

# Labyrinth in Bash

Betriebssysteme & Rechnernetze  
Portfolioprüfung - Werkstück A

Vorgetragen von: Tim Fichtner, Muhammed Erdal Akkoc, Emmanuel David

# Agenda

- Aufgabenstellung
- Algorithmus „rekursives Backtracking“
- Funktionen im Code:
  - Benutzereingabe
  - Erstellen des Maze
  - Funktionsweise des Algorithmus
  - Ausgabe des Labyrinths
- Spielvorgang

# Aufgabenstellung

- Implementierung eines zufällig generierten Labyrinths in Bash-Skript
- Anforderungen im Detail:
  - Bestimmung der Höhe und Breite durch Benutzereingabe
  - Navigation durch das Maze mit Tastatureingaben
  - Nur freie Felder sollen begehbar sein
  - Bei Erreichen des Ziels soll Ausgabe auf dem Bildschirm erscheinen
  - Kommandozeilenanwendung

# Algorithmus „rekursives Backtracking“

- Solange Teillösung existiert:
  - Wähle Teillösung aus und markiere diese als „besucht“
  - Wenn Ziel erreicht  
→ **Lösung!**
  - Ansonsten neuer Rekursionsschritt
- Keine Teillösung mehr vorhanden: **Sackgasse/Dead-End** (→ letzten Schritt rückgängig machen)

# Rekursives Backtracking

- Nachteile (Bsp.: Labyrinth):
  - Performanz bzgl. Verarbeitungsleistung:  
→ jede geöffnete Instanz muss im Zwischenspeicher des Rechners bis zur vollständigen Abarbeitung gespeichert werden
- Vorteile(Bsp.: Labyrinth):
  - Erzielt visuell ansprechende, lange Pfade durch das Labyrinth (mentale Herausforderung für den Spieler)
  - Algorithmus weist weniger Dead-Ends (Sackgassen) im Vergleich zu anderen Algorithmen (Kruskal-Algorithmus) auf.

# Funktionen im Code

- Funktion „input\_achsen“:

```
1 function input_achsen {
2     local sAchse=$1
3     while true; do
4         read -p "$sAchse: " nVarAchse
5         if [[ ! $nVarAchse =~ ^[0-9]+$ ]];then
6             echo "Bitte nur Zahlen eingeben!"
7         elif [[ $nVarAchse -gt 29 ]] || [[ $nVarAchse -lt 5 ]];then
8             echo "Bitte zwischen 5 und 29 eingeben!"
9         elif [[ $(($nVarAchse % 2)) -eq 0 ]];then
10            echo "Bitte eine ungerade Zahl eingeben!"
11        else
12            if [[ $sAchse == "Höhe" ]];then
13                nHoehe=$((nVarAchse + 2))
14            else
15                nBreite=$((nVarAchse + 2))
16            fi
17        break
18    fi
19 done
20 }
```

# Funktionen im Code

- manuelle Bestimmung der Höhe und Breite:
  - Limitation der Benutzereingabe:
    - Untergrenze 5
    - Obergrenze 29
  - Benutzereingabe darf nur ungeraden Zahlen entsprechen
- Zuweisung der Benutzereingaben zu den Variablen „nHoehe“ und „nBreite“ um den Wert +2 erhöht

# Funktionen im Code

- „lab\_array\_initiieren“:

```
1 function lab_array_initiieren {
2   for ((y=0; y<nHoehe; y++)) ; do
3     for ((x=1; x<$(nBreite-1); x++)) ; do
4       lab_array[$((y * nBreite + x))]=0
5     done
6     lab_array[$((y * nBreite + 0))]=1
7     lab_array[$((y * nBreite + (nBreite - 1)))]=1
8   done
9   for ((x=0; x<nBreite; x++)) ; do
10    lab_array[$x]=1
11    lab_array[$(((nHoehe - 1) * nBreite + x))]=1
12  done
13  lab_array[$((nBreite + 2))]=2
14  lab_array[$(((nHoehe - 2) * nBreite + nBreite - 3))]=1
15 }
```



# Funktionen im Code

- Warum Benutzereingabe +2?
- Deklaration eines Arrays mit den Werten der Variablen „nHoehe“ und „nBreite“
- Zuweisung der Indexeinträge zu den Werten 0, 1 oder 2
  - 0 = Wand
  - 1 = Durchgang
  - 2= Spielertoken

# Funktionen im Code

- Festlegung der Spielerposition für den Start (Eingang)
- Festlegung des Ausgangs
- Visualisierung des Arrays bis hier (Beispiel Benutzereingabe: Höhe=5 und Breite=5):

```
1111111 1020001 1000001 1000001 1000001 1000101 1111111
```

# Funktionen im Code

Zweidimensionale Darstellung des Arrays

1	1	1	1	1	1	1
1	0	2	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	1	0	1
1	1	1	1	1	1	1

	■	•	■	■	■	
	■	■	■	■	■	
	■	■	■	■	■	
	■	■	■	■	■	
	■	■	■		■	

# Funktionen im Code

- „rekursives Backtracking“

```
1 function rekursives_backtracking {
2   local nAktuelle_zelle=$1
3   local nZufallszahl=$RANDOM
4   local i=0
5   lab_array[$nAktuelle_zelle]=1
6   while [ $i -le 3 ] ; do
7     nModZufallszahl=$((nZufallszahl % 4))
8     if [[ $nModZufallszahl -eq 0 ]];then
9       nRichtung=1
10    elif [[ $nModZufallszahl -eq 1 ]];then
11      nRichtung=-1
12    elif [[ $nModZufallszahl -eq 2 ]];then
13      nRichtung=$nBreite
14    elif [[ $nModZufallszahl -eq 3 ]];then
15      nRichtung=$((-$nBreite))
16    fi
17    local nNaechste_zelle=$((nAktuelle_zelle + nRichtung))
18    if [[ lab_array[$nNaechste_zelle] -eq 0 ]];then
19      local nUebernaechste_zelle=$((nNaechste_zelle + nRichtung))
20      if [[ lab_array[$nUebernaechste_zelle] -eq 0 ]];then
21        lab_array[$nNaechste_zelle]=1
22        rekursives_backtracking $nUebernaechste_zelle
23      fi
24    fi
25    i=$((i + 1))
26    nZufallszahl=$((nZufallszahl + 1))
27  done
28 }
```

# Funktionen im Code

- Zufällige Generierung des Labyrinths:
  - Bei jedem Schritt zur nächsten Zelle wird neue Rekursionsinstanz aufgerufen
  - Prüft in eine zufällige Richtung (oben, unten, rechts links) ob nächstes Feld eine Wand (Wert 0 hat) ist
    - wenn ja wird übernächste Zelle ebenfalls geprüft, um keine Durchgänge in Außenwänden und Innenwänden zu erzeugen
  - Erst wenn diese Bedingung gilt wird das zu erst geprüfte Feld im Array von 0 (Wand) auf 1 (Durchgang) gesetzt

# Funktionen im Code

	■	•	■	■	■	
	■	1		2	■	
	■	■	■		■	
	■	4		3	■	
	■	■	■		■	

# Funktionen im Code

- Sobald Zelle erreicht wurde, die alle benachbarten Zellen bereits besucht hat  
→ bricht die Rekursionsinstanz ab und vorherige wird weiter abgearbeitet  
(„Backtracking“)
- Garantie, dass alle Zellen am Ende des Algorithmus besucht wurden
- Startpunkt des Algorithmus = Endpunkt des Algorithmus

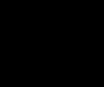
# Funktionen im Code

- „labyrinth\_ausgeben“:

```
1 function labyrinth_ausgeben {
2   for ((y=0; y<nHoehe; y++)) ; do
3     for ((x = 0; x<nBreite; x++ )) ; do
4       if [[ lab_array[$((y * nBreite + x))] -eq 0 ]];then
5         echo -n -e "\u2588\u2588"
6       elif [[ lab_array[$((y * nBreite + x))] -eq 2 ]];then
7         echo -n -e "\u25CF "
8       else
9         echo -n "  "
10      fi
11    done
12  echo
13 done
14 }
```



# Funktionen im Code

- Alle Felder im Array mit einer 0 werden auf der Konsole als doppeltes UTF-8 Symbol „“ ausgegeben
- Alle Felder im Array mit einer 2 werden auf der Konsole als UTF-8 Symbol „●“ ausgegeben
- Alle Felder im Array mit einer 1 werden auf der Konsole als leeren Feld ausgegeben

# Spielvorgang

- Eingabe von Höhe und Breite unter genannten Kriterien

```
Portfolioprüfung - Werkstück A - Alternative 7 - Labyrinth

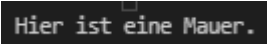
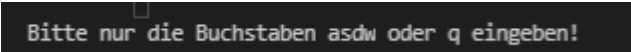
Bitte die Höhe und Breite des Labyrinths angeben.
Nur ungerade Zahlen zwischen 5 und 29 sind gültig.

Höhe: 5
Breite: 5
```

- Steuerung/Tastenfunktionen:
  - <W> = nach oben
  - <A> = nach links
  - <S> = nach unten
  - <D> = nach rechts
  - <Q> = Spiel jederzeit beenden

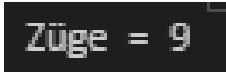
```
a -> Nach links bewegen
s -> Nach unten bewegen
d -> Nach rechts bewegen
w -> Nach oben bewegen
q -> Spiel beenden
```

# Spielvorgang

- Mittels der Steuerungstasten kann man den Spielertoken durch die Gänge bewegen
- Versucht man durch Wände zu gehen wird aufgrund einer Kollisionsprüfung ein entsprechender Hinweis ausgegeben 
- Betätigt man andere Tasten außer den Steuerungstasten und <Q> taucht ebenfalls ein Hinweis auf der Konsole auf 

# Spielvorgang

- Die Anzahl der Bewegungsschritte wird in einem Zähler auf der Konsole Ausgegeben

A terminal window snippet showing the text "Züge = 9" in a monospaced font.

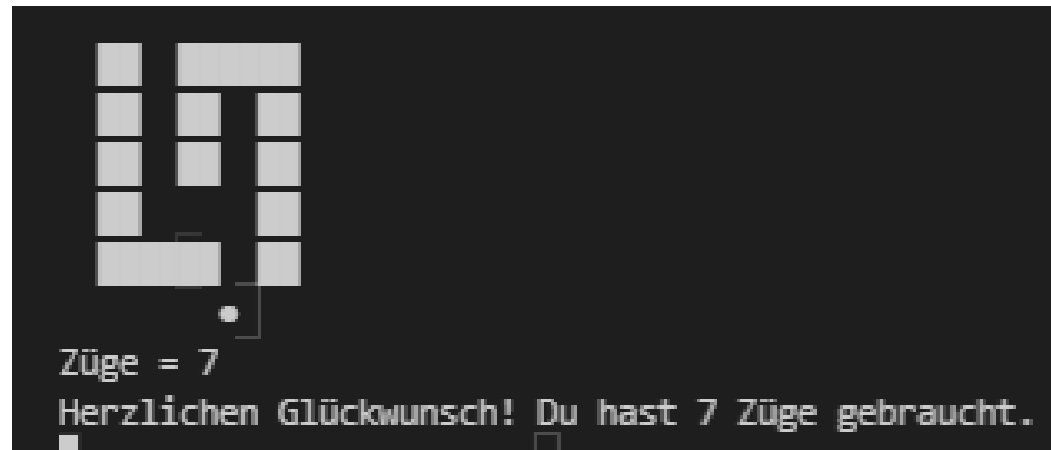
- Wenn man mit <Q> beendet, werden die bis dahin getätigten Schritte angezeigt und ein Hinweis, dass das Spiel beendet wurde

A terminal window showing a game board with a path of white squares on a black background. Below the board, it displays "Züge = 9" and "Das Spiel wurde beendet." followed by a cursor.

- Wenn man durch den Ausgang geht wird die Anzahl und eine Meldung, dass man das Spiel erfolgreich beendet hat ausgegeben

# Spielvorgang

- Mit <Enter> kann man, nach dem Verlassen durch <Q>, oder durch erfolgreichen Abschluss des Spiel das Programm beenden



Vielen Dank für Ihre Aufmerksamkeit!