

# Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

4. April 2024

# 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**
  - 7.1 Stack und FIFO
  - 7.2 Verkettete Listen
  - 7.3 Bäume

# 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**
  - 7.1 Stack und FIFO
  - 7.2 Verkettete Listen
  - 7.3 Bäume

# Algorithmen

## Differentialgleichungen

- Pendel
- Planetenbahnen

## Rekursion

- Türme von Hanoi

→ *Wie rechnet man das überhaupt aus?*

## Aufwandsabschätzungen

- Selectionsort
- Bubblesort
- Quicksort

→ *Wie rechnet man das möglichst effizient aus?*

- ? möglichst schnell
- ? mit möglichst wenig Speicherplatzverbrauch
- ? unter Berücksichtigung gegebener Randbedingungen

# Datenstrukturen

## Stack und FIFO

- effizientes Anfügen und Entfernen vorne und hinten
- effizienter direkter Zugriff auf Elemente in der Mitte
- ineffizientes Einfügen und Entfernen in der Mitte

## Verkettete Listen

- effizientes Einfügen und Entfernen in der Mitte
- ineffizienter direkter Zugriff auf Elemente in der Mitte

## Bäume

- Kompromiß

→ *Wie speichert man das möglichst effizient?*

- ? möglichst schnell
- ? mit möglichst wenig Speicherplatzverbrauch
- ? unter Berücksichtigung gegebener Randbedingungen

# Datenstrukturen

## Structs und Objekte

- zusammengehörige Daten gemeinsam speichern

## Stack und FIFO – Arrays

- effizientes Anfügen und Entfernen vorne und hinten
- effizienter direkter Zugriff auf Elemente in der Mitte
- ineffizientes Einfügen und Entfernen in der Mitte

## Verkettete Listen – Structs mit Pointern

- effizientes Einfügen und Entfernen in der Mitte
- ineffizienter direkter Zugriff auf Elemente in der Mitte

## Bäume – Structs mit Pointern

- Kompromiß

→ *Wie speichert man das möglichst effizient?*

- ? möglichst schnell
- ? mit möglichst wenig Speicherplatzverbrauch
- ? unter Berücksichtigung gegebener Randbedingungen

# Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Türme von Hanoi:  $\mathcal{O}(2^n)$

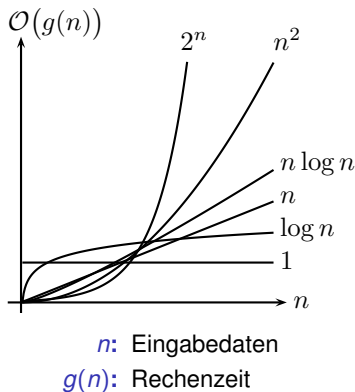
Für jede zusätzliche Scheibe  
verdoppelt sich die Rechenzeit!

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

für 64 Scheiben

Faustregel:

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



# Aufwandsabschätzungen – Komplexitätsanalyse

Wann ist ein Programm „schnell“?

Faustregel:

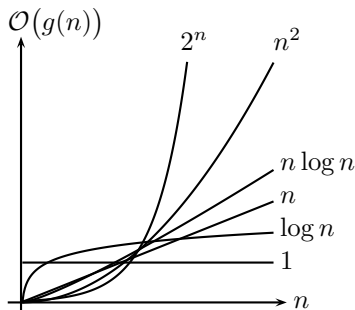
Schachtelung der Schleifen zählen

$k$  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



# 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  $d$ ):

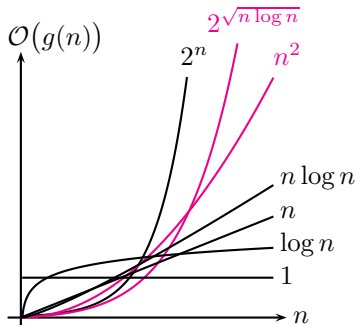
$$O((\log n)^2)$$

$$O(n^2)$$

- 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  $k$ ).**

# 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):

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

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

- Schlüsselerzeugung (Berechnung von  $d$ ):

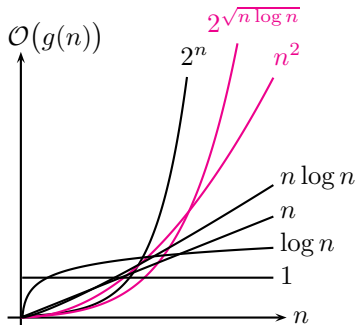
$$\mathcal{O}((\log n)^2)$$

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

- Verschlüsselung brechen (Primfaktorzerlegung):

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

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



$n$ : Eingabedaten

$g(n)$ : Rechenzeit

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

Mit Quantencomputer:  $\mathcal{O}((\log n)^3)$

# Algorithmen und Datenstrukturen in C/C++

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

- 1 Einführung
- 2 Arrays und Zeiger für Fortgeschrittene
- 3 Langzahl-Arithmetik
- 4 Kryptographie
- 5 C++
- 6 Datenorganisation
- ...



Änderungen  
vorbehalten

## 2 Arrays und Zeiger für Fortgeschrittene

Array:

```
char a[] = "Test";
```

Zeiger:

```
char *p = "Test";
```

- In beiden Fällen wird ein Array von ganzen Zahlen (5 **char**-Variable mit den Werten 84, 101, 115, 116 und 0) im Speicher angelegt.
- Links heißt das Array **a**; rechts ist es „anonym“.
- Rechts wird zusätzlich ein Zeiger **p** im Speicher angelegt, der auf das (anonyme) Array zeigt.
- **&a** ist dasselbe wie **a**, nämlich die Adresse des Arrays.
- **&p** ist die Adresse des Zeigers.
- **p** ist der Wert des Zeigers, momentan also die Adresse des (anonymen) Arrays.

## 2 Arrays und Zeiger für Fortgeschrittene

Array:

```
char a[] = "Test";
```

Zeiger:

```
char *p = "Test";
```

- In beiden Fällen wird ein Array von ganzen Zahlen (5 **char**-Variable mit den Werten 84, 101, 115, 116 und 0) im Speicher angelegt.
- Links heißt das Array **a**; rechts ist es „anonym“.
- Rechts wird zusätzlich ein Zeiger **p** im Speicher angelegt, der auf das (anonyme) Array zeigt.
- **&a** ist **fast** dasselbe wie **a**, nämlich die Adresse des Arrays **bzw. das Array selbst, das zuweisungskompatibel zu einem Zeiger auf Elemente des Arrays ist. & bewirkt hier eine (nicht explizite!) Typumwandlung.**
- **&p** ist die Adresse des Zeigers.
- **p** ist der Wert des Zeigers, momentan also die Adresse des (anonymen) Arrays.

## 2 Arrays und Zeiger für Fortgeschrittene

Array:

```
char *a[] = { "Dies", "ist", "ein", "Test" };
```

Zeiger:

```
char **p = a;
```

- Array von Zeigern auf **char**-Variable
- Zeiger auf das Array = Zeiger auf Zeiger auf **char**-Variable
- Schleife durch äußeres Array mit **p++** möglich

## 2 Arrays und Zeiger für Fortgeschrittene

Array:

```
char a[][5] = { "Dies", "ist", "ein", "Test" };
```

Zeiger:

```
char *p = a[0];
```

- zweidimensionales Array von **char**-Variablen
- Zeiger auf Array-Komponente  
= Zeiger auf eindimensionales Array  
= Zeiger auf **char**-Variable
- Schleife durch äußeres Array mit Zeiger-Arithmetik ~~nicht~~ möglich

  
nur mit Trick: `p += 5`

## 2 Arrays und Zeiger für Fortgeschrittene

```
typedef char string5[5];  
string5 a[] = { "Dies", "ist", "ein", "Test" };  
string5 *p = a;
```

- Array von Array von **char**-Variablen  
= zweidimensionales Array von **char**-Variablen
- Zeiger auf zweidimensionales Array
- Schleife durch äußeres Array mit **p++** möglich

→ Fazit:  
Ein Hoch auf **typedef**!

→ Trotzdem: **Vorsicht!**  
Ein **p++** auf einen Zeiger vom Typ **string5 \*p**  
ergibt anscheinend undefiniertes Verhalten!



## 2 Arrays und Zeiger für Fortgeschrittene

```
typedef char string5[5];  
string5 *p = { "Dies", "ist", "ein", "Test" };
```

- ~~• anonymes Array von Array von **char**-Variablen  
= anonymes zweidimensionales Array von **char**-Variablen~~
- ~~• Zeiger auf zweidimensionales Array~~
- ~~• Schleife durch äußeres Array mit **p++** möglich~~

Das Konstrukt { "Dies", "ist", "ein", "Test" }  
steht für ein Array von 4 Zeigern auf **char**-Variable.

**string5 \*p** hingegen erwartet einen Zeiger auf ein Array von 5 **char**-Variablen.  
Es bekommt die Adresse von "Dies" zugewiesen.

Durch das Erhöhen von **p** (um 5) zeigt es danach *zufällig* auf das "ist".  
Bei nochmaligem Erhöhen zeigt es auf das "in" von "ein".

(Auch ohne Optimierung werden die Strings "ist", "ein" und "Test"  
u. U. wegoptimiert.)

### 3 Langzahl-Arithmetik

Problem: Rechnen mit ganzen Zahlen, die größer sind als das, was der Rechner normalerweise verarbeiten kann

#### **Aufgabe: Addition langer Zahlen**

- (a) Überlegen Sie sich eine Datenstruktur, um eine lange Zahl zu speichern.
- (b) Schreiben Sie eine Funktion, die zwei lange Zahlen addiert.

### 3 Langzahl-Arithmetik

Problem: Rechnen mit ganzen Zahlen, die größer sind als das, was der Rechner normalerweise verarbeiten kann

- Grundrechenarten (einschließlich „modulo“):  
„schriftlich“ rechnen
- binäre Exponentiation:  
Basis fortlaufend quadrieren, ggf. damit multiplizieren  
Beispiel:  $x^9 = ((x^2)^2)^2 \cdot x$
- Suche nach  $d$  mit  $d \cdot e \bmod N = 1$ :  
erweiterter Euklidischer Algorithmus

→ RSA