Algorithmen und Datenstrukturen

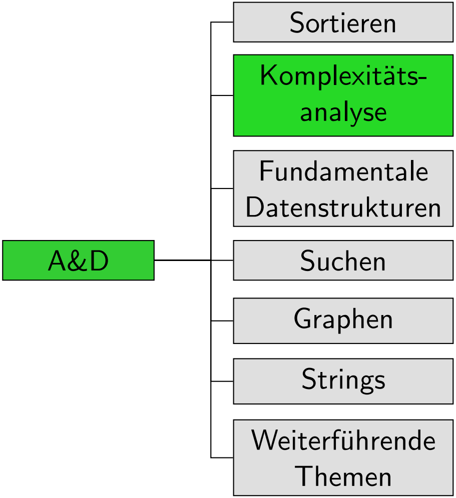
A10. Laufzeitanalyse: Anwendung

Marcel Lüthi and Gabriele Röger

Universität Basel

18. März 2021

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“

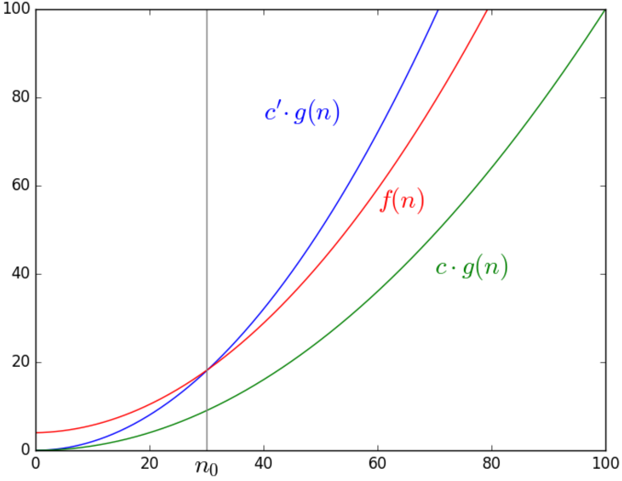
$$O(g) = \{f \mid \exists c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

- „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
  $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$

Rechenregeln

- Produkt
  $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$
- Summe
  $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$

Rechenregeln

- Produkt
  $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$
- Summe
  $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$
- Multiplikation mit Konstante
  $k > 0$ 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

`var = 4`	$O(1)$

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`	$O(1)$	
`...`	$\cdots$	
`var123 = 4`	$O(1)$	

Schnelle O-Analyse für häufige Code-Konstrukte II

- Schleife

`for i in range(n):`	$O(n)$	$O(n \cdot 1) = O(n)$
`  res += i * m`	$O(1)$	

Schnelle O-Analyse für häufige Code-Konstrukte II

- Schleife

<table>
<tr><td rowspan="2"><pre>for i in range(n):
  res += i * m</pre></td><td>$O(n)$</td><td rowspan="2">$O(n \cdot 1) = O(n)$</td></tr>
<tr><td>$O(1)$</td></tr>
</table>

<table>
<tr><td rowspan="3"><pre>for i in range(n):
  for j in range(i):
    res += i * (m - j)</pre></td><td>$O(n)$</td><td>$O(n)$</td><td rowspan="3">$O(n^2)$</td></tr>
<tr><td>$O(n)$</td><td rowspan="2">$O(n)$</td></tr>
<tr><td>$O(1)$</td></tr>
</table>

i hängt von n ab

Schnelle O-Analyse für häufige Code-Konstrukte III

- if-then-else

`if var < bound:`	$O(1)$	$O(1)$	
`  res += var`	$O(1)$	$O(1)$	
`else:`			$O(1 + \max\{1, n\})$ $= O(n)$
`  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:`	$O(1)$	$O(1)$	$O(1 + \max\{1, n\})$
`  res += var`	$O(1)$	$O(1)$	$= O(n)$
`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

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

Beispiel: Worst Case für Insertionsort

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

- Worst case: `break`-Fall tritt nie ein.

Beispiel: Worst Case für Insertionsort

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

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

Beispiel: Worst Case für Insertionsort

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                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

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

Beispiel: Best Case für Insertionsort

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

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

Beispiel: Best Case für Insertionsort

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
```

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

Beispiel: Best Case für Insertionsort

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        for j in range(i, 0, -1):  # j = i, ..., 1
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
            else:
                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 Θ-Notation an und begründen Sie Ihre Antwort kurz (1-2 Sätze).

```
int result = 0;
if (n > 23) {
    return result;
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        result += j;
    }
}
return result;
```

Klausuraufgabe 2019

Betrachten Sie folgendes Codefragment. Geben Sie die asymptotische Laufzeit in Abhängigkeit von $n \in \mathbb{N}$ in Θ-Notation an und begründen Sie Ihre Antwort mit der genauen Anzahl der Ausführungen der Anweisung in Zeile 4 (in Abhängigkeit von n).

```
1 int result = 0;
2 for (int i = 0; i < n; i++) {
3     for (int j = i; j < n; j++) {
4         result += j;
5     }
6 }
```

Jetzt: nur Θ-Notation

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 $\Theta(n^2)$