Lösung Aufgabe 1 „Wörter aufräumen“

Lösungsidee

Zunächst muss man überlegen, wann ein Wort „passt“: Dies ist der Fall, wenn die richtigen Buchstaben an den richtigen Stellen stehen, und das Wort die passende Länge besitzt.

Nach dieser Definition erhalten wir pro Lücke mehrere passende Wörter. Kommt für eine Lücke nur ein Wort infrage, ist klar, wie weiter zu verfahren ist: Das Wort muss eingesetzt werden.

Daraufhin kann es dann gestrichen werden: Wurde es schon so oft wie möglich (Wörter können mehrmals vorkommen) eingesetzt, kann es nicht mehr als passendes Wort für andere Lücken infrage kommen - muss also von der Menge passender Wörter anderer Lücken gestrichen werden. Für Lücken, die vorher zwei passende Wörter besaßen, von denen nun eins gestrichen wurde, ergibt sich daher wieder ein eindeutig einzusetzendes Wort... Dies geschieht so lange, bis schließlich alle Wörter eingesetzt sind.

Korrektheit

Damit für eine Lücke eindeutig feststeht, welches Wort einzusetzen ist - damit also die Lösung des Rätsels eindeutig ist -, darf nur ein Wort „passen“. Dieses muss dann wieder zu eindeutig passenden Wörtern führen, so lange, bis alle Lücken gefüllt sind.

Komplexität

Das Finden der passenden Wörter hat im Worst-Case quadratische Komplexität: Für jede der n Lücken müssen alle n Wörter geprüft werden, ob sie passen.

Das Einsetzen hat ebenfalls quadratische Komplexität: Es müssen n Lücken gefüllt werden. Bei bis zu $n-1$ weiteren Lücken muss das Wort als passendes Wort gestrichen werden.

Insgesamt ergibt sich also eine **Komplexität von** $O(n^2)$

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/raetsel0.txt`

Ausgabe

```
Lösung: <Satz mit eingesetzten Wörtern>
Zeit verstrichen: <Verstrichene Zeit in Millisekunden> ms
```

Bibliotheken

- `fmt`: Ausgabe & Formattierung
- `io/ioutil`: Einlesen der Rätseldatei
- `os`: Programmargumente
- `strings`: Auftrennen („splitten“) von Text
- `time`: Zeitmessung
- `unicode`: `IsLetter`-Funktion

Typen

`Text`

Rune-Slice: repräsentiert einen `string` in UTF-32

Zwar speicher-ineffizient im Vergleich zu Go UTF-8 `strings`, aber praktischer zu manipulieren und Buchstaben zu extrahieren:

```
utf8 := "Äpfel"
utf32 := Text(utf8)
// In UTF-8 sind Buchstaben wie "ä" zwei Zeichen
println(len(utf8)) // 6
// In UTF-32 ist ein Buchstabe genau ein Zeichen
println(len(utf32)) // 5
// Entsprechend fällt auch die Extraktion leichter aus
println(utf32[0] == 'Ä') // true
// Ebenso wie die Manipulation
raetsel := Text("Ä____ sind wohlschmeckend")
copy(raetsel[0:6], utf32)
println(string(raetsel)) // Äpfel sind wohlschmeckend
```

`Wort`

`Text` und die Anzahl der Vorkommen in der Wortliste.

`WortMitLuecken`

Auszufüllendes Lückenwort:

- Position im Lückentext
- Länge
- Gegebene Buchstaben als [position] = UTF-32-Zeichen
- Infragekommende („passende“) Wörter als [id] = true

Ablauf

Die Programmausführung beginnt mit dem Einlesen der durch die Programmargumente angegeben Datei und dem Zerlegen in zwei Zeilen. Die erste Zeile - der Lückentext - wird in einen UTF-32-Text konvertiert, die Wörter werden aus der zweiten Zeile extrahiert in dem man diese nach Leerzeichen „splittet“. Mit einer Abbildung von Wort nach ID wird sichergestellt, das Wörter nur einmal in den „Wortindex“, der Wörter nach Länge und ID kategorisiert, aufgenommen werden - bei mehrmaligem Vorkommen wird einfach die Anzahl erhöht (siehe `Wort`). Diese ID dient als Verweis auf das Wort (andere Programmiersprachen wie etwa Lua übernehmen dies sogar für einen; dies erlaubt dann Stringvergleiche in konstanter Zeit und beschleunigt Maps mit String-Keys - und spart zusätzlich noch Speicher).

Das Bestimmen passender Wörter ist relativ einfach: Mit zwei geschachtelten Schleifen prüft man für alle Wörter der passenden Länge, ob sie an jeder gegebenen Stelle den richtigen Buchstaben besitzen.

Weiter wird ein Index für Lücken nach passendem Wort angelegt. Die zum Einsetzen verwendete Funktion nimmt nun eine Lücke (`WortMitLuecken`) und die ID eines einzusetzenden Wortes. Das Wort wird dann eingesetzt, indem der entsprechende Bereich der Slice überschrieben wird. Die Anzahl der noch einzusetzenden Vorkommen wird dann verringert, sinkt sie auf 0 wird das Wort gestrichen: Überall, wo das Wort gepasst hätte - hier verwenden wir oben genannten Index für Lücken nach Wort - streichen wir das Wort. Passt dann nur noch ein Wort für eine Lücke, rufen wir rekursiv die Einsetzen-Funktion für diese Lücke und das passende Wort auf. Diese rekursiven Aufrufe halten so lange an, bis alle Wörter eingesetzt sind.

„Wörter mit Lücken“ werden aus der ersten Zeile extrahiert, indem diese zeichenweise durchgegangen ist. Buchstaben (nach `unicode.IsLetter`, also auch Unterstützung für andere Sprachen) und Leerzeichen sind Teil von Lückenworten, alles andere nicht. Wird ein Lückenwort durch ein Zeichen terminiert, kann die Suche nach passenden Wörtern gestartet werden, die eventuell schon das direkte Einsetzen des Wortes zur Folge haben kann.

Für die Ausgabe wird schließlich der UTF-32 Lösungstext in einen UTF-8 string umgewandelt. Zusätzlich wird noch die verstrichene Zeit ausgegeben.

Quellcode

main.go

```
1  package main
2
3  import (
4          "fmt"
5          "io/ioutil"
6          "os"
7          "strings"
8          "time"
9          "unicode"
10 )
11
12 // Text - Unicode-Zeichen-Array, um keine UTF-8-Probleme zu bekommen
```

```go
type Text []rune

// Wort - Text und Anzahl Vorkommen
type Wort struct {
	text   Text
	anzahl uint
}

// WortMitLuecken - Anfang und Ende im Lückentext, gegebene Buchstaben, und infragekommende Wörter
type WortMitLuecken struct {
	start              uint
	laenge             uint
	gegebeneBuchstaben map[uint]rune
	passendeWoerter    map[uint]bool
}

func main() {
	nanos := time.Now().UnixNano()
	// Einlesen
	if len(os.Args) != 2 {
		println("Verwendung: <pfad>")
		return
	}
	text, err := ioutil.ReadFile(os.Args[1])
	if err != nil {
		panic(err)
	}
	zeilen := strings.Split(string(text), "\n")
	aufgabe := Text(zeilen[0])
	_woerter := strings.Split(zeilen[1], " ")
	// Verarbeiten
	woerterNachLaenge := map[uint]map[uint]*Wort{}
	bekannteWoerter := map[string]uint{}
	for id, _wort := range _woerter {
		wort := Text(_wort)
		length := uint(len(wort))
		bekannt := bekannteWoerter[_wort]
		if bekannt != 0 {
			// Falls bekannt, einfach Anzahl erhöhen
			woerterNachLaenge[length][bekannt-1].anzahl++
		} else {
			// Ansonsten Wort einfügen in Index nach Länge
			if woerterNachLaenge[length] == nil {
				woerterNachLaenge[length] = map[uint]*Wort{}
```

```
57                          }
58                          woerterNachLaenge[length][uint(id)] = &Wort{wort, 1}
59                          // Wort als bekannt setzen. ID mit einem Offset von 1,
                             ↪  da bei einem miss 0 zurückgegeben wird, was von
                             ↪  der ID 0 unterschieden werden muss
60                          bekannteWoerter[_wort] = uint(id) + 1
61                  }
62          }
63          // Lücken, bei denen ein Wort passen würde, nach Wort-ID. Da die
             ↪  Wort-IDs von 0 - n sind, kann eine slice (array) verwendet werden
64          lueckenNachWort := make([]map[*WortMitLuecken]bool, len(_woerter))
65          for index := range lueckenNachWort {
66                  lueckenNachWort[index] = map[*WortMitLuecken]bool{}
67          }
68          // Aktuelles Wort mit Lücken
69          var wortMitLuecken *WortMitLuecken
70          // Setzt ein Wort an einer Stelle an
71          var setzeWortEin func(wortMitLuecken *WortMitLuecken, id uint)
72          setzeWortEin = func(wortMitLuecken *WortMitLuecken, id uint) {
73                  // Ersetzt mit dem richtigen Wort die entsprechende Stelle im
                     ↪  Text
74                  copy(aufgabe[wortMitLuecken.start:wortMitLuecken.start+wortMitLuecken.lae
                     ↪  Text(_woerter[id]))
75                  // Anzahl verringern
76                  woerterNachLaenge[wortMitLuecken.laenge][id].anzahl--
77                  if woerterNachLaenge[wortMitLuecken.laenge][id].anzahl == 0 {
78                          // Anzahl 0, Wort steht nicht mehr zur Verfügung
79                          delete(woerterNachLaenge[wortMitLuecken.laenge], id)
80                          delete(lueckenNachWort[id], wortMitLuecken)
81                          // Überall, wo das Wort infragekommt...
82                          for lueckenwort := range lueckenNachWort[id] {
83                                  // ... Wort streichen
84                                  delete(lueckenwort.passendeWoerter, id)
85                                  if len(lueckenwort.passendeWoerter) == 1 {
86                                          // Nur noch ein infragekommendes Wort
87                                          for id := range
                                             ↪  lueckenwort.passendeWoerter {
88                                                  // Einsetzen!
89                                                  setzeWortEin(lueckenwort,
    ↪  id)
90                                          }
91                                  }
92                          }
93                          // Fertig mit dem Wort: Keine Lücken dürfen es noch
                             ↪  als Kandidaten haben!
```

```
94                          lueckenNachWort[id] = nil
95                  }
96          }
97          findePassendeWoerter := func() {
98                  var id uint
99                  // Sucht ein passendes Wort
100         wortSuche:
101                 for _id, wort := range
                     ↪  woerterNachLaenge[wortMitLuecken.laenge] {
102                         for index, gegebenerBuchstabe := range
                             ↪  wortMitLuecken.gegebeneBuchstaben {
103                                 if wort.text[index] != gegebenerBuchstabe {
104                                         // Buchstabe an einer Stelle passt
                                             ↪  nicht: Wort kommt nicht infrage
105                                         continue wortSuche
106                                 }
107                         }
108                         id = _id
109                         // Passendes Wort merken
110                         wortMitLuecken.passendeWoerter[id] = true
111                         lueckenNachWort[id][wortMitLuecken] = true
112                 }
113
114                 if len(wortMitLuecken.passendeWoerter) == 1 {
115                         // Nur ein passendes Wort: Einsetzen!
116                         setzeWortEin(wortMitLuecken, id)
117                 }
118         }
119         for stelle, zeichen := range aufgabe {
120                 istBuchstabe := unicode.IsLetter(zeichen)
121                 istGesucht := zeichen == '_'
122                 if istBuchstabe || istGesucht {
123                         // Teil eines Wortes mit Lücken
124                         if wortMitLuecken == nil {
125                                 // Initialisierung falls erster Buchstabe /
                                     ↪  erste Lücke des Wortes
126                                 wortMitLuecken =
    ↪  &WortMitLuecken{uint(stelle), 0, map[uint]rune{}, map[uint]bool{}}
127                         }
128                         if istBuchstabe {
129                                 // Gegebenen Buchstaben eintragen
130                                 wortMitLuecken.gegebeneBuchstaben[uint(stelle)-wortMitLue
    ↪  = zeichen
131                         }
132                 } else if wortMitLuecken != nil {
```

```
133                         wortMitLuecken.laenge = uint(stelle) -
    ↪ wortMitLuecken.start
134                         findePassendeWoerter()
135                         wortMitLuecken = nil
136                 }
137         }
138         if wortMitLuecken != nil {
139                 // Falls der Text mit einem Wort endet
140                 wortMitLuecken.laenge = uint(len(aufgabe)) -
    ↪ wortMitLuecken.start
141                 findePassendeWoerter()
142         }
143         // lueckenNachWort nach Wörtern durchgehen, für die nur eine einzige
        ↪  Lücke infrage kommt, ist nicht nötig
144         // Ausgabe der Lösung
145         fmt.Println("Lösung:", string(aufgabe))
146         fmt.Println("Zeit verstrichen:",
    ↪ float64(time.Now().UnixNano()-nanos)/1e6, "ms")
147 }
```

Beispiele

raetsel0.txt

```
Lösung: oh je, was für eine arbeit!
Zeit verstrichen: 0.191576 ms
```

raetsel1.txt

```
Lösung: Am Anfang wurde das Universum erschaffen. Das machte viele Leute sehr
 ↪ wütend und wurde allenthalben als Schritt in die falsche Richtung
 ↪ angesehen.
Zeit verstrichen: 0.321691 ms
```

raetsel2.txt

```
Lösung: Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fand
 ↪ er sich in seinem Bett zu einem ungeheueren Ungeziefer verwandelt.
Zeit verstrichen: 0.208128 ms
```

raetsel3.txt

```
Lösung: Informatik ist die Wissenschaft von der systematischen Darstellung,
 ↪ Speicherung, Verarbeitung und Übertragung von Informationen, besonders
 ↪ der automatischen Verarbeitung mit Digitalrechnern.
Zeit verstrichen: 0.222585 ms
```

raetsel4.txt

```
Lösung: Opa Jürgen blättert in einer Zeitschrift aus der Apotheke und findet
 ↪ ein Rätsel. Es ist eine Liste von Wörtern gegeben, die in die richtige
 ↪ Reihenfolge gebracht werden sollen, so dass sie eine lustige Geschichte
 ↪ ergeben. Leerzeichen und Satzzeichen sowie einige Buchstaben sind schon
 ↪ vorgegeben.
Zeit verstrichen: 0.383151 ms
```

raetsel5.txt

Zusätzliches Beispiel:

```
p____c _____c __i_ __in
public static void main
```

```
Lösung: public static void main
Zeit verstrichen: 0.148274 ms
```