

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

19. Dezember 2019

Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp>

1 Einführung

2 Einführung in C

3 Bibliotheken

3.1 Der Präprozessor

3.2 Bibliotheken einbinden

3.3 Bibliotheken verwenden

3.4 Projekt organisieren: make

4 Hardwarenahe Programmierung

...

4.4 volatile-Variable

4.6 Byte-Reihenfolge – Endianness

4.7 Binärdarstellung negativer Zahlen

4.8 Speicherausrichtung – Alignment

5 Algorithmen

5.1 Differentialgleichungen

...

...

3.4 Projekt organisieren: make

- Regeln
- Makros

3.4 Projekt organisieren: make

- Regeln

```
hello-6: hello-6.o pruzzel.o  
        gcc hello-6.o pruzzel.o -o hello-6
```

```
pruzzel.o: pruzzel.c pruzzel.h  
        gcc -Wall -O pruzzel.c -c
```

```
hello-6.o: hello-6.c pruzzel.h  
        gcc -Wall -O hello-6.c -c
```

- Makros

3.4 Projekt organisieren: make

- Regeln
- Makros

TARGET = hello-6

OBJECTS = hello-6.o pruzzel.o

HEADERS = pruzzel.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

pruzzel.o: pruzzel.c \$(HEADERS)

gcc \$(CFLAGS) pruzzel.c -c

hello-6.o: hello-6.c \$(HEADERS)

gcc \$(CFLAGS) hello-6.c -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

3.4 Projekt organisieren: make

- explizite und implizite Regeln

```
TARGET = hello-6
```

```
OBJECTS = hello-6.o pruzzel.o
```

```
HEADERS = pruzzel.h
```

```
CFLAGS = -Wall -O
```

```
$(TARGET): $(OBJECTS)
```

```
gcc $(OBJECTS) -o $(TARGET)
```

```
%.o: %.c $(HEADERS)
```

```
gcc $(CFLAGS) $< -c
```

```
clean:
```

```
rm -f $(OBJECTS) $(TARGET)
```

- Makros

3.4 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

4.4 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

PORTD = 0x01;

→ `(*(volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;`

↑
Umwandlung in Zeiger
auf **volatile** uint8_t

Zahl: 0x2B

Dereferenzierung des Zeigers

→ **volatile** uint8_t-Variable an Speicheradresse 0x2B

→ `PORTA = PORTB = PORTC = PORTD = 0` ist eine schlechte Idee.

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache ... **außer bei Datenaustausch!**

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

—→ Geschmackssache

... **außer bei Datenaustausch!**

- Dateiformate
- Datenübertragung

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

4.6 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

4.6 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

4.6 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

4.6 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

4.6 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→ $127 + 1 = -128$

4.6 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

16-Bit-Zahlen ohne Vorzeichen: `uint16_t`

→ Zahlenwerte von `0x0000` bis `0xffff` = 0 bis 65535

→ $65535 + 1 = 0$

`uint8_t`

0 bis 255

$255 + 1 = 0$

16-Bit-Zahlen mit Vorzeichen: `int16_t`

`0xffff` = 65535 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

`int8_t`

`0xff` = 255 = -1

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→ $32767 + 1 = -32768$

Literatur: <http://xkcd.com/571/>

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;—)

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;—)

Little-Endian:

als <code>int8_t</code> :	−93	(nur erstes Byte)
als <code>uint8_t</code> :	163	(nur erstes Byte)
als <code>int16_t</code> :	−28509	
als <code>uint16_t</code> :	37027	
<code>int32_t</code> oder größer:	37027	(zusätzliche Bytes mit Nullen aufgefüllt)

4.6 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;–)

Little-Endian:

als <code>int8_t</code> :	–93	(nur erstes Byte)
als <code>uint8_t</code> :	163	(nur erstes Byte)
als <code>int16_t</code> :	–28509	
als <code>uint16_t</code> :	37027	
<code>int32_t</code> oder größer:	37027	(zusätzliche Bytes mit Nullen aufgefüllt)

Big-Endian:

als <code>int8_t</code> :	–93	(nur erstes Byte)
als <code>uint8_t</code> :	163	(nur erstes Byte)
als <code>int16_t</code> :	–23664	
als <code>uint16_t</code> :	41872	
als <code>int32_t</code> :	–1550843904	(zusätzliche Bytes mit Nullen aufgefüllt)
als <code>uint32_t</code> :	2744123392	
als <code>int64_t</code> :	–6660823848880963584	
als <code>uint64_t</code> :	11785920224828588032	

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;
```

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;    uint8_t a;  
uint16_t b;        uint8_t c;  
uint8_t c;         uint16_t b;
```

4.7 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;  
  
uint8_t a;  
uint8_t c;  
uint16_t b;
```

Fazit:

- **Adressen von Variablen sind systemabhängig**
- Bei Definition von Datenformaten Alignment beachten → effizienter

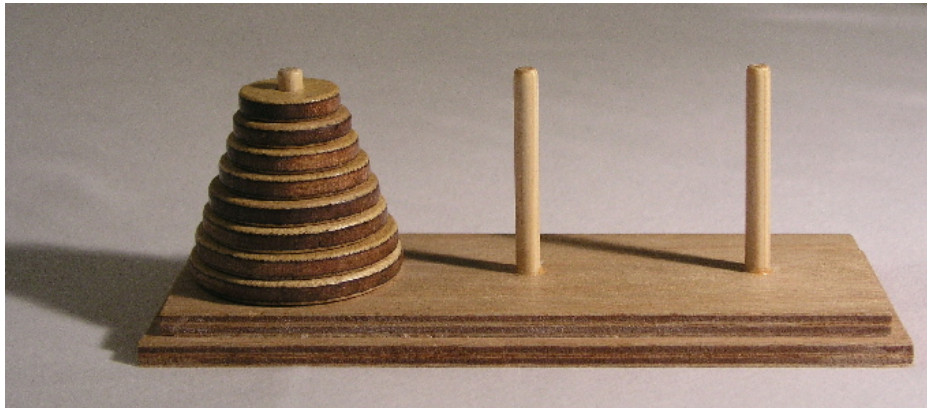
5.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

5.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

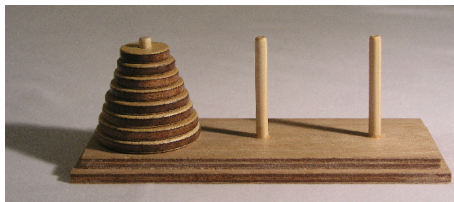


5.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.

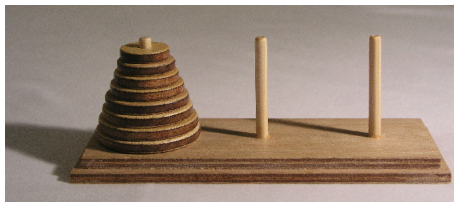


5.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz



5.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz

```
void move (int from, int to, int disks)
{
    if (disks == 1)
        move_one_disk (from, to);
    else
    {
        int help = 0 + 1 + 2 - from - to;
        move (from, help, disks - 1);
        move (from, to, 1);
        move (help, to, disks - 1);
    }
}
```

5.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz

32 Scheiben:

```
$ time ./hanoi-9a
...
real      0m32,712s
user      0m32,708s
sys       0m0,000s
```

5.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz

32 Scheiben:

```
$ time ./hanoi-9a
```

```
...
```

```
real      0m32,712s
```

```
user      0m32,708s
```

```
sys       0m0,000s
```

→ etwas über 1 Minute
für 64 Scheiben

5.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz

32 Scheiben:

```
$ time ./hanoi-9a
...
real      0m32,712s
user      0m32,708s
sys       0m0,000s
```

~~→ etwas über 1 Minute
für 64 Scheiben~~

Für jede zusätzliche Scheibe verdoppelt sich die Rechenzeit!

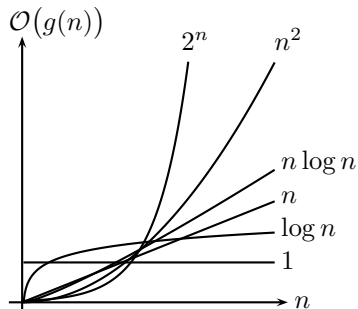
→ $\frac{32,712 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4452 \text{ Jahre}$
für 64 Scheiben

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Für jede zusätzliche Scheibe verdoppelt sich die Rechenzeit!

→ $\frac{32,712 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4452 \text{ Jahre}$
für 64 Scheiben



n : Eingabedaten

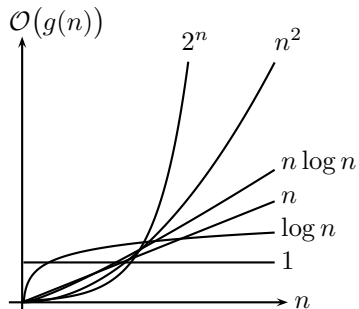
$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Beispiel: Sortieralgorithmen

- Minimum suchen: $\mathcal{O}(?)$



n : Eingabedaten

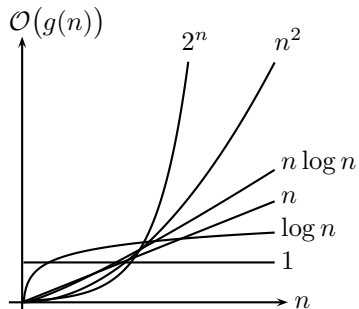
$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Beispiel: Sortieralgorithmen

- Minimum suchen: $\mathcal{O}(?)$
- ... mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

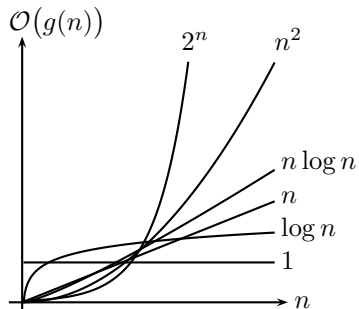
$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Beispiel: Sortieralgorithmen

- Minimum suchen: $\mathcal{O}(n)$
- ... mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

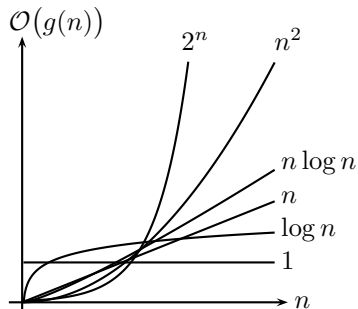
$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Beispiel: Sortieralgorithmen

- Minimum suchen: $\mathcal{O}(n)$
- ... mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

$g(n)$: Rechenzeit

Faustregel:

Schachtelung der Schleifen zählen

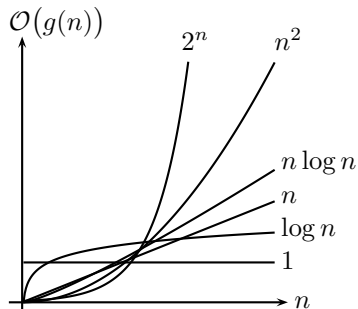
x Schleifen $\longrightarrow \mathcal{O}(n^x)$

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Beispiel: Sortieralgorithmen

- Minimum suchen: $\mathcal{O}(n)$
- ... mit Schummeln: $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:
→ Selectionsort



n : Eingabedaten

$g(n)$: Rechenzeit

Faustregel:

Schachtelung der Schleifen zählen

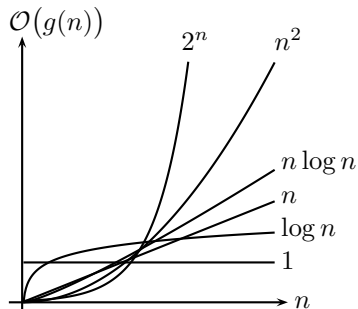
x Schleifen $\rightarrow \mathcal{O}(n^x)$

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

- Türme von Hanoi: $\mathcal{O}(2^n)$

Beispiel: Sortieralgorithmen

- Minimum suchen: $\mathcal{O}(n)$
- ... mit Schummeln: $\mathcal{O}(1)$
- Minimum an den Anfang tauschen, nächstes Minimum suchen:
→ Selectionsort: $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

Faustregel:

Schachtelung der Schleifen zählen

x Schleifen → $\mathcal{O}(n^x)$