wird ein „sanity-check“ mit den Beispieldaten durchgeführt: Es darf nur einen besten Spieler geben, ansonsten bricht das Programm aufgrund fehlerhafter Eingaben ab.

Zentral ist zunächst eine Funktion, die den Gewinner eines einzigen Spiels bestimmt. Diese ist nach Spezifikation:

- Erster Spieler gewinnt, wenn seine Kugel gezogen wird
  - Die gezogene Kugel ist eine Zufallszahl von 1 bis zu den addierten Spielstärken
  - Diese ist eine Kugel des ersten Spielers, wenn sie  $\leq$  der Spielstärke des ersten Spielers ist
    - \* Anschaulich: Von den nummerierten Kugeln gehören die ersten Spielstärke-viele Kugeln dem ersten Spieler, alle „danach“ dem Zweiten
- Sonst gewinnt der zweite Spieler

## Liga

In der Implementation von Liga muss nach Spezifikation jeder Spieler gegen jeden anderen einmal spielen. Entsprechend geht man alle Spieler durch. Für jeden Spieler iteriert man dann über alle Anderen mit einer höheren Spielernummer als Kontrahenten (zwei geschachtelte Schleifen). An einem einfachen Beispiel mit drei Spielern 1, 2, 3 wird sofort klar, wieso dies funktioniert:

1. Betrachte Spieler 1
2. Spiele gegen Spieler 2
3. Spiele gegen Spieler 3
4. Betrachte Spieler 2
5. Gegen Spieler 1 muss nicht mehr gespielt werden
6. Spiele Gegen Spieler 3
7. Betrachte Spieler 3

8. Gegen keinen Spieler muss noch gespielt werden

Für die Siege der Spieler wird eine Slice angelegt mit [Spielernummer] = Siege. Nach Simulieren aller Spiele wird der Spieler mit den meisten Siegen bestimmt (einfache Maximumsuche), wobei nach Spezifikation bei Gleichstand der Spieler mit der kleineren Spielernummer gewinnt.

## **K.O.**

Wir implementieren K.O. rekursiv:

### **Rekursionsanfang**

**Sieger des Turniers** ist derjenige, der siegt, wenn wir den kompletten Turnierplan als Ausschnitt wählen.

### **Rekursiver Aufruf**

Wir betrachten einen Ausschnitt / Teil des Turnierplans. Sieger des Ausschnittes ist derjenige Spieler, der im Spiel zwischen dem Sieger der linken Hälfte des Ausschnitts und dem Sieger der rechten Hälfte des Ausschnitts siegt.

### **Rekursionsende**

Umfasst der betrachtete Ausschnitt nur zwei Spieler, ist der Sieger derjenige, der im Spiel der beiden siegt.

## **K.O.x5**

Wir nutzen die K.O.-Implementation, ersetzen aber die Funktion, die entscheidet, wer ein Spiel gewinnt:

Anstatt eines einzigen Spiels werden bis zu fünf Spiele simuliert. Hat der erste Spieler drei gewonnen, gewinnt er und es wird abgebrochen. Kommt dies nicht vor, gewinnt sein Kontrahent.

## **Simulation**

Schließlich werden alle Turniere mit einer einfachen Schleife eine Million Male gespielt. Dabei wird eine Zählvariable erhöht, wenn der beste Spieler gewinnt.

## **Ausgabe**

Mit `fmt` wird nach jeder Simulation eine Zeile im Format `<Turniervariante>: <Siege erster Spieler in Prozent> %` ausgegeben.

Am Ende der Programmausführung steht die Ausgabe der verstrichenen Zeit.

## Quellcode

### main.go

```
1 package main
2
3 import (
4     "fmt"
5     "io/ioutil"
6     "math/rand"
7     "os"
8     "strconv"
9     "strings"
10    "time"
11 )
12
13 func main() {
14     // Zeitmessung
15     nanos := time.Now().UnixNano()
16     // Eingabe
17     if len(os.Args) != 2 {
18         println("Verwendung: <pfad>")
19         return
20     }
21     // Einlesen
22     text, err := ioutil.ReadFile(os.Args[1])
23     if err != nil {
24         panic(err)
25     }
26     lines := strings.Split(string(text), "\n")
27     anzahl, err := strconv.Atoi(lines[0])
28     if err != nil {
29         panic(err)
30     }
31     // Spielstärken, bester Spieler
32     spielstaerken := make([]int, anzahl)
33     besterSpieler := 0
34     for index := range spielstaerken {
35         // Atoi = Text zu Zahl
36         spielstaerke, err := strconv.Atoi(lines[index+1])
37         if err != nil {
38             panic(err)
39         }
40         spielstaerken[index] = spielstaerke
41         if spielstaerke > spielstaerken[besterSpieler] {
```

```

42         besterSpieler = index
43     }
44 }
45 for index, spielstaerke := range spielstaerken {
46     if index != besterSpieler && spielstaerke ==
47         ↪ spielstaerken[besterSpieler] {
48         // "Sanity-check": Es darf nur einen besten Spieler
49         ↪ geben
50         panic("Mehrere beste Spieler!")
51     }
52 }
53 // Random "seeden" - ansonsten ist Go-Random determiniert
54 rand.Seed(time.Now().UnixNano())
55
56 // Gibt zurück, ob Spieler 1 gewonnen hat
57 spieler1Gewinnt := func(spieler1, spieler2 int) bool {
58     if rand.Intn(spielstaerken[spieler1]+spielstaerken[spieler2])
59     ↪ < spielstaerken[spieler1] {
60         return true
61     }
62     return false
63 }
64
65 // Gewinner eines Spiels, Gibt Spielernummer zurück
66 gewinner := func(spieler1, spieler2 int) int {
67     if spieler1Gewinnt(spieler1, spieler2) {
68         return spieler1
69     }
70     return spieler2
71 }
72
73 // Eine Runde Liga: Gibt 1 zurück, wenn der beste Spieler gewonnen hat,
74 ↪ sonst 0
75 liga := func() int {
76     // Slice der Siege
77     siege := make([]int, anzahl)
78     // Jeder gegen jeden
79     for spieler1 := range spielstaerken {
80         for spieler2 := spieler1 + 1; spieler2 <
81             ↪ len(spielstaerken); spieler2++ {
82             // Gewinner erhält den Sieg
83             siege[gewinner(spieler1, spieler2)]++
84         }
85     }
86 }

```

```

81 // Sieger ermitteln: Spieler von kleiner zu größer
   ↳ Spielernummer durchgehen
82 meisteSiege := 0
83 for spieler, anzahlSiege := range siege {
84     // Nur bei mehr Siegen neuer Sieger: Bei gleich vielen
   ↳ bleibt es der mit der kleineren Spielernummer
85     if siege[meisteSiege] < anzahlSiege {
86         meisteSiege = spieler
87     }
88 }
89 if meisteSiege == besterSpieler {
90     // Bester Spieler hat gesiegt
91     return 1
92 }
93 return 0
94 }
95
96 // Gibt eine Funktion zurück, die eine Runde K.O. simuliert
97 koVariante := func(gewinner func(int, int) int) func() int {
98     return func() int {
99         // Turnierplan erstellen
100         turnierplan := make([]int, anzahl)
101         for index := range turnierplan {
102             turnierplan[index] = index
103         }
104         // Mischen (verwendet Fisher-Yates)
105         rand.Shuffle(len(turnierplan), func(i, j int) {
106             turnierplan[i], turnierplan[j] =
↳ turnierplan[j], turnierplan[i]
107         })
108         // Rekursiv Sieger eines "Bereiches" des Turnierplans
   ↳ ermitteln
109         var sieger func(int, int) int
110         sieger = func(start, ende int) int {
111             diff := ende - start
112             if diff == 1 {
113                 // Linke & rechte Hälfte umfassen nur
   ↳ einen Spieler: Gegeneinander
   ↳ antreten lassen!
114                 return gewinner(turnierplan[start],
   ↳ turnierplan[ende])
115             }
116             mitte := start + diff/2
117             // Es spielt der Gewinner der linken Hälfte
   ↳ gegen den der rechten Hälfte

```

```

118         return gewinner(sieger(start, mitte),
119             ↪ sieger(mitte, ende))
120     }
121     if sieger(0, anzahl-1) == besterSpieler {
122         // 1 zurückgeben, wenn der beste Spieler
123         ↪ gewonnen hat
124         return 1
125     }
126     // Sonst 0
127     return 0
128 }
129 // Einfache K.O.-Variante: Ein Spiel entscheidet
130 ko := koVariante(gewinner)
131
132 // K.O. x5: "Best of 5"
133 ko5 := koVariante(func(spieler1, spieler2 int) int {
134     // Siege des ersten Spielers
135     siegeSpieler1 := 0
136     for i := 0; i < 5; i++ {
137         if spieler1Gewinnt(spieler1, spieler2) {
138             siegeSpieler1++
139             if siegeSpieler1 == 3 {
140                 // 3. Sieg, Spieler 1 hat gewonnen!
141                 return spieler1
142             }
143         }
144     }
145     return spieler2
146 })
147
148 // Tester für Turniervariante: Lässt viele Simulationen laufen
149 anzahlLaeufe := int(1e6)
150 testeTurniervariante := func(name string, runde func() int) {
151     siegeBesterSpieler := 0
152     for laeufe := 0; laeufe < anzahlLaeufe; laeufe++ {
153         // Spiele eine Runde!
154         siegeBesterSpieler += runde()
155     }
156     // Ausgeben
157     fmt.Println(name+":",
158 ↪ (float32(siegeBesterSpieler)/float32(anzahlLaeufe))*100.0, "%")
159 }

```

```

160         // Varianten testen
161         testeTurniervariante("Liga", liga)
162         testeTurniervariante("K.O.", ko)
163         testeTurniervariante("K.O.x5", ko5)
164
165         fmt.Println("Zeit verstrichen:",
↪      float32(time.Now().UnixNano()-nanos)/1e9, "s")
166     }

```

## Beispiele

### spielstaerken1.txt

Liga: 34.6763 %  
 K.O.: 43.722702 %  
 K.O.x5: 64.1482 %  
 Zeit verstrichen: 5.116692 s

### spielstaerken2.txt

Liga: 20.9622 %  
 K.O.: 31.089602 %  
 K.O.x5: 37.2073 %  
 Zeit verstrichen: 4.753782 s

### spielstaerken3.txt

Liga: 31.5096 %  
 K.O.: 18.2694 %  
 K.O.x5: 31.031502 %  
 Zeit verstrichen: 13.191793 s

### spielstaerken4.txt

Liga: 11.4277 %  
 K.O.: 7.0034 %  
 K.O.x5: 7.7537003 %  
 Zeit verstrichen: 12.942738 s

### spielstaerken5.txt

Zusätzliches Beispiel:

16  
 100  
 5  
 5



5  
5  
5  
5  
5  
5  
5  
5  
5  
5  
5  
5  
5

Liga: 99.786 %

K.O.: 87.4557 %

K.O.x5: 99.7381 %

Zeit verstrichen: 12.901049 s

### Fazit

- K.O.x5 weist immer bessere Siegesquoten des besten Spielers auf als einfaches K.O., da die Wahrscheinlichkeit, dass der schlechtere Spieler in einem Aufeinandertreffen gewinnt, weiter gesenkt wird, indem dieser den Großteil Spiele gewinnen müsste. Insgesamt ist K.O.x5 genauer als K.O.
- Durchschnittlich erweist sich K.O.x5 auch im Vergleich zur Liga als zuverlässiger (35 % vs. 64 %, 21 % vs. 37 %, 31.5 % vs. 31 %, 11.5 % vs. 8 %, 99.8 % vs. 99.7 %).
- Im Falle vieler ähnlich starker Kontrahenten (spielstaerken4.txt) hat der beste Spieler es schwieriger, sich im K.O.x5 zu behaupten, da der Verlust eines einzigen Aufeinandertreffens reicht, damit er verliert. Hier ist Liga etwas zuverlässiger (11.5 % vs. 8 %).
- Insgesamt empfehle ich Tobi entsprechend **K.O.x5**