

## Lösung Aufgabe 2 „Dreieckspuzzle“

### Lösungsidee

Wir nutzen eine **Brute-Force**, um systematisch mögliche Lösungen des Puzzles auszuprobieren, wie es auch ein Mensch tun würde.

Hierfür unterteilen wir ein gelöstes Puzzle zunächst in „Eckteile“ und „Kernteile“:

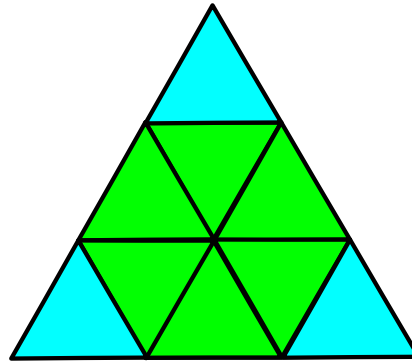


Abbildung 1: Eckteile (türkis) und Kernteile (grün)

Den Kern wiederum unterteilen wir in „Randfiguren“ (äußere Figuren des Kerns) und „Kernfiguren“ (innere Figuren des Kerns).

Zunächst probieren wir alle möglichen Kerne aus:

1. Wir beginnen mit einem Teil und einer der drei Seiten des Teils
2. Wir suchen ein *anderes*, in der aktuellen Anordnung noch nicht verwendetes Teil, das mindestens eine zur gewählten Seite passende Seite besitzt
3. Wir bestimmen die gegenüberliegende Seite des passenden Teils, und suchen nun dafür wieder passende Teile (zurück zu Schritt 1)

Dies wiederholen wir so lange, bis wir einen Kern bestehend aus 6 Kernteilen bestimmt haben. Dann muss noch die letzte Seite des letzten Teils zur ersten Seite des ersten Teils passen, und der Kern ist geschlossen!

Ein geschlossener Kern reicht allerdings noch nicht unbedingt. Die drei übrigen Teile müssen noch als Eckteile zu den Randfiguren des Kerns passen. Dies probieren wir mit einer kleinen Brute-Force aus, die alle möglichen Arten, die übrigen Teile als Ecken anzuordnen durchgeht. Findet diese eine funktionierende Anordnung, haben wir eine Lösung für das Puzzle!

### Korrektheit

Abgesehen von Implementierungsfehlern könnte die Brute-Force nur falsche Resultate liefern, wenn einige Lösungsmöglichkeiten nicht betrachtet/probiert würden. Dies ist hier aber nicht der Fall.

### Komplexität

Wir überlegen uns eine obere Grenze für die durchzugehenden Anordnungen der Puzzlteile.

Für das erste Kernteil bestehen 9 Wahlmöglichkeiten, für das zweite nur noch 8, usw.

Es gibt also maximal  $\frac{9!}{3!} = 60.480$  Möglichkeiten, die Kernteile und ihre Reihenfolge auszuwählen.

Jedes Kernteil kann nun maximal zwei passende Seiten besitzen, also gibt es für jedes Kernteil nochmal zwei Drehmöglichkeiten.

Somit kommt man auf maximal  $\frac{9!}{3!} \cdot 2^6 = 60.480 \cdot 64 = 3.870.720$  mögliche Kernanordnungen.

Für jede dieser Kernanordnungen müssen noch Ecken probiert werden. Hierbei gibt es zwei Möglichkeiten, den Kern zu drehen, und für jede  $3! = 6$  infragekommende Eckanordnungen.

Tatsächlich brauchen wir als „Startteil“ des Kerns nur 4 Teile auszuprobieren (unter 4 Teilen muss ein Kernteil sein), was diese Anzahl wiederum halbiert. Andererseits müssen wir für diese jeweils alle 3 Seiten probieren, was wiederum zu einer Verdreifachung führt.

Schließlich erhält man  $\frac{9!}{3!} \cdot 2^6 \cdot 2 \cdot 3! \div 2 \cdot 3 = 9! \cdot 2^6 \cdot 3 = 69.672.960$  maximal auszuprobierende Lösungen.

Selbst wenn dieser theoretische worst-case - insofern er überhaupt trotz der vielen Abbruchkriterien möglich ist - einträte, würde das Programm schätzungsweise noch in sinnvoller Zeit terminieren (siehe verstrichene Zeiten).

## 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/puzzle0.txt`

## Ausgabe

Figuren werden in den jeweiligen Buchstaben im Alphabet umgewandelt (1: A, 2: B usw.). Die unteren Figurenteile (negative Zahlen) werden dabei mit Kleinbuchstaben repräsentiert (-1: a, -2: b usw.).

Teile:

<Teile als ASCII-Art>

Lösung:

<Gelöstes Puzzle als ASCII-Art>

Zeit verstrichen: <Verstrichene Zeit in Millisekunden> ms

oder

Puzzle unlösbar

Zeit verstrichen: <Verstrichene Zeit in Millisekunden> ms

### **Bibliotheken**

- `fmt`: Ausgabe & Formattierung
- `io/ioutil`: Einlesen der Datei
- `os`: Programmargumente
- `strconv`: String/Integer-Konversion
- `strings`: Auftrennen von Text
- `time`: Zeitmessung

### **Typen**

#### **Figur**

Alias für `int8` für bessere Lesbarkeit.

#### **Teil**

Puzzleteil: 3-Figuren-Array.

#### **Kernteil**

Kernteil: Seite, and die das nächste Kernteil „anzudocken“ hat, Teil-ID, und Zeiger auf vorangehendes Kernteil (Knoten einer Single-Linked-List)

#### **Eckteil**

Eckteil: Seite, mit der es an zur Randfigur passt, und Teil-ID.

#### **VerwendeteTeile**

Bitflag für verwendete Teile: Alias für `uint16`

#### **Verwendet(`teil uint8`) bool**

Prüft, ob das Teil mit der jeweiligen Nummer schon verwendet ist.

#### **Verwende(`teil uint8`) VerwendeteTeile**

Gibt neuen Flag zurück mit verwendetem Teil

### **Ablauf**

Wir implementieren die Brute-Force über zwei geschachtelte rekursiven Funktionen: **probiereKerne** und **probiereEcken**. Die Erste probiert wie beschrieben mögliche Kerne. Ist ein Kern komplett, wird mit der Zweiten eine passende Anordnung der Ecken gesucht - entsprechend die Schachtelung.

Die Argumente von `probiereKerne` sind wie folgt:

- Die zur „Startfigur“ (Figur gewählte Seite erstes Teil) passende Figur
- Einen Zeiger auf das vorangehende Kernteil
- `VerwendeteTeile`
- Die Anzahl bisher aneindergehangener Kernteile

In jedem Schritt werden dann alle neun Teile durchgegangen. Für die noch nicht Verwendeten wird geprüft, ob sie an das letzte Kernteil passen; wenn ja, wird mit einer Variante des Kerns mit angehängenem Teil weiterprobiert. Hierbei wird die Anzahl aneindergehangener Teile natürlich um eins erhöht, das Teil als verwendet markiert, und ein neues Kernteil erzeugt, welches als Vorgänger den Zeiger auf das aktuelle Kernteil erhält.

Gestartet wird die Rekursion für jedes der ersten 4 Teile und für jede der drei Seiten. Sie terminiert, sobald ein kompletter Kern mit 6 Teilen (nach Zählvariable Anzahl) und passendem letzten Teil (Figur = `passendZurStartfigur`) gefunden ist. Dann wird die zweite Rekursion gestartet.

Diese nimmt:

- Einen Versatz (Startwerte: 0 (erster Aufruf), 1 (zweiter Aufruf))
- `VerwendeteTeile` (Startwert: von Kernteil-Rekursion verwendete Teile)
- `Eckteil-Slice` (Startwert: leere Slice)

Die Randfiguren, zu denen passende Ecken gefunden werden müssen, werden durch Go's „lexical scope“ zugänglich: entsprechend ist die Funktion im Körper von `probiereKerne` deklariert.

Diese geht ebenfalls noch nicht verwendete Teile durch und versucht, diese als Eckteile einzusetzen. Passt eines, wird es an eine Kopie der Slice angehängen und es wird nach Markieren des Teils als verwendet rekursiv weiterprobiert, bis schließlich alle drei Ecken bestimmt sind. Funktioniert keine Eckenanordnung, wird das Kernteil schlicht „verworfen“: Die Funktion `returnt`, und das Ausprobieren der Kernteile fährt fort.

Wurde eine Eckenanordnung gefunden, erfolgt nach einem einfachen „sanity-check“ die Ausgabe. Diese ist algorithmisch nicht besonders interessant; das Ausgabeformat für das gelöste Puzzle ist als String in ASCII-Art gegeben, mit durchnummerierten Platzhaltern für die „Figurengruppen“:

- 0 - erste Ecke
- 1 - zweite Ecke
- 2 - dritte Ecke
- 3 - Randfiguren
- 4 - Kernfiguren

Um die Ausgabe einer Figur auf ein Zeichen zu beschränken, werden Buchstaben anstatt von Zahlen verwendet (siehe Ausgabe).

Da Figurengruppen im Format-String nicht unbedingt die richtige Reihenfolge haben, müssen hierfür Slices die Umordnung der Elemente angeben (siehe Quellcode Z. 170-184 die Zuweisung der `reihenfolge`).

Schließlich werden gegebene Teile, gefundene Lösung und die verstrichene Zeit ausgegeben, woraufhin das Programm mit `os.Exit` beendet wird, so dass die Brute-Force, nach erfolgreicher Lösungsfindung,

stoppt.

Wird keine Lösung gefunden, kommt es nicht zur Programmbeendigung und der Code am Programmende wird ausgeführt. Hier wird zunächst ausgegeben, dass keine Lösung möglich ist, gefolgt von der verstrichenen Zeit.

## Quellcode

### main.go

```
1 package main
2
3 import (
4     "fmt"
5     "io/ioutil"
6     "os"
7     "strconv"
8     "strings"
9     "time"
10 )
11
12 // Figur - Ganzzahl von -128 bis 127
13 type Figur int8
14
15 // Teil - Drei Figuren
16 type Teil [3]Figur
17
18 // Kernteil - Seite, Teil, und Vorheriges
19 type Kernteil struct {
20     seite      uint8
21     teil       uint8
22     vorheriges *Kernteil
23 }
24
25 // Eckteil - Seite und Teil
26 type Eckteil struct {
27     seite uint8
28     teil  uint8
29 }
30
31 // VerwendeteTeile - Flag
32 type VerwendeteTeile uint16
33
34 // Verwendet - Gibt zurück, ob ein Teil verwendet ist
35 func (teile VerwendeteTeile) Verwendet(teil uint8) bool {
36     return teile&(1<<teil) > 0
```

```

37 }
38
39 // Verwende - Verwendet ein Teil und gibt Flag zurück
40 func (teile VerwendeteTeile) Verwende(teil uint8) VerwendeteTeile {
41     return teile | (1 << teil)
42 }
43
44 func main() {
45     // Zeitmessung
46     nanos := time.Now().UnixNano()
47     verstricheneZeit := func() {
48         fmt.Println("Zeit verstrichen:",
↪ float64(time.Now().UnixNano()-nanos)/1e6, "ms")
49     }
50     // Eingabe
51     if len(os.Args) != 2 {
52         println("Verwendung: <pfad>")
53         return
54     }
55     text, err := ioutil.ReadFile(os.Args[1])
56     if err != nil {
57         panic(err)
58     }
59     lines := strings.Split(string(text), "\n")
60     // Teile einlesen
61     teile := make([]Teil, 9)
62     for l := 2; l < 11; l++ {
63         var teil Teil
64         figuren := strings.Split(lines[l], " ")
65         for f := 0; f < 3; f++ {
66             teilF, _ := strconv.Atoi(figuren[f])
67             teil[f] = Figur(teilF)
68         }
69         teile[l-2] = teil
70     }
71     // Rekursive Funktion, die "Kerne" erzeugt und probiert
72     var probiereKerne func(Figur, *Kernteil, VerwendeteTeile, uint8)
73     probiereKerne = func(passendZurStartfigur Figur, vorheriges
↪ *Kernteil, verwendeteTeile VerwendeteTeile, anzahlKernteile uint8) {
74         if anzahlKernteile == 6 {
75             // Abbruchbedingung: Sechs Kernteile
76             // Probieren: Erster Test: Passt die Seite des
↪ Kernteils zur ersten Seite (ist der Kern
↪ geschlossen?)

```

```

77         if teile[vorheriges.teil][vorheriges.seite] !=
           ↳ passendZurStartfigur {
78             return
79         }
80         // Randfiguren des Kerns
81         randfiguren := [6]Figur{}
82         cursor := vorheriges
83         for i := range randfiguren {
84             // Umgekehrte Reihenfolge
85             randfiguren[5-i] =
           ↳ teile[cursor.teil][(cursor.seite+2)%3]
86             if i < 5 {
87                 cursor = cursor.vorheriges
88             }
89         }
90         // Rekursive Funktion, die alle möglichen
           ↳ Eckenordnungen probiert
91         var probiereEcken func(uint8, VerwendeteTeile,
           ↳ []Eckteil)
92         probiereEcken = func(versatz uint8, verwendeteTeile
           ↳ VerwendeteTeile, ecken []Eckteil) {
93             if len(ecken) == 3 {
94                 if verwendeteTeile != 0b11111111 {
95                     // "Sanity-check": Am Ende
           ↳ müssen alle Teile
           ↳ verwendet worden sein
96                     panic("Teile mehrfach
           ↳ verwendet")
97                 }
98                 // 3 Passende Ecken wurden gefunden,
           ↳ das Puzzle ist gelöst!
99                 // Ausgabe:
100                 // Obere Figurenteile sind
           ↳ Großbuchstaben, untere
           ↳ Kleinbuchstaben
101                 figurenOben, figurenUnten :=
           ↳ [27]rune{}, [27]rune{}
102                 for f := 0; f < 27-8; f++ {
103                     figurenOben[f],
           ↳ figurenUnten[f] = rune('A'+f), rune('a'+f)
104                 }
105                 // Gibt für eine Figur den Buchstaben
           ↳ zurück
106                 figur := func(num Figur) rune {
107                     if num < 0 {

```





```

145         for i, ecke := range ecken {
146             teil := teile[ecke.teil]
147             figuren[i] =
↳ []Figur{teil[ecke.seite], teil[(ecke.seite+1)%3],
↳ teil[(ecke.seite+2)%3]}
148         }
149         if versatz == 0 {
150             // Kein Versatz: Randfiguren
↳ und Kernteile sind in
↳ richtiger Reihenfolge
151             figuren[3] = randfiguren[:]
152             cursor = vorheriges
153         } else {
154             // Versatz von 1: Randfiguren
↳ und Kernteile müssen um
↳ eins verschoben werden
155             // Dabei wird das erste
↳ Element zum neuen Letzten
156             figuren[3] =
↳ append(randfiguren[1:], randfiguren[0])
157             // "cursor" ist ein einfach
↳ verlinkter Listenknoten,
↳ der auf das Erste Element
↳ zeigt
158             // Dessen Nachfolger wird
↳ jetzt der Zweite Knoten
159             cursor.vorheriges =
↳ vorheriges
160         }
161         // Kernfiguren einsetzen
162         figuren[4] = make([]Figur, 12)
163         for i := 5; i >= 0; i-- {
164             teil := teile[cursor.teil]
165             // Pro Teil immer Zwei Figuren
166             figuren[4][i*2] =
↳ teil[(cursor.seite+1)%3]
167             figuren[4][i*2+1] =
↳ teil[cursor.seite]
168             cursor = cursor.vorheriges
169         }
170         // Im String sind die Figurengruppen
↳ nicht in der richtigen
↳ Reihenfolge:
171         // - Die Ecken müssen noch gedreht
↳ werden

```

```

172 // - Die Randfiguren sind kreisförmig
    ↳ angeordnet, und nicht von oben
    ↳ nach unten - links nach rechts
173 // - Die Kernfiguren ebenfalls
174 reihenfolge := [5][int]{
175     // Drehung der Ecken
176     {1, 2, 0},
177     {0, 1, 2},
178     {2, 0, 1},
179     // Kreisform Randfiguren
180     {0, 5, 1, 4, 2, 3},
181     // Kreisform Kernfiguren
182     {11, 0, 1, 2, 10, 3, 9, 4, 8,
    ↳ 7, 6, 5},
183 }
184 // N-tes Element jeder Figurengruppe
185 n := [5]int{}
186 for i, c := range format {
187     // Platzhalter für
    ↳ Figurengruppen sind die
    ↳ jeweiligen Zahlen
188     if c >= '0' && c <= '4' {
189         // Zahl 48 - 52
    ↳ (ASCII) in Zahl 0
    ↳ - 4 konvertieren,
    ↳ "c" gibt
    ↳ Figurengruppe an
190         c -= '0'
191         // N-te Figur aus der
    ↳ passenden
    ↳ Figurengruppe in
    ↳ der richtigen
    ↳ Reihenfolge
192         format[i] =
    ↳ figur(figuren[c][reihenfolge[c][n[c]])
193
194         // Nächstes Element
    ↳ der Figurengruppe
    ↳ beim nächsten
    ↳ Platzhalter
195         n[c]++
196     }
197 }
198 // Ausgeben
199 fmt.Println(string(format))
    verstricheneZeit()

```

```

200         // Programm beenden
201         os.Exit(0)
202     }
203     // Passende Figur ist anderer Teil der
204     ↪ Randfigur an entsprechender Stelle
    passendeFigur :=
205     ↪ -randfiguren[uint8(2*len(ecken))+versatz]
    for teil := uint8(0); teil < 9; teil++ {
206         if verwendeteTeile.Verwendet(teil) {
207             // Teil schon verwendet
208             continue
209         }
210         for seite, figur := range teile[teil]
211             ↪ {
212             // Seiten probieren
213             if figur == passendeFigur {
214                 // Erstelle Kopie der
215                 ↪ Ecken & füge neue
216                 ↪ Ecke hinzu
217                 eckenKopie :=
218                 ↪ make([]Eckteil, len(ecken)+1)
219
220                 for i, ecke := range
221                     ↪ ecken {
222                     eckenKopie[i]
223
224                 }
225                 eckenKopie[len(ecken)]
226                 ↪ = ecke
227
228                 = Eckteil{uint8(seite), teil}
229
230                 // Probiere weitere
231                 ↪ Ecken
232                 probiereEcken(versatz,
233                 ↪ verwendeteTeile.Verwende(teil), eckenKopie)
234
235                 break
236             }
237         }
238     }
239     // Ecken probieren, mit Versatz 0 und 1
240     probiereEcken(0, verwendeteTeile, []Eckteil{})
241     probiereEcken(1, verwendeteTeile, []Eckteil{})
242     return
243 }
244 for teil := uint8(0); teil < 9; teil++ {
245     if verwendeteTeile.Verwendet(teil) {
246         // Teil schon verwendet

```

```

234         continue
235     }
236     // Passende Seiten ermitteln
237     passendeFigur :=
↪ -teile[vorheriges.teil][vorheriges.seite]
238     passendeSeiten := []uint8{}
239     for seite, figur := range teile[teil] {
240         if figur == passendeFigur {
241             passendeSeiten =
↪ append(passendeSeiten, uint8(seite))
242         }
243     }
244     // Alle Seiten passen: Seiten sind identisch, Drehung
↪ ist egal
245     if len(passendeSeiten) == 3 {
246         passendeSeiten = []uint8{0}
247     }
248     for _, seite := range passendeSeiten {
249         // Kernteil erzeugen. Seite ist hierbei die,
↪ an die das nächste Kernteil anbinden muss.
250         kernteil := &Kernteil{(seite + 2) % 3, teil,
↪ vorheriges}
251         // Neues Kernteil übergeben, Teil verwenden,
↪ Anzahl Kernteile erhöhen
252         probiereKerne(passendZurStartfigur,
↪ kernteil, verwendeteTeile.Verwende(teil), anzahlKernteile+1)
253     }
254 }
255 }
256 for teil := uint8(0); teil < 5; teil++ {
257     for seite := uint8(0); seite < 3; seite++ {
258         // Unter 4 Teilen muss eines dabei sein, dass keine
↪ Ecke, sondern ein Kernteil ist
259         // Starte Brute-Force mit Kernteil
260         probiereKerne(-teile[teil][seite], &Kernteil{(seite
↪ + 2) % 3, teil, nil}, VerwendeteTeile(0).Verwende(teil), 1)
261     }
262 }
263 // Kein Abbruch ist erfolgt: Das Puzzle konnte nicht gelöst werden
264 fmt.Println("Puzzle unlösbar")
265 verstricheneZeit()
266 }

```

## Beispiele

In loesungen als Textdateien mit gleichem Namen wie die Aufgabe.

### puzzle0.txt

Teile:

```

    /-\      /-\      /-\      /-\      /-\      /-\      /-\      /-\
    ↪ /-\
    /b c\    /C b\    /b c\    /b D\    /C d\    /b D\    /C C\    /d C\
    ↪ /c B\
    / B \   / b \   / C \   / B \   / D \   / c \   / b \   / b \
    ↪ / d \
    /-----\ /-----\ /-----\ /-----\ /-----\ /-----\ /-----\
    ↪ /-----\ /-----\

```

Lösung:

```

    /-\
    /C b\
    / c \
    /-----\
    / \ C / \
    /c B\b b/B b\
    / b \ / D \
    /-----\
    / \ B / \ d / \
    /b C\c d/D c\C D/d C\
    / C \ / b \ / b \
    /-----\

```

Zeit verstrichen: 0.275944 ms

### puzzle1.txt

Teile:

```

    /-\      /-\      /-\      /-\      /-\      /-\      /-\      /-\
    ↪ /-\
    /B b\    /C d\    /b b\    /B b\    /d D\    /b D\    /B c\    /b c\
    ↪ /D c\
    / C \   / b \   / D \   / c \   / b \   / D \   / d \   / D \
    ↪ / C \
    /-----\ /-----\ /-----\ /-----\ /-----\ /-----\ /-----\
    ↪ /-----\ /-----\

```

Lösung:

```

    /-\

```

```

      /D b\
     /  c  \
    /-----\
   / \  C  / \
  /b C\c D/d b\
 /  B  \ /  C  \
/-----\
/ \  b  / \  c  / \
/b d\D D/d B\b B/b b\
 /  D  \ /  c  \ /  D  \
/-----\

```

Zeit verstrichen: 0.321481 ms

### puzzle2.txt

Teile:

```

      /-\      /-\      /-\      /-\      /-\      /-\      /-\      /-\
      ⇨ /-\
    /d c\    /c e\    /B C\    /d b\    /C D\    /d c\    /d B\    /D E\
      ⇨ /c d\
  /  b  \  /  B  \  /  e  \  /  B  \  /  B  \  /  e  \  /  c  \  /  b  \
      ⇨ /  E  \
/-----\ /-----\ /-----\ /-----\ /-----\ /-----\
      ⇨ /-----\ /-----\

```

Lösung:

```

      /-\
     /e B\
    /  C  \
   /-----\
  / \  c  / \
 /c b\B e/E c\
 /  d  \ /  d  \
/-----\
/ \  D  / \  D  / \
/B c\C B/b B\b E/e d\
 /  d  \ /  d  \ /  c  \
/-----\

```

Zeit verstrichen: 0.386294 ms

### puzzle3.txt

Teile:

```

    /-\      /-\      /-\      /-\      /-\      /-\      /-\      /-\
    ↪ /-\
    /K K\    /J I\    /K f\    /c K\    /G F\    /C D\    /g i\    /j K\
    ↪ /K d\
    / E \   / h \   / K \   / H \   / c \   / e \   / K \   / K \
    ↪ / C \
    /-----\ /-----\ /-----\ /-----\ /-----\ /-----\
    ↪ /-----\ /-----\

```

Lösung:

```

    /-\
    /K K\
    / j \
    /-----\
    / \ J / \
    /K H\h I/i K\
    / c \ / g \
    /-----\
    / \ C / \ G / \
    /K E\e D/d C\c F/f K\
    / K \ / K \ / K \
    /-----\

```

Zeit verstrichen: 0.272592 ms

### puzzle4.txt

Zusätzliches Beispiel:

```

10
9
10 -10 4
9 8 -7
10 -5 10
-2 10 7
6 5 -2
2 3 -4
-6 -8 10
-9 10 10
10 -3 -2

```

Puzzle unlösbar

Zeit verstrichen: 0.19849 ms