

# Algorithmen und Datenstrukturen

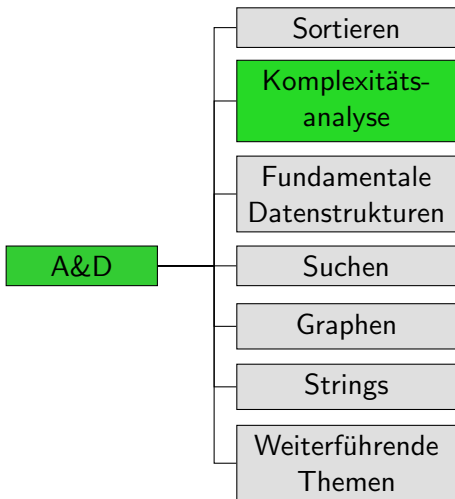
## A10. Laufzeitanalyse: Anwendung

Marcel Lüthi and Gabriele Röger

Universität Basel

23. März 2022

# Inhalt dieser Veranstaltung



# Kurze Wiederholung

# Landau-Symbole

- „ $f$  wächst genauso schnell wie  $g$ “

$$\Theta(g) = \{f \mid \exists c > 0 \exists c' > 0 \exists n_0 > 0 \forall n \geq n_0 : \\ c \cdot g(n) \leq f(n) \leq c' \cdot g(n)\}$$

# Landau-Symbole

- „ $f$  wächst genauso schnell wie  $g$ “

$$\Theta(g) = \{f \mid \exists c > 0 \exists c' > 0 \exists n_0 > 0 \forall n \geq n_0 : \\ c \cdot g(n) \leq f(n) \leq c' \cdot g(n)\}$$

- „ $f$  wächst nicht wesentlich schneller als  $g$ “

$$O(g) = \{f \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

# Landau-Symbole

- „ $f$  wächst genauso schnell wie  $g$ “

$$\Theta(g) = \{f \mid \exists c > 0 \exists c' > 0 \exists n_0 > 0 \forall n \geq n_0 : \\ c \cdot g(n) \leq f(n) \leq c' \cdot g(n)\}$$

- „ $f$  wächst nicht wesentlich schneller als  $g$ “

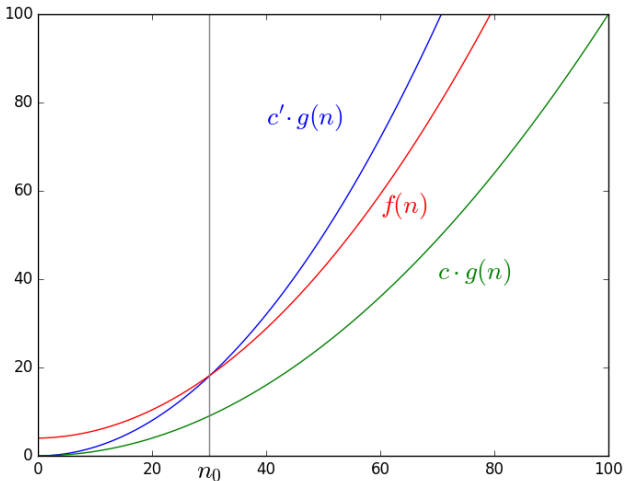
$$O(g) = \{f \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

- „ $f$  wächst nicht wesentlich langsamer als  $g$ “

$$\Omega(g) = \{f \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : c \cdot g(n) \leq f(n)\}$$

# Landau-Symbol Theta: Illustration

$$f \in \Theta(g)$$



# Interessante Funktionsklassen

In aufsteigender Ordnung (abgesehen von allgemeinen  $n^k$ ):

$g$	Wachstum
1	konstant
$\log n$	logarithmisch
$n$	linear
$n \log n$	leicht überlinear
$n^2$	quadratisch
$n^3$	kubisch
$n^k$	polynomiell (Konstante $k$ )
$2^n$	exponentiell



# Zusammenhänge

Es gilt:

- $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^k) \subset O(2^n)$   
(für  $k \geq 2$ )

# Zusammenhänge

Es gilt:

- $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^k) \subset O(2^n)$   
(für  $k \geq 2$ )
- $O(n^{k_1}) \subset O(n^{k_2})$  für  $k_1 < k_2$   
z.B.  $O(n^2) \subset O(n^3)$

# Rechenregeln

## ■ Produkt

$$f_1 \in O(g_1) \text{ und } f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

# Rechenregeln

## ■ Produkt

$$f_1 \in O(g_1) \text{ und } f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

## ■ Summe

$$f_1 \in O(g_1) \text{ und } f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

# Rechenregeln

## ■ Produkt

$$f_1 \in O(g_1) \text{ und } f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

## ■ Summe

$$f_1 \in O(g_1) \text{ und } f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

## ■ Multiplikation mit Konstante

$$k > 0 \text{ und } f \in O(g) \Rightarrow kf \in O(g)$$

$$k > 0 \Rightarrow O(kg) = O(g)$$

# Anwendung

# Schnelle $O$ -Analyse für häufige Code-Konstrukte I

- konstante Operation

<code>var = 4</code>	$O(1)$
----------------------	--------

# Schnelle $O$ -Analyse für häufige Code-Konstrukte I

- konstante Operation

var = 4	$O(1)$
---------	--------

- Sequenz konstanter Operationen

var1 = 4	$O(1)$	$O(123 \cdot 1) = O(1)$
var2 = 4	$O(1)$	
...	...	
var123 = 4	$O(1)$	



# Schnelle $O$ -Analyse für häufige Code-Konstrukte II

## ■ Schleife

<pre>for i in range(n):     res += i * m</pre>	$O(n)$	$O(n \cdot 1) = O(n)$
	$O(1)$	

# Schnelle $O$ -Analyse für häufige Code-Konstrukte II

## ■ Schleife

<code>for i in range(n):</code> <code>res += i * m</code>	$O(n)$	$O(n \cdot 1) = O(n)$
	$O(1)$	

<code>for i in range(n):</code> <code>for j in range(i):</code> <code>res += i * (m - j)</code>	$O(n)$	$O(n)$	$O(n^2)$
	$O(n)$	$O(n)$	
	$O(1)$		

$i$  hängt von  $n$  ab

# Schnelle $O$ -Analyse für häufige Code-Konstrukte III

## ■ if-then-else

if var < bound:	$O(1)$	$O(1)$	$O(1 + \max\{1, n\})$ $= O(n)$
res += var	$O(1)$	$O(1)$	
else:			
for i in range(n):	$O(n)$	$O(n \cdot 1)$	
res += i * n	$O(1)$	$= O(n)$	

# Schnelle $O$ -Analyse für häufige Code-Konstrukte III

## ■ if-then-else

if var < bound:	$O(1)$	$O(1)$	$O(1 + \max\{1, n\})$ $= O(n)$
res += var	$O(1)$	$O(1)$	
else:			
for i in range(n):	$O(n)$	$O(n \cdot 1)$	
res += i * n	$O(1)$	$= O(n)$	

**Achtung:** Kann zu unnötig hoher Abschätzung führen, wenn teurer Fall nur für kleine  $n$  auftritt (durch Konstante begrenzt).

## Beispiel: Worst Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

## Beispiel: Worst Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

- Worst case: break-Fall tritt nie ein.

## Beispiel: Worst Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

- Worst case: break-Fall tritt nie ein.
- $O(1 + n \cdot n \cdot 1) = O(n^2)$

## Beispiel: Worst Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

- Worst case: break-Fall tritt nie ein.
- $O(1 + n \cdot n \cdot 1) = O(n^2)$
- **Überschätzt?**  
Nein, beide Schleifen haben jeweils  $\Omega(n)$  Durchläufe.



## Beispiel: Best Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

## Beispiel: Best Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

- Best case: break jeweils direkt bei  $j = i$

## Beispiel: Best Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

- Best case: break jeweils direkt bei  $j = i$
- $O(1 + n \cdot 1 \cdot 1) = O(n)$

## Beispiel: Best Case für Insertionsort

---

```
1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 array[j], array[j-1] = array[j-1], array[j]
9             else:
10                break
```

---

- Best case: break jeweils direkt bei  $j = i$
- $O(1 + n \cdot 1 \cdot 1) = O(n)$
- Überschätzt?  
Nein, die äussere Schleifen hat  $\Omega(n)$  Durchläufe.

# Klausuraufgabe 2019

Betrachten Sie folgendes Codefragment. Geben Sie die asymptotische Laufzeit in Abhängigkeit von  $n \in \mathbb{N}$  in  $\Theta$ -Notation an und begründen Sie Ihre Antwort kurz (1-2 Sätze).

```
1  int result = 0;
2  if (n > 23) {
3      return result;
4  }
5  for (int i = 0; i < n; i++) {
6      for (int j = 0; j < n; j++) {
7          result += j;
8      }
9  }
10 return result;
```

# Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!

## Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!
- Beispiel: GTA online hatte viele Jahre eine Ladezeit von mehreren Minuten

# Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!
- Beispiel: GTA online hatte viele Jahre eine Ladezeit von mehreren Minuten
  - mehrere Minuten zum Parsen von 10 Megabyte JSON-Daten!



# Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!
- Beispiel: GTA online hatte viele Jahre eine Ladezeit von mehreren Minuten
  - mehrere Minuten zum Parsen von 10 Megabyte JSON-Daten!
  - vmtl. schlechte Library zum Parsen

# Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!
- Beispiel: GTA online hatte viele Jahre eine Ladezeit von mehreren Minuten
  - mehrere Minuten zum Parsen von 10 Megabyte JSON-Daten!
  - vmtl. schlechte Library zum Parsen
  - ungeeignete Datenstruktur zum Testen auf Duplikate

# Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!
- Beispiel: GTA online hatte viele Jahre eine Ladezeit von mehreren Minuten
  - mehrere Minuten zum Parsen von 10 Megabyte JSON-Daten!
  - vmtl. schlechte Library zum Parsen
  - ungeeignete Datenstruktur zum Testen auf Duplikate
  - nach Fix: 70% weniger Ladezeit

# Warum interessiert uns das alles?

- Weil Algorithmen/Datenstrukturen mit schlechter Laufzeitkomplexität zurückschlagen!
- Beispiel: GTA online hatte viele Jahre eine Ladezeit von mehreren Minuten
  - mehrere Minuten zum Parsen von 10 Megabyte JSON-Daten!
  - vmtl. schlechte Library zum Parsen
  - ungeeignete Datenstruktur zum Testen auf Duplikate
  - nach Fix: 70% weniger Ladezeit
  - <https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/index.html>

# Zusammenfassung

# Zusammenfassung

- In der Praxis können wir mit einfachen “Kochrezepten” recht schnell einen Eindruck von der Laufzeit eines Verfahrens bekommen.
- **Insertionsort** hat
  - im **besten Fall** Laufzeit  $\Theta(n)$
  - im **schlechtesten Fall** Laufzeit  $\Theta(n^2)$