

---

# Scripting und Algorithmen

Komplexität

Harald Lampesberger

FH OÖ, Department für Sichere Informationssysteme  
SAL2VO, SS23, Version: 27. März 2023

# Algorithmus

Ein Algorithmus ist eine Prozedur, die in einer endlichen Anzahl von Schritten (inklusive Schleifen und Verzweigungen) ein gegebenes Problem löst.

– nach Hetland, Python Algorithms (2014), S. 9

## Beispiele

- Suchen
- Sortieren
- Datenstruktur erzeugen, zB Liste von Aktionen in einem Spiel, Pfad in einem Graphen
- „Etwas Ausrechnen“, zB numerische Integration einer Gleichung

# Ziele des Algorithmenentwurfs

## Korrektheit

- Ein Algorithmus muss in allen spezifizierten Szenarien ein korrektes Ergebnis liefern
- Nachweis zB mittels formaler Verifikation, Model Checking, Testing

## Ressourcenverbrauch

- Wir haben zwei grundlegende Ressourcen: Zeit und Speicher
- Uns interessiert insbesondere, wie der Bedarf proportional zum Problem skaliert
- Nachweis mittels Laufzeit- und Speicherbedarfsanalyse, Testing (Benchmark)
- Ist ein „effizienter“ Algorithmus möglich?

# Analyse von Algorithmen

# Rekursion und Iteration

Ohne Schleifen wäre der Ressourcenverbrauch konstant

- Maßgeblich sind die Schleifendurchläufe / rekursiven Aufrufe, die proportional ( $\sim$ ) zur Problemgröße stattfinden

Problemgröße

- Übliches Symbol  $n$
- Wenn zB bei einem Algorithmus der Input eine lineare Datenstruktur ist, dann ist  $n$  die Größe der Datenstruktur

# Zeit- und Speicherkomplexität

## Zeitkomplexität

- Wie viele Schritte sind bis zur Lösung des Problems erforderlich?
- Effizienz = Programm läuft schnell

## Speicherkomplexität

- Wie viel (dynamischer) Speicher wird während der Ausführung benötigt?
- Effizienz = Programm und Daten sind möglichst kompakt

## Laufzeitanalyse

Das Ziel einer Laufzeitanalyse ist den ungünstigsten, günstigsten und mittleren Fall des Zeitbedarfs  $T$  eines Algorithmus, abhängig von der Problemgröße  $n$ , zu ermitteln.

Beispiel

```
def contains(strng, letter):  
    for char in strng:  
        if char == letter:  
            return True  
    return False
```

Ungünstig:  $T_{max}(n) = n \cdot \text{if ...}$

Günstig:  $T_{min}(n) = 1 \cdot \text{if ...}$

Im Mittel:  $T_{mit}(n) = \frac{n}{2} \cdot \text{if ...}$

## Laufzeitanalyse

Das Ziel einer Laufzeitanalyse ist den ungünstigsten, günstigsten und mittleren Fall des Zeitbedarfs  $T$  eines Algorithmus, abhängig von der Problemgröße  $n$ , zu ermitteln.

Beispiel

```
def contains(strng, letter):  
    for char in strng:  
        if char == letter:  
            return True  
    return False
```

Ungünstig:  $T_{max}(n) = n$  if ...

Günstig:  $T_{min}(n) = 1$  if ...

Im Mittel:  $T_{mit}(n) = \frac{n}{2}$  if ...

Zeitbedarf für Einzelschritte egal,  
da von Architektur/Kompiler abhängig



# Wachstum

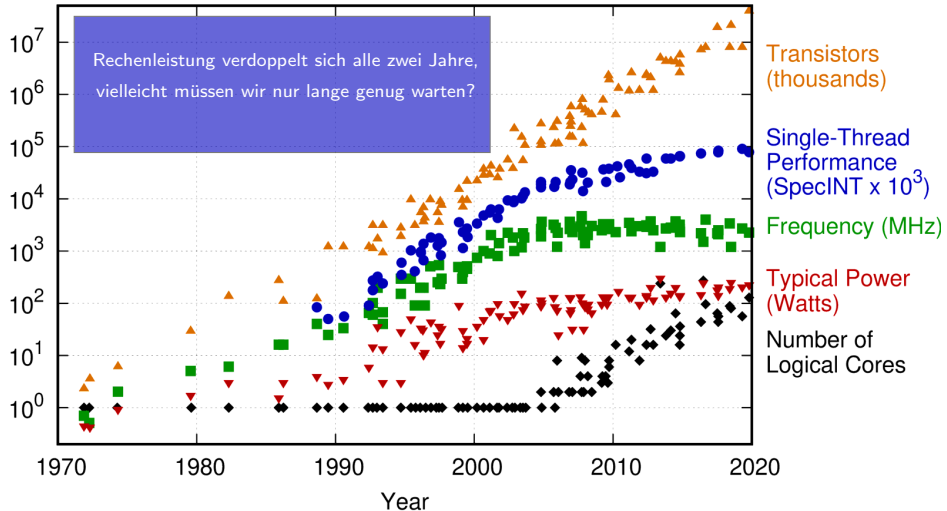
Uns interessiert primär, wie stark der Ressourcenaufwand mit der Problemgröße wächst

- Was passiert, wenn sich die zu sortierende Liste verdoppelt?
- Macht es für den Brute-Force-Angriff Sinn, die Rechenleistung zu verdoppeln?

Uns interessieren drei Fälle

- Ungünstiger Fall (worst case)
- Bester Fall (best case)
- Mittlerer Fall (average case)

## 48 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2019 by K. Rupp

<https://github.com/karlrupp/microprocessor-trend-data>

# Maschinenmodell

## Maschinenmodell für unsere Abschätzungen

- Ein Prozessor, keine Parallelisierung
- Wahlfreier Speicherzugriff (RAM)
- Speicherhierarchie (Cache, usw.) wird ignoriert
- Einzelne Schritte (Zuweisung, Verzweigung, usw.) werden in konstanter Zeit ausgeführt
- Laufzeitunterschiede zwischen Schritten werden vernachlässigt

# Landau-Notation (Big- $\mathcal{O}$ )

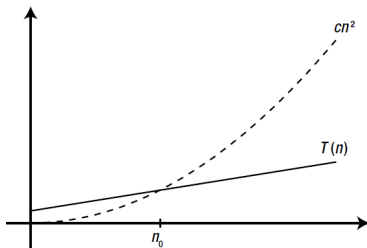
## Landau-Notation

Es sei  $g(n)$  eine Funktion der Problemgröße  $n$ . Wir definieren  $\mathcal{O}(g)$  als Menge aller Funktionen  $f(n)$ , für die gilt:

$$f(n) \leq c \cdot g(n).$$

für alle  $n \geq n_0$  und Symbol  $c$  ist ein beliebiger konstanter Faktor.

In anderen Worten:  $\mathcal{O}(g)$  ist die Menge der Funktionen, die nicht schneller wachsen als  $g$ .



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

Aus Hegland, Python Algorithms (2014), S. 13

# Asymptotisches Wachstum

Durch  $\mathcal{O}$ -Notation Fokus auf Wachstum

- Dominierender Term eines Polynoms zählt
- Niederrangige Terme und Faktoren werden gestrichen

Beispiele

- $4n^3 + 77n^2 - n \in \mathcal{O}(n^3)$
- $1000n + 100000 \in \mathcal{O}(n)$
- $-n^5 - n^3 + 100 \in \mathcal{O}(n^5)$
- $2^{100} + n^2 \in \mathcal{O}(n^2)$
- $2^n + n^{100} \in \mathcal{O}(2^n)$

# Die wichtigsten Landau-Symbole

$\mathcal{O}$  ist die asymptotische obere Schranke

$\Omega$  (Omega) ist die asymptotische untere Schranke

- $f$  ist in  $\Omega(g)$ , wenn  $f(n) \geq c \cdot g(n)$  für  $n \geq n_0$  gilt.
- In anderen Worten:  $\Omega(g)$  ist die Menge der Funktionen, die schneller wachsen als  $g$ .

$\Theta$  (Theta) ist die Schnittmenge aus  $\mathcal{O}$  und  $\Omega$

- $\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$
- Funktion  $f$  ist in  $\Theta(g)$ , wenn es zwei Konstanten  $c_1, c_2$  gibt, sodass  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  für  $n \geq n_0$  gilt.
- In anderen Worten:  $\Theta(g)$  ist die Menge aller Funktionen, die asymptotisch wie  $g$  wachsen.

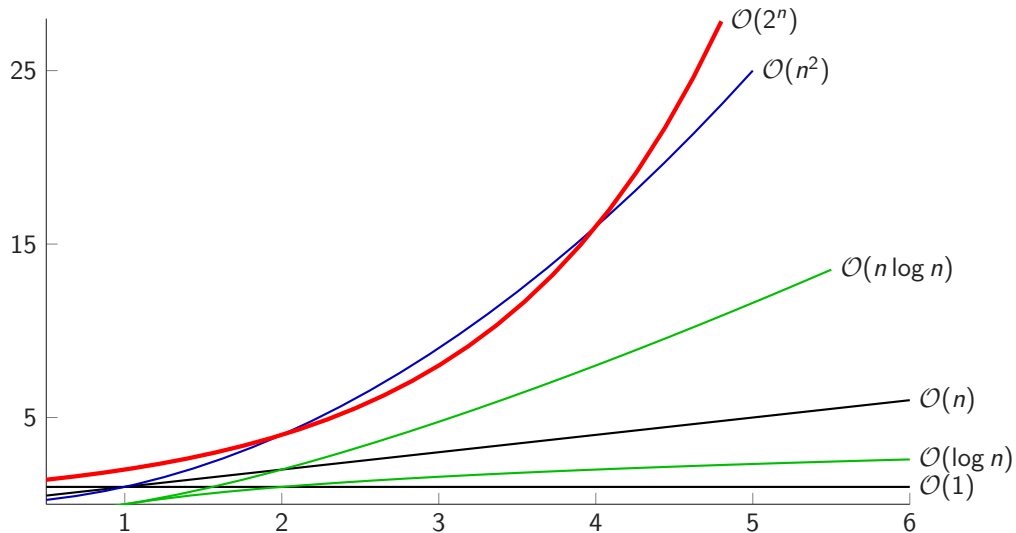
# Beispiele für asymptotische Laufzeiten

Komplexität	Aufwand	Beispiele
$\Theta(1)$	konstant	Array-Wert laden/schreiben, Hash-Table-Lookup
$\Theta(\log n)$	logarithmisch	Binäre Suche
$\Theta(n)$	linear	Schleife über eine Liste / Array
$\Theta(n \log n)$	loglinear	Optimales Sortieren
$\Theta(n^2)$	quadratisch	Vergleiche $n$ Objekte paarweise miteinander
$\Theta(n^3)$	kubisch	Alle kürzesten Pfade zwischen Knoten in einem Graph
$\mathcal{O}(n^k)$	polynomial	$k$ verschachtelte Schleifen über $n$ ( $k > 0$ )
$\Omega(k^n)$	exponentiell	Jede Teilmenge aus $n$ Objekten
$\Theta(n!)$	faktoriell	Alle Ordnungen (Permutationen) einer Liste

Nach Hegland, Python Algorithms (2014), S. 14



# Wachstum graphisch



# Rechenregeln

Einfache Anweisungen sind  $\sim \mathcal{O}(1)$

- Lesen/Schreiben von Variablen, usw. hängen nicht von Problemgröße  $n$  ab

Sequenzielle Ausführung  $\mathcal{O}_{seq} = \max(\mathcal{O}(p_1), \mathcal{O}(p_2))$

- Codestücke  $p_1$  und  $p_2$  werden nacheinander ausgeführt
- zB zwei einfache Anweisungen:  $\mathcal{O}_{seq} = \max(\mathcal{O}(1), \mathcal{O}(1)) = \mathcal{O}(1)$

Verzweigung  $\mathcal{O}_{if} = \max(\mathcal{O}(b), \max(\mathcal{O}(p_{if}), \mathcal{O}(p_{else})))$

- Bedingung  $b$  sequenziell vor Codestücken  $p_{if}, p_{else}$

Schleife  $\mathcal{O}_{loop} = \mathcal{O}(x \cdot \mathcal{O}_{pass})$

- $x$  ist die Anzahl der Durchläufe
- $\mathcal{O}_{pass} = \max(\mathcal{O}(b), \mathcal{O}(p_{loop}))$ , wo  $b$  die Schleifenbedingung und  $p_{loop}$  das Codestück bzw. der Rumpf der Schleife sind

# Anwendung

# Fallunterscheidungen

Reminder: Wir unterscheiden folgende Fälle

- Speicher- oder Zeitbedarf
- worst case / best case / average case
- Wenn nicht näher definiert, ist üblicherweise der worst case gemeint
- Oft wird in Literatur nur die obere Schranke  $\mathcal{O}$  betrachtet

Vorgehen

- Zuerst Problemgröße erkennen
- Alle einfachen Anweisungen
- Von inneren Schleifen zu den äußeren Schleifen

Praxisbeispiele

- Python-Datenstrukturen: <https://wiki.python.org/moin/TimeComplexity>
- $x$  **in**  $l$ st bei `list()`, average case  $\mathcal{O}(n)$
- $x$  **in**  $s$  bei `set()`, average case  $\mathcal{O}(1)$ , worst case  $\mathcal{O}(n)$
- $a + b$  mit `str()`, in allen Fällen  $\mathcal{O}(n^2)$ , wobei  $n = \max(\text{len}(a), \text{len}(b))$  ist

## Beispiel: Bubblesort

```
def bubblesort(lst):  
    n = len(lst) ■  $\sim \mathcal{O}(1)$   
    for upper_bound in range(n-1, 0, -1):  
        # countdown from n-1 .. 1  
        for i in range(upper_bound):  
            # swap elements if they are not in order  
            if lst[i] > lst[i+1]:  
                lst[i], lst[i+1] = lst[i+1], lst[i] ■  $\sim \mathcal{O}(1)$   
    return lst
```

bubblesort([7, 5, 3, 2])

## Beispiel: Bubblesort

```
def bubblesort(lst):  
    n = len(lst) ■  $\sim \mathcal{O}(1)$   
    for upper_bound in range(n-1, 0, -1):  
        # countdown from n-1 .. 1  
        for i in range(upper_bound):  
            # swap elements if they are not in order  
            if lst[i] > lst[i+1]: ■  $\sim \mathcal{O}(1)$   
                lst[i], lst[i+1] = lst[i+1], lst[i]  
    return lst
```

bubblesort([7, 5, 3, 2])


# Beispiel: Bubblesort

```
def bubblesort(lst):  
    n = len(lst) ■  $\sim \mathcal{O}(1)$   
    for upper_bound in range(n-1, 0, -1):  
        # countdown from n-1 .. 1  
        for i in range(upper_bound): ■  $\sim \mathcal{O}(n)$   
            # swap elements if they are not in order  
            if lst[i] > lst[i+1]:  
                lst[i], lst[i+1] = lst[i+1], lst[i]  
    return lst
```

bubblesort([7, 5, 3, 2])

## Beispiel: Bubblesort

```
def bubblesort(lst):  
    n = len(lst)  
    for upper_bound in range(n-1, 0, -1):  
        # countdown from n-1 .. 1  
        for i in range(upper_bound):  
            # swap elements if they are not in order  
            if lst[i] > lst[i+1]:  
                lst[i], lst[i+1] = lst[i+1], lst[i]  
    return lst
```



$\sim \mathcal{O}(1)$   
 $\sim \mathcal{O}(n^2)$

```
bubblesort([7, 5, 3, 2])
```

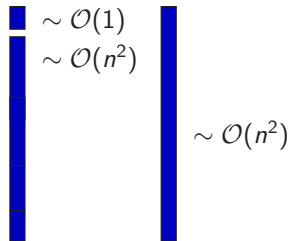
In allen Fällen: Zeitkomplexität  $\mathcal{O}(n^2)$ . Speicherkomplexität?



## Beispiel: Bubblesort

```
def bubblesort(lst):  
    n = len(lst)  
    for upper_bound in range(n-1, 0, -1):  
        # countdown from n-1 .. 1  
        for i in range(upper_bound):  
            # swap elements if they are not in order  
            if lst[i] > lst[i+1]:  
                lst[i], lst[i+1] = lst[i+1], lst[i]  
    return lst
```

bubblesort([7, 5, 3, 2])



In allen Fällen: Zeitkomplexität  $\mathcal{O}(n^2)$ . Speicherkomplexität?

# Zusammenfassung

Tradeoff: Zeit- und Speicherbedarf

- Zeit sparen auf Kosten von Speicher (zB Cache)
- Speicher sparen auf Kosten von Zeit (zB Kompression)

Laufzeitanalyse betrachtet Wachstum in Bezug zur Problemgröße

- Landau-Symbole als Abstraktion
- Konstante Faktoren gehen verloren, die dominierenden Terme bleiben

In der Praxis machen die konstanten Faktoren aber (oft) einen Unterschied

- zB  $T_A(n) = 100 \cdot n$  und  $T_B(n) = 25 \cdot n$
- Beide Algorithmen in  $\mathcal{O}(n)$ , aber in der Praxis wäre  $B$  4x schneller
- Darum gibt es Benchmarking (Python timeit)