

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

14. 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**
 - 6.0** Dynamische Speicherverwaltung
 - 6.1** Konzepte und Ziele
 - 6.2** Beispiel: Zahlen und Buchstaben
 - 6.3** Unions
 - 6.4** Virtuelle Methoden
 - ...
- 7 Datenstrukturen**

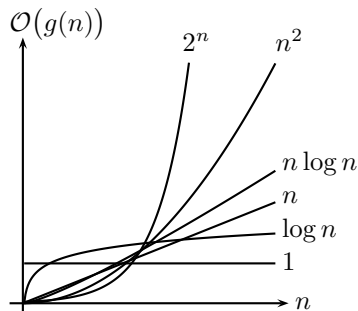
5.3 Aufwandsabschätzungen – Komplexitätsanalyse

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: $\mathcal{O}(2^n)$

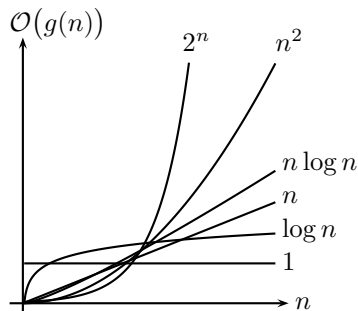
Für jede zusätzliche Scheibe
verdoppelt sich die Rechenzeit!

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

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“?

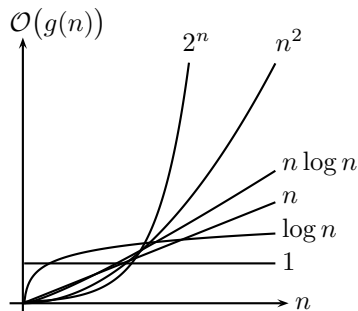
Faustregel:

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:

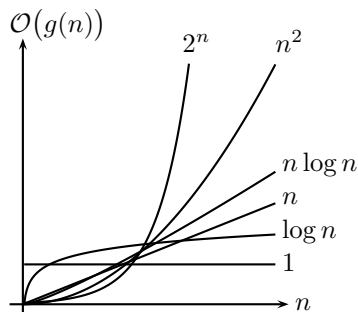
Schachtelung der Schleifen zählen

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:

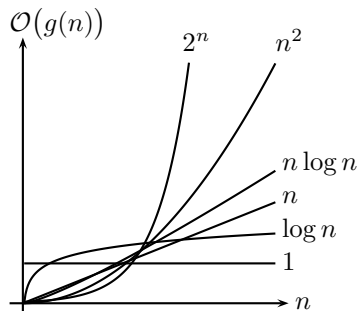
Schachtelung der Schleifen zählen

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:

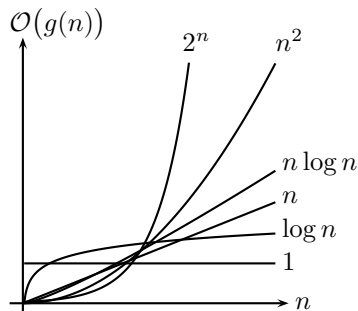
Schachtelung der Schleifen zählen

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:

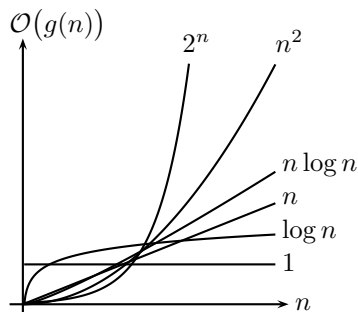
Schachtelung der Schleifen zählen

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:

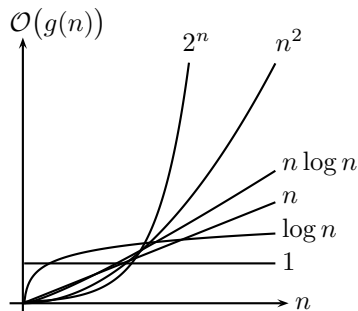
Schachtelung der Schleifen zählen

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:

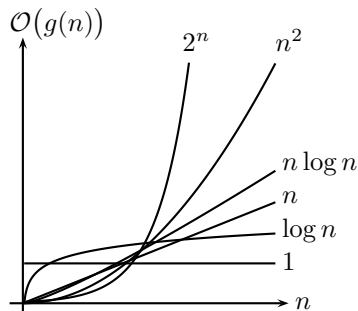
Schachtelung der Schleifen zählen

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:

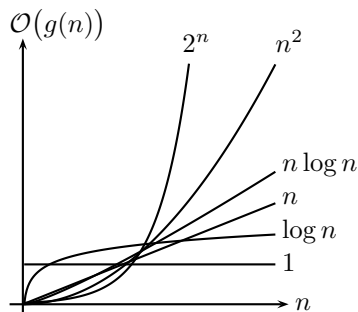
Schachtelung der Schleifen zählen

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:

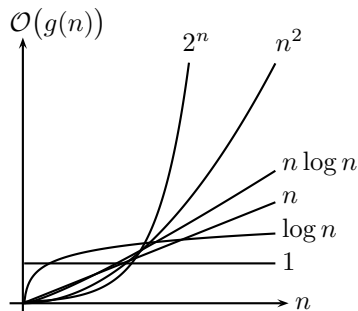
Schachtelung der Schleifen zählen

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:

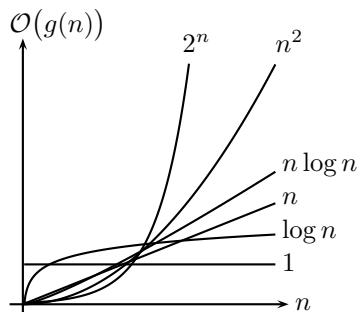
Schachtelung der Schleifen zählen

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:

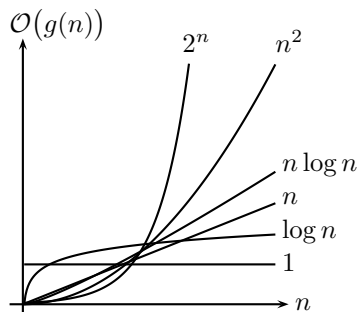
Schachtelung der Schleifen zählen

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“)

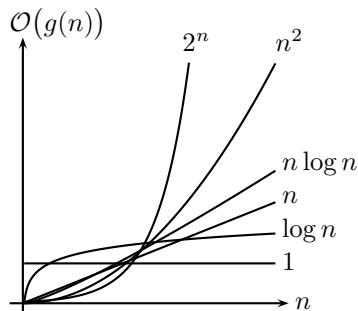
```
int c = 1;
for (int i = 0; i < e; i++)
    c = (c * m) % N;
```

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



n : Eingabedaten

$g(n)$: Rechenzeit

5.3 Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

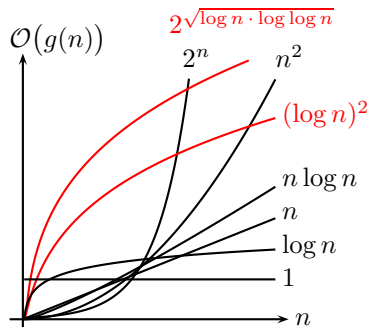
Schachtelung der Schleifen zählen

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

Wie schnell ist RSA?

(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):

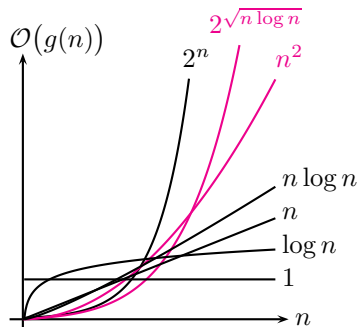
$$O((\log n)^2)$$

$$O(n^2)$$

- 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).

6 Objektorientierte Programmierung

6.0 Dynamische Speicherverwaltung

- Array: feste Anzahl von Elementen desselben Typs (z. B. 3 ganze Zahlen)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

...

~~name[3] = "Dieter";~~

6 Objektorientierte Programmierung

6.0 Dynamische Speicherverwaltung

```
#include <stdlib.h>
```

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name);  
    /* Speicherplatz freigeben */
```

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Array: Elemente desselben Typs (z. B. 3 ganze Zahlen)
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern → *Objekt*
- Problem: Die Elemente sind unterschiedlich groß (Speicherplatz).
- Lösung: Im Array nicht die Objekte selbst speichern, sondern Zeiger darauf.
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Was die Funktion bewirkt, hängt vom Typ des Objekts ab
- Realisierung über endlose **if**-Ketten

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Array: Elemente desselben Typs (z. B. 3 ganze Zahlen)
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern → *Objekt*
- Problem: Die Elemente sind unterschiedlich groß (Speicherplatz).
- Lösung: Im Array nicht die Objekte selbst speichern, sondern Zeiger darauf.
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- ~~Was die Funktion bewirkt~~ Welche Funktion aufgerufen wird, hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über ~~endlose if-Ketten~~ Zeiger, die im Objekt gespeichert sind (Genaugenommen: Tabelle von Zeigern)

→ nächste Woche

6 Objektorientierte Programmierung

6.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern → *Objekt*
- *Methoden* und *virtuelle Methoden*
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassenhierarchie* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Zeiger auf die Basisklasse dürfen auf Objekte der *abgeleiteten Klasse* zeigen
→ *Polymorphie*

6 Objektorientierte Programmierung

6.2 Beispiel: Zahlen und Buchstaben

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```


```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```

```
t_integer i = { 1, 42 };  
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```


explizite
Typumwandlung