Der Groß-O-Kalkül

- Zunächst ein paar einfache "Rechen"-Regeln:

$$f \in O(f)$$
$$O(O(f)) = O(f)$$
$$k \cdot O(f) = O(k \cdot f) = O(f) \text{ für konstantes k}$$
$$O(f) + k = O(f + k) = O(f) \text{ für konstantes k}$$

Additionsregel

- Lemma, Teil 1: Für beliebige Funktionen f und g gilt:

$$f + g \in O(f + g) = O(\max(f, g))$$

- Zu beweisen: nur das rechte "="
- Zu beweisen: jede der beiden Mengen ist jeweils in der anderen Menge enthalten

"$\subseteq$": Sei $t(n) \in O(f(n) + g(n)) \Rightarrow$

$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0 : t(n) \leq c \cdot (f(n) + g(n))$

Abschätzung nach oben:

$c \cdot (f(n) + g(n)) \leq c \cdot 2 \cdot max\{f(n), g(n)\}$

Mit $\bar{c} = 2 \cdot c$ gilt $t(n) \leq \bar{c} \cdot max\{f(n), g(n)\}$

Also ist $t(n) \in O(max\{f(n), g(n)\})$

"$\supseteq$": Sei $t(n) \in O(\max\{f(n), g(n)\})$

$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : t(n) \leq c \cdot \max\{f(n), g(n)\}$

Abschätzung nach oben:

$max\{f(n), g(n)\} \leq f(n) + g(n)$

Also ist Bedingung für $t \in O(f(n) + g(n))$ mit denselben c, n_0 erfüllt

- Lemma, Teil 2: Für beliebige Funktionen f und g gilt:

$$O(f) + O(g) = O(f + g)$$

- Additionsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile hintereinander ausgeführt werden.

Multiplikationsregel

- Lemma: Für beliebige Funktionen f und g gilt:

$$f \cdot g \in O(f \cdot g) = O(f) \cdot O(g)$$

- Multiplikationsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile ineinander geschachtelt werden (Schleifen)

Teilmengenbeziehungen

- Lemma: Es gelten die folgenden Aussagen:
  1. $O(f(n)) \subseteq O(g(n)) \Leftrightarrow f(n) \in O(g(n))$
  2. $O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in O(g(n))$ und $g(n) \in O(f(n))$
  3. $O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \in O(g(n))$ und $g(n) \notin O(f(n))$
- Beweis von Teil 1:
  1. " $\Rightarrow$ " : trivial
  2. " $\Leftarrow$ " : $f(n) \in O(g(n))$.

Also $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$

Zu zeigen: jedes $t(n) \in O(f(n))$ ist auch in $O(g(n))$

Sei also $t(n) \in O(f(n))$.

Per Def gilt $\exists c' \in \mathbb{R}^+, n_0' \in \mathbb{N}\ \forall n \geq n_0' : t(n) \leq c' f(n)$

Wähle $\bar{n_0} = \max\{n_0, n_0'\}$ und $\bar{c} = c \cdot c'$

Damit gilt $t(n) \leq \bar{c} \cdot g(n)$

Also $t(n) \in O(g(n))$

- Teil 2 & 3 : analog

Transitivität von Groß–O

- Lemma: Falls $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$, dann ist $f(n) \in O(h(n))$.
- Beweis:
  Sei $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$
  $\Rightarrow$ $\exists c', c'' \in \mathbb{R}^+, n_0', n_0'' \in \mathbb{N}\ \forall n \geq \max\{n_0', n_0''\}$:
  $f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n)$
  $\Rightarrow$ Behauptung

Einfache Beziehungen

- Lemma: Für alle $m \in \mathbb{N}$ gilt
  $$O(n^m) \subseteq O(n^{m+1}).$$
  - Beweis: Übung
- Satz: Sei $p(n) := a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$, wobei $a_i \in \mathbb{R}_{\geq 0}$ für 0 ≤ i ≤ m.
  Dann gilt $p(n) \in O(n^m)$.
- Insbesondere: Es seien p_1 und p_2 Polynome vom Grad d_1 bzw. d_2, wobei die Koeffizienten vor n^{d1} und n^{d2} positiv sind.
  Dann gilt:
  a) $p_1 \in \Theta(p_2) \Leftrightarrow d_1 = d_2$
  b) $p_1 \in O(p_2) \Leftrightarrow d_1 < d_2$
  c) $p_1 \in \Omega(p_2) \Leftrightarrow d_1 > d_2$

- Für alle k, k fest, gilt: $n^k \in O(2^n)$
- Für alle k > 0 und ϵ > 0 gilt: $\log^k n \in O(n^\epsilon)$
- $2^{n/2} \in O(2^n)$
- Für beliebige positive Zahlen *a, b* ≠ 1 gilt:
  $$f(n) = \log_a n \in O(\log_b n)$$
  - Insbesondere:
    $$O(\log_b n) = O(\log_2 n)$$
  - Beweis: Übungsaufgabe. (Gleichzeitig ein Beleg, daß die Analysis-Vorlesung Anwendung hat.)

- Bemerkung: Groß–O definiert keine totale Ordnungsrelation auf der Menge aller Funktionen $\mathbb{R} \to \mathbb{R}$
- Beweis: Es gibt positive Funktionen f und g so, daß
  $f(n) \notin O(g(n))$ und auch $g(n) \notin O(f(n))$.
  Wähle zum Beispiel $f(n) = \sin(n) + 1$ und
  $g(n) = \cos(n) + 1$.
- Definition "polynomielle Zeit" :
  wir sagen, ein Algorithmus A mit Komplexität f(n) braucht höchstens polynomielle Rechenzeit (*polynomial time*), falls es ein Polynom P(n) gibt, so daß $f(n) \in O(P(n))$. A braucht höchstens exponentielle Rechenzeit (*exponential time*), falls es eine Konstante $a \in \mathbb{R}^+$ gibt, so daß $f(n) \in O(a^n)$.

Beweishilfe

Lemma: Es gilt:

(1) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c > 0 \Rightarrow O(f(n)) = O(g(n))$

(2) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f(n)) \subset O(g(n))$

Insbesondere hat man dann in Fall (1): $f(n) \in \Theta(g(n))$

Wie überprüft man diesen Limes?

Satz 7.44. (3. Regel von de l'Hôpital: "$x \to \infty$") *Seien f und g auf dem Intervall $]a, \infty[$ differenzierbar und es gelte $\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = 0$ (bzw. $= \infty$). Es existiere $\lim_{x\to\infty} \frac{f'(x)}{g'(x)} =: L$. Dann existiert auch $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ und ist gleich L. Kurz:*

$$\lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} \frac{f(x)}{g(x)}.$$

Hierarchie von Größenordnungen

Größenordnung	Name
`O(1)`	konstante Funktionen
`O(log n)`	logarithmische Funktionen
`O(log² n)`	quadratisch logarithmische Funktionen
`O(n)`	lineare Funktionen
`O(n log n)`	$n \log n$-wachsende Funktionen
`O(n²)`	quadratische Funktionen
`O(n³)`	kubische Funktionen
`O(nᵏ)`	polynomielle Funktionen (k konstant)

`f` heißt polynomiell beschränkt,
wenn es ein Polynom p mit $f \in O(p)$ gibt.

`f` wächst exponentiell, wenn es ein $k > 0$ gibt mit $f \in \Theta(k^n)$.

Skalierbarkeiten

- Annahme: 1 Rechenschritt ≙ 0.001 Sekunden → Maximale Eingabelänge bei gegebener Rechenzeit:

Laufzeit T(n)	1 Sekunde	1 Minute	1 Stunde
n	1000	60000	3600000
$n \log n$	140	4895	204094
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

- Annahme: Wir können einen 10-fach schnelleren Rechner verwenden → Statt eines Problems der Größe p kann in gleicher Zeit dann berechnet werden:

Laufzeit T(n)	neue Problemgröße
n	$10p$
$n \log n$	(fast 10)p $\approx 10 \frac{\log(p)}{\text{Lösung}(10p \text{ in } (p))} p$
n^2	$3.16p$ $\approx \sqrt{10}p$
n^3	$2.15p$ $\approx \sqrt[3]{10}p$
2^n	$3.32 + p$ $\approx \log 10 + p$

Allg. Bestimmung des Zeitaufwands mit Groß-O

- Sei A ein Programmstück, dann ist Zeitaufwand T(A) im Fall:
  - A ist einfache Anweisung oder arithm./log. Ausdruck → $T(A) = const \in O(1)$
  - A ist Folge von Anweisungen → Additionsregel anwenden
  - A ist `if`-Anweisung →
    (a) `if cond: B` → T(A) = T(cond) + T(B)
    (b) `if cond: B else: C` → T(A) = T(cond) + max(T(B), T(C))
  - A ist eine Schleife (while, for, ...) →

$$T(A) = \sum_{\text{Umlauf i}} T(\text{Anweisungen i}) + T(\text{Terminierungsbedingung i})$$

oft einfach

$$T(A) = \#\text{Umläufe} \cdot (T(\text{Anweisungen}) + T(\text{Terminierungsbedingung}))$$

  - A ist Rekursion → später

Beispiele zur Laufzeitabschätzung

Algorithmus *prefixAverages1(X)*

Eingabe: Ein Array X von n Zahlen

Ausgabe: Ein Array A von Zahlen, so daß gilt: A[i] ist das arithmetische Mittel der Zahlen X[0], ..., X[i]

Methode:

```
for i in range(0,n):
    a = 0                          → O(1)
    for j in range(0,i+1):
        a += X[j]                  → O(1)
    A[i] = a / (i + 1)             → O(1)
return A
```

i: O(1) = O(i) ⊆ O(n), da i ≤ n
O(1)+O(1)+O(n) = O(n+2) = O(n)
n·O(n) = O(n²)