

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

7. Dezember 2023

Hardwarenahe Programmierung

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

- 1 Einführung
- 2 Einführung in C
- 3 Bibliotheken
- 4 Hardwarenahe Programmierung
- 5 Algorithmen
 - 5.1 Differentialgleichungen
 - 5.2 Rekursion
 - 5.3 Aufwandsabschätzungen
- 6 Objektorientierte Programmierung
- 7 Datenstrukturen

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \quad \Rightarrow \quad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt = y_0 + v_{0y} \cdot t - \frac{1}{2}gt^2$$

$$v'_x(t) = 0 \quad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \quad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x += v_x * dt;$$

$$y'(t) = v_y(t) \quad y += v_y * dt;$$

$$v_x'(t) = 0 \quad v_x += 0 * dt;$$

$$v_y'(t) = -g \quad v_y += -g * dt;$$

5 Algorithmen

5.1 Differentialgleichungen

Beispiel 1: Gleichmäßig beschleunigte Bewegung

Beispiel 2: Mathematisches Pendel

$$\varphi'(t) = \omega(t)$$

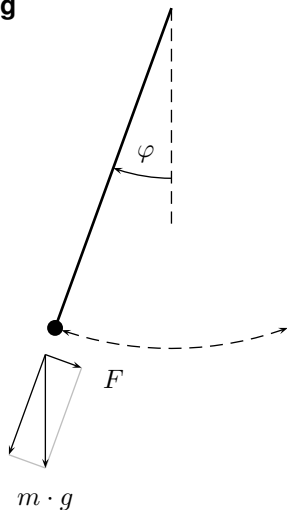
$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch):
Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch):
Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

Beispiel 3: Weltraum-Simulation

Praktikumsaufgabe



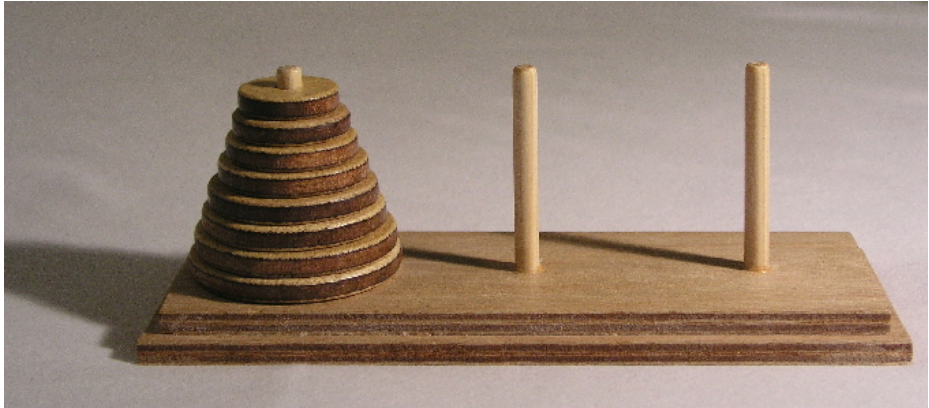
5.2 Rekursion

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

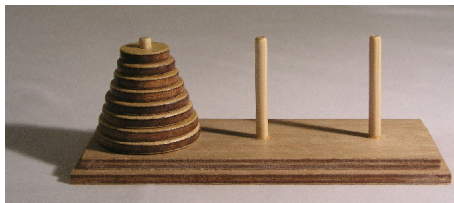


5.2 Rekursion

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.

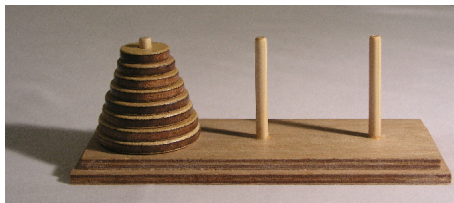


5.2 Rekursion

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);
    }
}
```

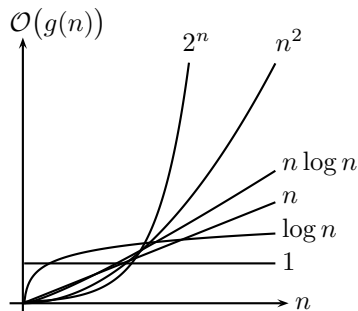
5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

→ $\frac{30,672 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4174 \text{ Jahre}$
für 64 Scheiben



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi: $\mathcal{O}(2^n)$

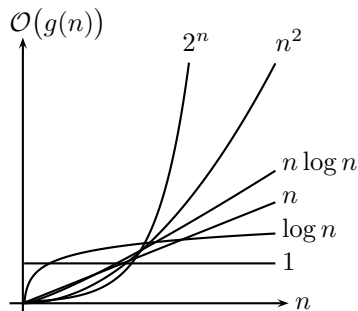
Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

$$\rightarrow \frac{30,672 \text{ s} \cdot 2^{32}}{3600 \cdot 24 \cdot 365,25} \approx 4174 \text{ Jahre}$$

für 64 Scheiben

Faustregel:

Schachtelung der Schleifen zählen
 k Schleifen ineinander $\rightarrow \mathcal{O}(n^k)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

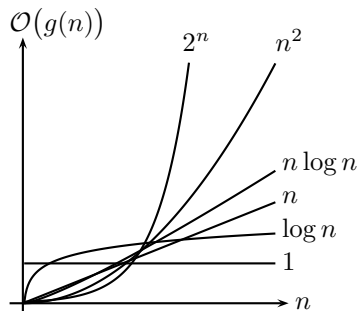
Faustregel:

Schachtelung der Schleifen zählen
 k Schleifen ineinander $\rightarrow O(n^k)$

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

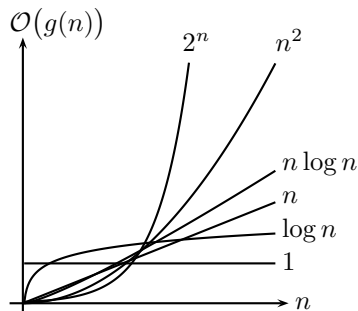
Schachtelung der Schleifen zählen

k Schleifen ineinander $\rightarrow O(n^k)$

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

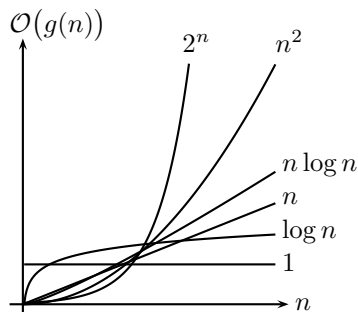
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

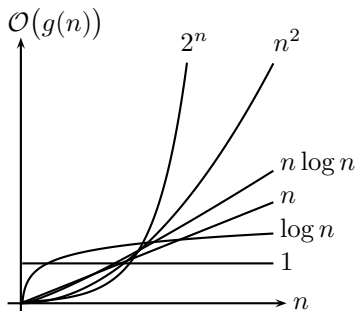
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

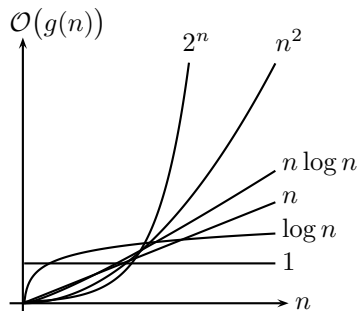
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

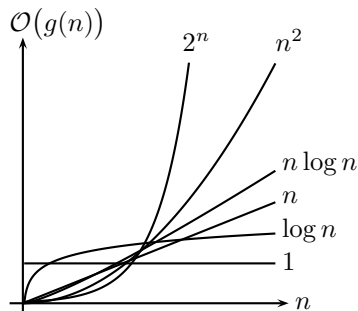
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

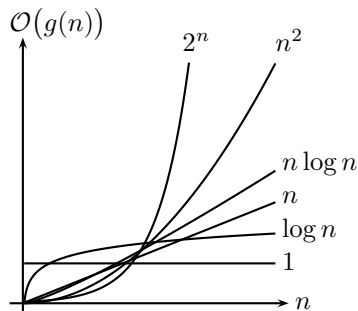
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort: $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

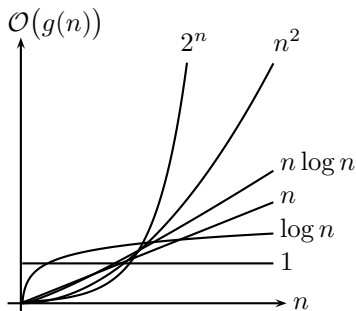
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort: $\mathcal{O}(n^2)$
- Bubble-Sort



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

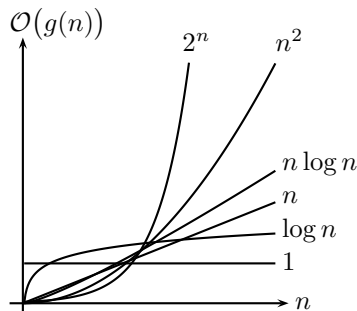
Faustregel:

Schachtelung der Schleifen zählen
 k Schleifen ineinander $\rightarrow \mathcal{O}(n^k)$

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort: $\mathcal{O}(n^2)$
- Bubble-Sort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten
 $g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

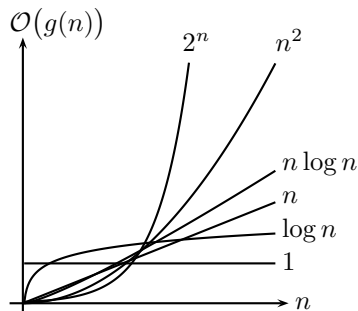
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort: $\mathcal{O}(n^2)$
- Bubble-Sort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Quicksort



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

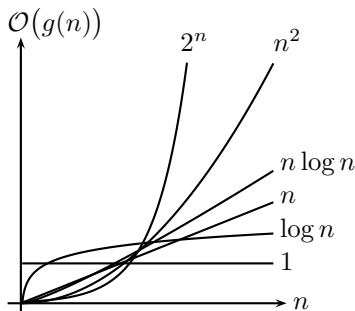
Schachtelung der Schleifen zählen

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

Beispiel: Sortieralgorithmen

Anzahl der Vergleiche bei n Strings

- Maximum suchen mit Schummeln: $\mathcal{O}(1)$
- Maximum suchen: $\mathcal{O}(n)$
- Selection-Sort: $\mathcal{O}(n^2)$
- Bubble-Sort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

Schachtelung der Schleifen zählen

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

Wie schnell ist RSA-Verschlüsselung?

$c = m^e \% N$ („%“ = „modulo“)

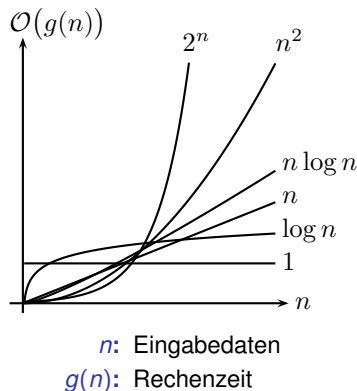
```
int c = 1;
for (int i = 0; i < e; i++)
    c = (c * m) % N;
```

- $\mathcal{O}(e)$ Iterationen
- mit Trick: $\mathcal{O}(\log e)$ Iterationen ($\log e$ = Anzahl der Ziffern von e)

Jede Iteration enthält eine Multiplikation und eine Division.

Aufwand dafür: $\mathcal{O}(\log e)$

\rightarrow Gesamtaufwand: $\mathcal{O}((\log e)^2)$



5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

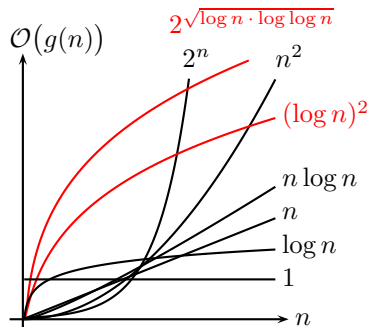
Schachtelung der Schleifen zählen

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

Wie schnell ist RSA?

(n = typische beteiligte Zahl, z. B. e, p, q)

- Ver- und Entschlüsselung (Exponentiation):
 $\mathcal{O}((\log n)^2)$
- Schlüsselerzeugung (Berechnung von d):
 $\mathcal{O}((\log n)^2)$
- Verschlüsselung brechen (Primfaktorzerlegung):
 $\mathcal{O}(2^{\sqrt{\log n \cdot \log \log n}})$



n : Eingabedaten

$g(n)$: Rechenzeit

Die Sicherheit von RSA beruht darauf, daß das Brechen der Verschlüsselung aufwendiger ist als $\mathcal{O}((\log n)^k)$ (für beliebiges k).

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

Schachtelung der Schleifen zählen

k Schleifen ineinander $\rightarrow O(n^k)$

Wie schnell ist RSA?

(n = typische beteiligte Zahl, z. B. e, p, q)

- Ver- und Entschlüsselung (Exponentiation):

$$O((\log n)^2)$$

$$O(n^2)$$

- Schlüsselerzeugung (Berechnung von d):

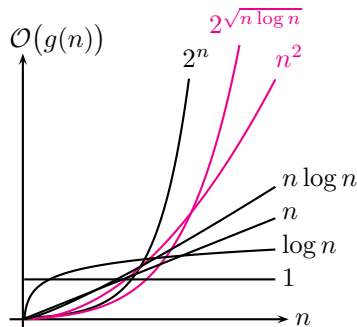
$$O((\log n)^2)$$

$$O(n^2)$$

- Verschlüsselung brechen (Primfaktorzerlegung):

$$O(2^{\sqrt{\log n \cdot \log \log n}})$$

$$O(2^{\sqrt{n \log n}})$$



n : Eingabedaten

$g(n)$: Rechenzeit

Die Sicherheit von RSA beruht darauf, daß das Brechen der Verschlüsselung aufwendiger ist als $O((\log n)^k)$ (für beliebiges k).