Komplexität und Algorithmen

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw-mannheim.de, Raum 149B
Version: 1.0

Quelle: Die Folien sind teilweise inspiriert von oder basierend auf Lehrmaterial von Prof. Dr. Ritterbusch, Prof. Dr. Baumgart oder Prof. Dr. Albers.

Folien: https://delors.github.io/theo-algo-komplexitaet/folien.de.rst.html

https://delors.github.io/theo-algo-komplexitaet/folien.de.rst.html.pdf

Fehler melden:

https://github.com/Delors/delors.github.io/issues

1. Einführung - Landau'sche O-Notation

Berechnungskomplexität

Analyse des Aufwands zur Berechnung von Ergebnissen ist wichtig ...

- im Design,
- in der Auswahl
- und der Verwendung von Algorithmen.

Für relevante Algorithmen und Eingangsdaten können Vorhersagen getroffen werden:

- Um Zusammenhänge sind zwischen Eingangsdaten und Aufwand zu finden.
- Aufwand kann Rechenzeit, Speicherbedarf oder auch Komponentennutzung sein.

Der Rechenaufwand ist zentral und wird hier betrachtet, die Verfahren sind aber auch für weitere Ressourcen anwendbar.
Die Vorhersagen erfolgen über asymptotische Schätzungen

- mit Hilfe der Infinitesimalrechnung,
- durch Kategorisierung im Sinne des Wachstumsverhaltens,
- damit ist oft keine exakte Vorhersage möglich.

Unterschiedliche Systeme sind unterschiedlich schnell, relativ dazu wird es

interessant.

Im Folgenden geht es um:

- die Beschreibung des asymptotischen Wachstumsverhaltens
- die Analyse von iterativen Algorithmen
- die Analyse von rekursiv teilenden Algorithmen

Die Infinitesimalrechnung bezeichnet die Differenzial- und Integralrechnung. Es wird mit unendlich kleinen Größen gerechnet.

Exkursion: Dynamische Programmierung

Der folgende Abschnitt behandelt die dynamische Programmierung, um ein Problem effizient zu lösen. Er zeigt gleichzeitig wie die Wahl des Algorithmus und der Implementierung die Laufzeit dramatisch beeinflussen kann.

Übung

Berechnung der Fibonacci-Zahlen

Implementieren Sie eine **rekursive Funktion**, die die n-te Fibonacci-Zahl berechnet!

Hinweis

Die Fibonacci-Zahlen sind definiert durch die Rekursionsformel $F(n) = F(n-1) + F(n-2)$ mit den Anfangswerten $F(0) = 0$ und $F(1) = 1$.

Bis zu welchem n können Sie die Fibonacci-Zahlen in vernünftiger Zeit berechnen (d. h. < 10 Sekunden) ?

Berechnung der Fibonacci-Zahlen

Implementieren Sie eine **rekursive Funktion**, die die n-te Fibonacci-Zahl berechnet!

Hinweis

Die Fibonacci-Zahlen sind definiert durch die Rekursionsformel $F(n) = F(n-1) + F(n-2)$ mit den Anfangswerten $F(0) = 0$ und $F(1) = 1$.

Bis zu welchem n können Sie die Fibonacci-Zahlen in vernünftiger Zeit berechnen (d. h. < 10 Sekunden) ?

Technik der dynamischen Programmierung

Rekursiver Ansatz:
Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt.

Phänomen:
Mehrfachberechnungen von Lösungen

Methode: Speichern einmal berechneter Lösungen in einer Tabelle für spätere Zugriffe.

Beispiel: Berechnung der Fibonacci-Zahlen (rekursiv)

Definition

$F(0) = 0$

$F(1) = 1.$

$F(n) = F(n-1) + F(n-2)$

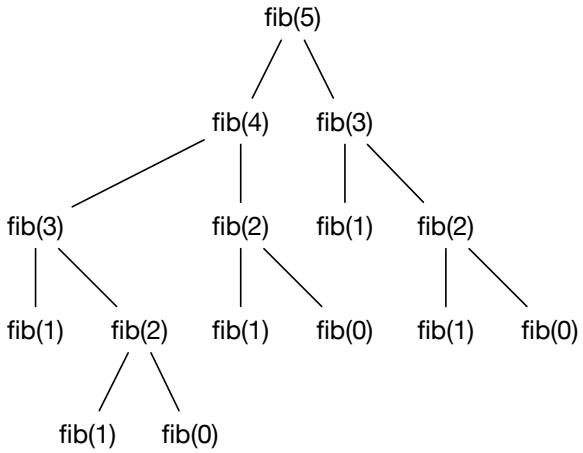
$F(n)$ als stehende Formel:

$$F(n) = \left[\frac{1}{\sqrt{5}} (1.618\ldots)^n \right]$$

Warnung

Die Berechnung der Fibonacci-Zahlen mit Hilfe einer naiven rekursiven Funktion ist sehr ineffizient.

Aufrufbaum



Vorgehen beim dynamischen Programmieren

1. Rekursive Beschreibung des Problems P
2. Bestimmung einer Menge T, die alle Teilprobleme von P enthält, auf die bei der Lösung von P – auch in tieferen Rekursionsstufen – zurückgegriffen wird.
3. Bestimmung einer Reihenfolge $T_0, \ldots, T_k$ der Probleme in T, so dass bei der Lösung von T_i nur auf Probleme T_j mit $j < i$ zurückgegriffen wird.
4. Sukzessive Berechnung und Speicherung von Lösungen für $T0, \ldots, Tk$.

Beispiel: Berechnung der Fibonacci-Zahlen mit dynamischer Programmierung

1. Rekursive Definition der Fibonacci-Zahlen nach gegebener Gleichung.
2. $T = f(0), \ldots, f(n-1)$
3. $T_i = f(i), i = 0, \ldots, n–1$
4. Berechnung von $fib(i)$ benötigt von den früheren Problemen nur die zwei letzten Teillösungen $fib(i–1)$ und $fib(i–2)$ für $i \geq 2$.

1

Lösung mit linearer Laufzeit und konstantem Speicherbedarf

```
1 procedure fib (n : integer) : integer
2     f_n_m2 := 0; f_n_m1 :=1
3     for k := 2 to n do
4         f_n := f_n_m1 + f_n_m2
5         f_n_m2 := f_n_m1
6         f_n_m1 := f_n

7     if n ≤ 1 then return n
8     else          return f_n
```

2

Lösung mit Memoisierung (🇺🇸 *Memoization*)

Berechne jeden Wert genau einmal, speichere ihn in einem Array F[0...n]:

```
 1 procedure fib (n : integer) : integer
 2     F[0] := 0; F[1] := 1;
 3     for i := 2 to n do
 4         F[i] := ∞ // Initialisierung
 5     return lookupfib(n)
 6
 7 procedure lookupfib (n : integer) : integer
 8     if F[n] = ∞ then
 9         F[n] := lookupfib(n-1) + lookupfib(n-2)
10     return F[n]
```

3

Übung

Fibonacci-Zahl effizient berechnen

Implementieren Sie den Pseudocode der ersten Lösung zur Berechnung der Fibonacci-Zahlen.

Bis zur welcher Fibonacci-Zahl können Sie die Berechnung nun durchführen?

Fibonacci-Zahl effizient berechnen

Implementieren Sie den Pseudocode der ersten Lösung zur Berechnung der Fibonacci-Zahlen.

Bis zur welcher Fibonacci-Zahl können Sie die Berechnung nun durchführen?

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Laufzeiten von Algorithmen

Folgen

Im Allgemeinen werden Laufzeiten oder Aufwände in Abhängigkeit von einer Eingangsgröße als Folge beschrieben:

Definition

Eine Folge (a_n) ist eine Abbildung, die jedem $n \in \mathbb{N}$ ein a_n zuweist.

- Folgenglieder

  Beispiel: $(a_n) : a_1 = 2, a_2 = 3, a_3 = 7, a_4 = 11, \ldots$

- Rekursive Definition

  Beispiel: $(c_n) : c_1 = 1, c_2 = 1, c_{n+2} = c_n + c_{n+1}$ $für$ $n \in (N)$

- Explizite Definition

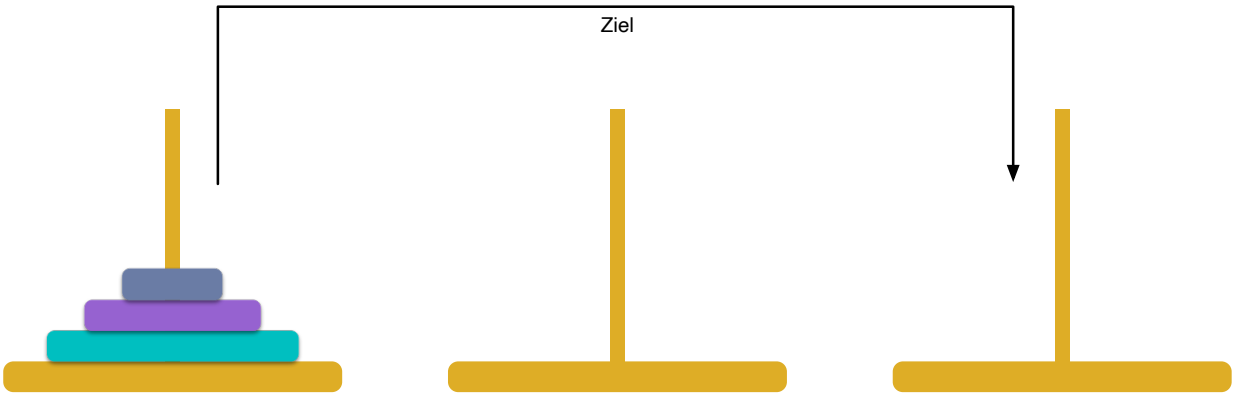
  Beispiel: $(b_n) : b_n = n^2$ für $n \in \mathbb{N}$

Eine rekursive Definition ist eine Definition, die sich auf sich selbst bezieht. Häufiger schwieriger zu analysieren. Die explizite Definition ist eine direkte Zuweisung und meist die beste Wahl.

Folgen und Laufzeiten

- Die explizite Definition von Laufzeiten ist zur Auswertung vorzuziehen.
- Die rekursive Definition tritt oft bei rekursiven Verfahren auf, und sollte dann in eine explizite Definition umgerechnet werden.

Berechnung der Anzahl der Schritte zum Lösen der Türme von Hanoi.



Türme von Hanoi mit 3 Scheiben.

Die Türme von Hanoi (ChatGPT)

Die Türme von Hanoi sind ein klassisches mathematisches Puzzle. Es besteht aus drei Stäben und einer bestimmten Anzahl von unterschiedlich großen Scheiben, die anfangs alle in absteigender Reihenfolge auf einem Stab gestapelt sind – der größte unten und der kleinste oben.

Das Ziel des Spiels ist es, alle Scheiben auf einen anderen Stab zu bewegen, wobei folgende Regeln gelten:

- Es darf immer nur eine Scheibe auf einmal bewegt werden.
- Eine größere Scheibe darf nie auf einer kleineren liegen.
- Alle Scheiben müssen auf den dritten Stab bewegt werden, indem sie über den mittleren Stab verschoben werden.

Laufzeit der Lösung der Türme von Hanoi

Für die Lösung sind für jeden Ring n die folgenden an Schritte erforderlich:

1. Alle $n-1$ kleineren Ringe über Ring n müssen mit a_{n-1} Schritten auf den Hilfsstab.
2. Der Ring n kommt auf den Zielstab mit einem Schritt.
3. Alle $n-1$ Ringe vom Hilfsstab müssen mit a_{n-1} Schritten auf den Zielstab.

Bei nur einem Ring ist $a_1 = 1$ und sonst $a_n = a_{n-1} + 1 + a_{n-1} = 2a_{n-1} + 1$.

Also: $a_1 = 1$, $a_2 = 2 \cdot 1 + 1 = 3$, $a_3 = 2 \cdot 3 + 1 = 7$, $a_4 = 2 \cdot 7 + 1 = 15$, ...

Damit liegt nahe, dass der Aufwand dem Zusammenhang $a_n = 2^n - 1$ entspricht.

Beweis durch vollständige Induktion

- Induktionsanfang $n = 1$: $a_1 = 2^n - 1 = 2^1 - 1 = 1$
- Induktionsvoraussetzung: $a_{n-1} = 2^{n-1} - 1$ und $a_n = 2a_{n-1} + 1$
- Induktionsschritt ($n - 1 \to n$):

$$
\begin{aligned}
a_n &= 2 \cdot (2^{n-1} - 1) + 1 \\
&= 2^n - 2 + 1 \\
&= 2^n - 1
\end{aligned}
$$

Damit ist die Vermutung bestätigt.

Eigenschaften von Folgen - Konvergenz

Definition

- Eine Folge (a_n) ist konvergent zum Grenzwert a, wenn es zu jeder Zahl $\varepsilon > 0$ ein $N \in \mathbb{N}$ gibt, so dass $|a_n - a| < \varepsilon$ für alle $n > N$ gilt.

Dies wird dann

$$a_n \xrightarrow{n\to\infty} a, a_n \to a \text{ oder } \lim_{n\to\infty} a_n = a$$

geschrieben.

- Eine Folge ist divergent, wenn es keinen Grenzwert gibt.

Eigenschaften von Folgen - Beispiel für Konvergenz

Betrachten wir die Folge (a_n) mit $a_n = \frac{(-1)^n}{n} + 2$, $n \in \mathbb{N}$:

Entwicklung der Folge:

$$a_1 = -1 + 2 = 1, a_2 = 0.5 + 2 = 2.5, a_3 = -0.33.. + 2 \approx 1.67, a_4 = 0.25 + 2 = 2.25, \ldots$$

Die Folge konvergiert zu 2, da für ein gegebenes $\varepsilon > 0$ ein N existiert so dass $|a_n - a| < \varepsilon$:

$$|a_n - a| = |\frac{(-1)^n}{n} + 2 - 2| = |\frac{(-1)^n}{n}| = \frac{1}{n} < \varepsilon$$

wenn $n > \frac{1}{\varepsilon}$ ist, also $a_n \to 2$ oder $lim_{n \to \infty} a_n = 2$

Konvergenz von Folgen - Rechenregeln

Satz

Die beiden Folgen (a_n) und (b_n) seien konvergent $a_n \to a$, $b_n \to b$ und $\lambda \in \mathbb{C}$, sowie $p, q \in \mathbb{N}$. Dann gilt:

$$\begin{aligned} lim_{n\to\infty} \lambda a_n &= \lambda a \\ lim_{n\to\infty} (a_n \pm b_n) &= a \pm b \\ lim_{n\to\infty} (a_n \cdot b_n) &= a \cdot b \\ lim_{n\to\infty} \tfrac{a_n}{b_n} &= \tfrac{a}{b}, \text{ für } b \neq 0, b_n \neq 0 \\ lim_{n\to\infty} a_n^{p/q} &= a^{p/q}, \text{wenn } a^{p/q} \text{ existiert} \end{aligned}$$

Konvergenz von Folgen - wichtige Grenzwerte

$$
\begin{aligned}
\lim_{n\to\infty} q^n &= 0 \quad \text{wenn } |q| < 1 \\
\lim_{n\to\infty} q^n &= \infty \quad \text{wenn } q > 1 \\
\lim_{n\to\infty} \tfrac{q^n}{n!} &= 0 \quad \text{für } q \in \mathbb{C} \\
\lim_{n\to\infty} \sqrt[n]{a} &= 1 \quad \text{wenn } a > 0 \\
\lim_{n\to\infty} \sqrt[n]{n} &= 1 \\
\lim_{n\to\infty} \sqrt[n]{n!} &= \infty
\end{aligned}
$$

Konvergenz von Folgen - Beispiel

Die Folge $a_n = \frac{n^2+1}{n^3}$ konvergiert gegen 0, da:

$$\lim_{n\to\infty} \frac{n^2+1}{n^3} = \lim_{n\to\infty} \frac{n^3(1/n+1/n^3)}{n^3} = \lim_{n\to\infty} \frac{(1/n+1/n^3)}{1} = 0$$

Die Folge konvergiert gegen 0, da der Zähler gegen 0 strebt ($\lim_{n\to\infty}(1/n) = 0$ und $\lim_{n\to\infty}(1/n^3) = 0$) und der Nenner konstant ist.

Die allgemeine Vorgehensweise ist es, die größte Potenz im Zähler und Nenner zu finden und dann diese auszuklammern. Im zweiten Schritt kürzen wir dann. In diesem Fall ist es n^3.

D. h. das Ziel ist es den Ausdruck so umzuformen, dass der Grenzwert direkt abgelesen werden kann. Dies ist inbesondere dann der Fall, wenn n nur noch im Nenner oder Zähler steht.

Analyse des asymptotischen Verhaltens

Wir möchten $f(x) = \frac{\ln(x)}{x^{2/3}}$ für $x \to \infty$ untersuchen.

Beobachtung

1. Der Zähler, $\ln(x)$, wächst gegen unendlich, aber sehr langsam im Vergleich zu Potenzfunktionen.
2. Der Nenner, $x^{2/3}$, wächst viel schneller als $\ln(x)$ für große x.

Es liegt somit ein unbestimmter Ausdruck vom Typ $\frac{\infty}{\infty}$ vor. Wir verwenden nun die Regel von L'Hôpital.

$$\lim_{x\to\infty} \frac{\ln(x)}{x^{2/3}} = \lim_{x\to\infty} \frac{\frac{d}{dx}(\ln(x))}{\frac{d}{dx}\left(x^{2/3}\right)} = \lim_{x\to\infty} \frac{\frac{1}{x}}{\frac{2}{3}x^{-1/3}}$$

Das vereinfacht sich zu:

$$= \lim_{x\to\infty} \frac{1}{x} \cdot \frac{3}{2}x^{1/3} = \lim_{x\to\infty} \frac{3}{2} \cdot \frac{1}{x^{2/3}} = 0$$

Die **Regel von L'Hôpital** ermöglicht es Grenzwerte von Ausdrücken des Typs $\frac{0}{0}$ oder $\frac{\infty}{\infty}$ zu berechnen. In diesem Fall nehmen wir die Ableitungen des Zählers und des Nenners.

Die Regel besagt:

Falls $\lim_{x\to a} \frac{f(x)}{g(x)}$ den unbestimmten Ausdruck $\frac{0}{0}$ oder $\frac{\infty}{\infty}$ ergibt, dann gilt:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)},$$

sofern der Grenzwert auf der rechten Seite existiert oder unendlich ist.

Übung - Konvergenz von einfachen Folgen

Erste Folge - zum Aufwärmen

Zeigen Sie, dass die Folge $a_n = \frac{n^2}{n^2+1}$ konvergiert und bestimmen Sie den Grenzwert.

Zweite Folge

Bestimmen Sie den Grenzwert der Folge, wenn er denn existiert: $b_n = \frac{1-n+n^2}{n(n+1)}$.

Erste Folge - zum Aufwärmen

Zeigen Sie, dass die Folge $a_n = \frac{n^2}{n^2+1}$ konvergiert und bestimmen Sie den Grenzwert.

Zweite Folge

Bestimmen Sie den Grenzwert der Folge, wenn er denn existiert: $b_n = \frac{1-n+n^2}{n(n+1)}$.

Übung - Konvergenz von Folgen

Hinweis

Die Binomischen Formeln sind ggf. hilfreich.

Folge mit Wurzel

Bestimmen Sie den Grenzwert $\lim_{n\to\infty} \sqrt{n^2+n} - n$.

Hier könnte die dritte Binomische Formel ($(a-b)(a+b) = a^2 - b^2$) hilfreich sein.

Folge mit mehreren Termen

Berechnen Sie den Grenzwert Folge $b_n = \frac{n^2-1}{n+3} - \frac{n^2+1}{n-1}$ falls er existiert.

Zwei Wurzeln

Bestimmen Sie den Grenzwert $\lim_{n\to\infty} \sqrt{n^2+1} - \sqrt{n^2+4n}$.

Um eine Potenz aus einer Wurzel zu bekommen, hilft ggf. das Wurzelgesetz $\sqrt{a} \cdot \sqrt{b} = \sqrt{a \cdot b}$.

Beispiel: $\sqrt{x^4+x^2} = \sqrt{x^4(1+1/x^2)} = \sqrt{x^4} \cdot \sqrt{(1+1/x^2)} = x^2 \cdot \sqrt{(1+1/x^2)}$.

Folge mit Wurzel

Bestimmen Sie den Grenzwert $\lim_{n\to\infty} \sqrt{n^2+n} - n$.

Hier könnte die dritte Binomische Formel ($(a-b)(a+b) = a^2 - b^2$) hilfreich sein.

Folge mit mehreren Termen

Berechnen Sie den Grenzwert Folge $b_n = \frac{n^2-1}{n+3} - \frac{n^2+1}{n-1}$ falls er existiert.

Zwei Wurzeln

Bestimmen Sie den Grenzwert $\lim_{n\to\infty} \sqrt{n^2+1} - \sqrt{n^2+4n}$.

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Landau-Notation

Asymptotische Abschätzung

Definition

Landau-Notation

Folgenden Mengen von Funktionen können asymptotisch von $g(n)$ ...

- nach oben abgeschätzt werden, $\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} | \lim_{x\to\infty} \frac{f(n)}{g(n)} < \infty\}$
- nach unten abgeschätzt werden, $\Omega(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} | \lim_{x\to\infty} \frac{f(n)}{g(n)} > 0\}$
- in gleicher Ordnung abgeschätzt werden,
  $\Theta(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} | \lim_{x\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}_{>0}\}$

Es gilt der folgende Zusammenhang für die Mengen $\mathcal{O}(g)$[1], $\Omega(g)$ und $\Theta(g)$:

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$$

[1] Im Folgenden verwenden wir einfach O statt $\mathcal{O}$.

Wenn eine Funktion f in der Menge $O(g)$ ist, dann wächst die Funktion g schneller als die Funktion f. Typischerweise ist der Grenzwert von $f(n)/g(n)$ für $n \to \infty$ in diesem Falle 0.

Die Verwendung der O-Notation zur Beschreibung der Komplexität von Algorithmen wurde von Donald E. Knuth eingeführt.

Alternative Schreibweisen

Insbesondere für die obere Abschätzung $O(g)$ gibt es eine alternative Schreibweise:

$$f(n) \in O(g(n)) \Leftrightarrow \exists c_0, n_0 \forall n : n > n_0 \Rightarrow f(n) \leq c_0 \cdot g(n)$$

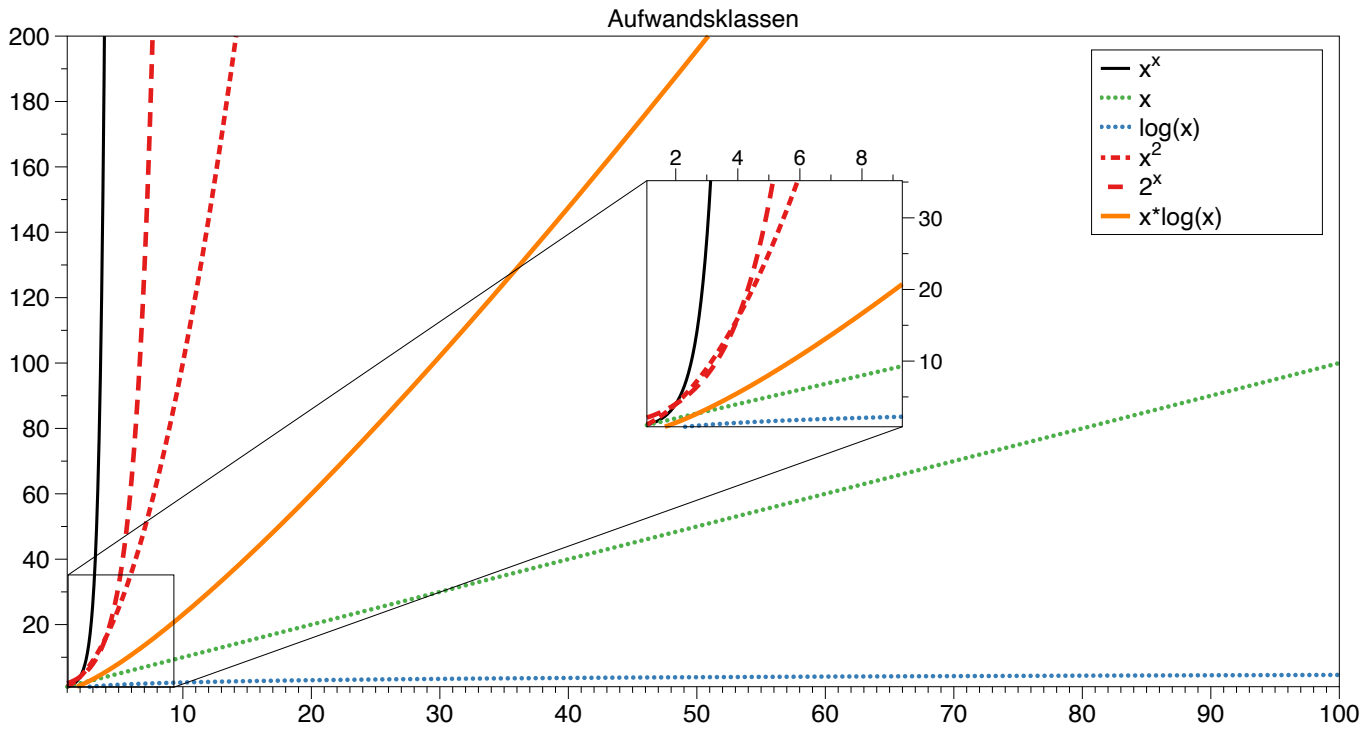
D. h. ab einem Wert n_0 liegt die Komplexität der Funktion f unter der c_0-fachen Komplexität der Funktion g.

Beispiel: $f(n) = 4n + 7 \in O(n)$

$$4n + 7 \leq c_0 \cdot n \Leftrightarrow n \cdot (4 - c_0) \leq -7$$

Wähle: $c_0 = 5$ und $n_0 = 7$ sowie $g(n) = n$.

Verstehen von Aufwandsklassen



Häufige Vergleichsfunktionen sind zum Beispiel Monome wie n^k für $k \in \mathbb{N}_0$.

Achtung bei asymptotischen Abschätzungen

Asymptotische Laufzeitabschätzungen können zu Missverständnissen führen:

1. Asymptotische Abschätzungen werden nur für steigende Problemgrößen genauer, für kleine Problemstellungen liegt oft eine ganz andere Situation vor.
2. Asymptotisch nach oben abschätzende Aussagen mit $O(g)$-Notation können die tatsächliche Laufzeit beliebig hoch überschätzen, auch wenn möglichst scharfe Abschätzungen erwünscht sein sollten, gibt es diese teilweise nicht in beliebiger Genauigkeit, oder sind nicht praktikabel.
3. Nur Abschätzungen von gleicher Ordnung $\Theta(g)$ können direkt verglichen werden, oder wenn zusätzlich zu $O(g)$ auch $\Omega(h)$ Abschätzungen vorliegen.

Gegenseitige asymptotische Abschätzung I

Bestimmen Sie welche Funktionen sich gegenseitig asymptotisch abschätzen:

$f_1(x) = \sqrt[3]{x},\ f_2(x) = e^{-1+ln\,x}, f_3(x) = \frac{x}{ln(x)+1}$.

D. h. berechnen Sie:

$$\lim_{x\to\infty} \frac{f_1(x)}{f_2(x)}, \lim_{x\to\infty} \frac{f_2(x)}{f_3(x)}, \text{ und ggf. } \lim_{x\to\infty} \frac{f_1(x)}{f_3(x)}$$

Denken Sie daran, dass die erste Ableitung von $f(x) = ln(x)$ die Funktion $f'(x) = \frac{1}{x}$ ist.

Gegenseitige asymptotische Abschätzung I

Bestimmen Sie welche Funktionen sich gegenseitig asymptotisch abschätzen:

$f_1(x) = \sqrt[3]{x},\ f_2(x) = e^{-1+ln\,x}, f_3(x) = \frac{x}{ln(x)+1}$.

D. h. berechnen Sie:

$$\lim_{x\to\infty} \frac{f_1(x)}{f_2(x)}, \lim_{x\to\infty} \frac{f_2(x)}{f_3(x)}, \text{ und ggf. } \lim_{x\to\infty} \frac{f_1(x)}{f_3(x)}$$

Übung - Asymptotische Abschätzungen

Gegenseitige asymptotische Abschätzung II

Vergleichen Sie: $f_1(x) = e^{2ln(x)+1}$ und $f_2(x) = \frac{x^3+1}{x}$.

Gegenseitige asymptotische Abschätzung III

Vergleichen Sie: $f_1(x) = 2^{1+2x}$ und $f_2(x) = 4^x + 2^x$.

Gegenseitige asymptotische Abschätzung II

Vergleichen Sie: $f_1(x) = e^{2ln(x)+1}$ und $f_2(x) = \frac{x^3+1}{x}$.

Gegenseitige asymptotische Abschätzung III

Vergleichen Sie: $f_1(x) = 2^{1+2x}$ und $f_2(x) = 4^x + 2^x$.

2. Algorithmische Komplexität

Algorithmen

Algorithmen sind Verfahren, die gegebene Ausprägungen von Problemen in endlich vielen Schritten lösen können.

Dabei muss jeder Schritt

- ausführbar und
- reproduzierbar sein.

Es gibt aber oft viele Methoden die Probleme zu lösen:

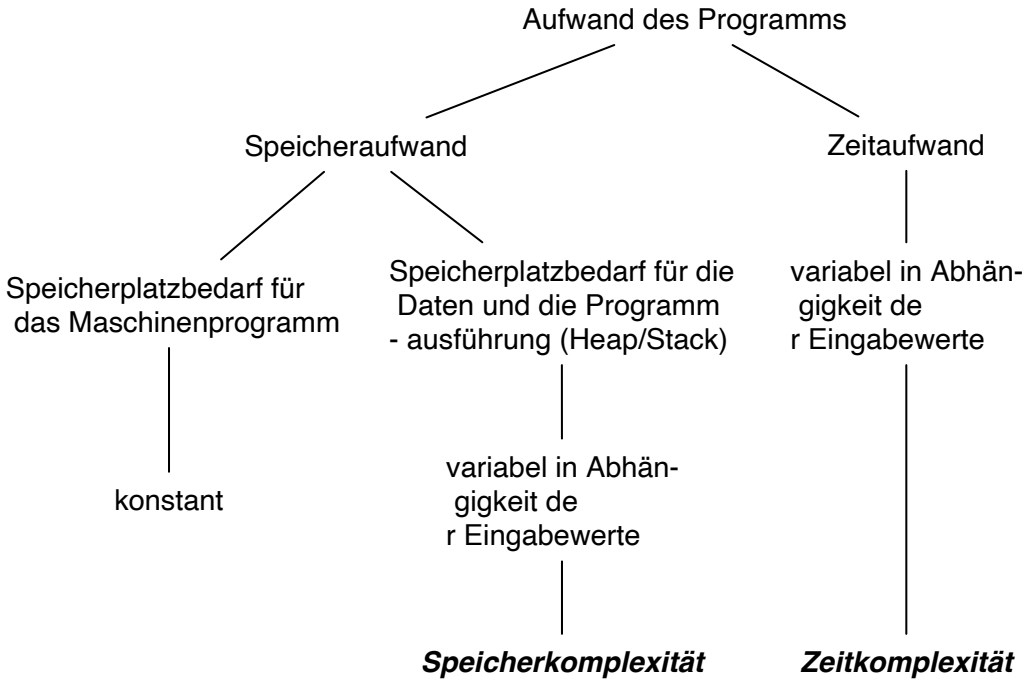
- Daher ist es wichtig, Eigenschaften von Algorithmen zu analysieren!
- Insbesondere z.B.
- Zeitaufwand und
- Speicherbedarf
- in Abhängigkeit von der Problemgröße.

Problemumfang (Problemgröße) n

Konkrete Beispiele für Problemgrößen:

- Konkreter Wert von n: $f(n)$
- Stellenanzahl des Eingabewertes (der Eingabewerte) → $f(z_1 z_2 \dots z_n)(z_i \in 0, \dots, 9)$
- Anzahl der Eingabewerte: $f(x_1, x_2, \dots, x_n)$

Aufwand - Übersicht



Algorithmen - Zeitaufwand

Tatsächlicher Zeitaufwand hängt vom ausführenden Rechnersystem ab.

- Beeindruckende Entwicklung der Rechentechnik.
- Größere Probleme können gelöst werden.
- **Langsamere Algorithmen bleiben langsamer auch auf schnellen Systemen.**

Eine möglichst sinnvolle Annahme eines Rechnersystems gesucht:

- Von-Neumann System
- *mit einer Recheneinheit*
- genaue Geschwindigkeit nicht relevant.

Bemerkung

Wir unterscheiden:

- Komplexität eines Algorithmus

  Asymptotischer Aufwand ($n \to \infty$) der Implementierung des Algorithmus.

- Komplexität eines Problems

  Minimale Komplexität eines Algorithmus zur Lösung des Problems Algorithmus.

Die Komplexität eines Problems zu bestimmen ist oft ausgesprochen schwierig, da man hierfür den besten Algorithmus kennen muss. Es stellt sich dann weiterhin die Frage wie man beweist, dass der beste Algorithmus vorliegt.

Bei vielen Komplexitätsanalysen steht die Zeitkomplexität im Vordergrund.

Die Zeitkomplexität misst nicht konkrete Ausführungszeiten (z. B. 1456 ms), da die Ausführungszeit von sehr vielen Randbedingungen abhängig ist, die direkt nichts mit demAlgorithmus zu tun haben, z. B.:

- Prozessortyp und Taktfrequenz
- Größe des Hauptspeichers
- Zugriffszeiten der Peripheriegeräte
- Betriebssystem → wird z. B. ein virtueller Speicher unterstützt
- Compiler- oder Interpreter-Version
- Systemlast zum Zeitpunkt der Ausführung

Wichtige Komplexitätsklassen

Klasse	Eigenschaft
$O(1)$	Die Rechenzeit ist unabhängig von der Problemgröße
$O(log\, n)$	Die Rechenzeit wächst logarithmisch (zur Basis 2) mit der Problemgröße
$O(n)$	Die Rechenzeit wächst linear mit der Problemgröße
$O(n \cdot log\, n)$	Die Rechenzeit wächst linear logarithmisch mit der Problemgröße
$O(n^2)$	Die Rechenzeit wächst quadratisch mit der Problemgröße
$O(n^3)$	Die Rechenzeit wächst kubisch mit der Problemgröße
$O(2^n)$	Die Rechenzeit wächst exponentiell (zur Basis 2) mit der Problemgröße
$O(n!)$	Die Rechenzeit wächst entsprechend der Fakultätsfunktion mit der Problemgröße

Komplexität und bekannte Algorithmen/Probleme

$O(1)$

- Liegt typischerweise dann vor, wenn das Programm nur einmal linear durchlaufen wird.
- Es liegt keine Abhängigkeit von der Problemgröße vor, d. h. beispielsweise keine Schleifen in Abhängigkeit von n.
- Beispiel:

  Die Position eines Datensatzes auf einem Datenträger kann mit konstanten Aufwand berechnet werden.

$O(log\, n)$

- Beispiel:

  Binäre Suche; d. h. in einem sortierten Array mit n Zahlen eine Zahl suchen.

$O(n)$

- Beispiel:

  Invertieren eines Bildes oder sequentielle Suche in einem unsortierten Array.

$O(n \cdot log\, n)$

- Beispiel:

  Bessere Sortierverfahren wie z. B. Quicksort.

$O(n^2)$

- Häufig bei zwei ineinander geschachtelten Schleifen.
- Beispiel:

  Einfache Sortierverfahren wie z. B. Bubble-Sort oder die Matrixaddition.

$O(n^3)$

- Häufig bei drei ineinander geschachtelten Schleifen.
- Beispiel:

  Die Matrixmultiplikation.

$M(m,t)$ ist eine Matrix mit m Zeilen und t Spalten.

$C(m,t) = A(m,n) \cdot B(n,t)$ mit

$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j} \qquad i = 1,\ldots,m \qquad j = 1,\ldots,t$

$O(2^n)$

- Typischerweise der Fall, wenn für eine Menge mit n Elementen alle Teilmengen berechnet und verarbeitet werden müssen.
- Beispiel:

Rucksackproblem (🇺🇸 *Knapsack Problem*)

Ein Rucksack besitzt eine maximale Tragfähigkeit und n Gegenstände unterschiedlichen Gewichts liegen vor, deren Gesamtgewicht über der Tragfähigkeit des Rucksacks liegt. Ziel ist es jetzt eine Teilmenge von Gegenständen zu finden, so dass der Rucksack optimal gefüllt wird.

$O(n!)$

- Typischerweise der Fall, wenn für eine Menge von n Elementen alle Permutationen dieser Elemente zu berechnen und zu verarbeiten sind.

- Beispiel:

Problem des Handlungsreisenden (🇺🇸 *Traveling Salesman Problem (TSP)*)

Gegeben sind n Städte, die alle durch Straßen direkt miteinander verbunden sind und für jede Direktverbindung ist deren Länge bekannt.

Gesucht ist die kürzeste Rundreise, bei der jede Stadt genau einmal besucht wird

Approximation von Laufzeiten

Sei die Problemgröße $n = 128$:

Klasse	Laufzeit
$O(log_2\, n)$	$1,75\, ns$
$O(n)$	$32\, ns$
$O(n \cdot log_2\, n)$	$224\, ns$
$O(n^2)$	$4,096\, \mu s$
$O(n^3)$	$524,288\, \mu s$
$O(2^n)$	$2,70 \cdot 10^{21}\, a$
$O(3^n)$	$9,35 \cdot 10^{43}\, a$
$O(n!)$	$3,06 \cdot 10^{198}\, a$

Bemerkung

Für die Approximation sei ein Rechner mit 4 GHz Taktrate angenommen und ein Rechenschritt soll einen Takt benötigen.

Verwendete Abkürzungen:

- $1ns = 10^{-9}s$ → Nanosekunde
- $1\mu s = 10^{-6}s$ → Mikrosekunde
- $1ms = 10^{-3}s$ → Millisekunde
- $1h = 3600s$ → Stunde
- $1d = 86400s$ → Tag
- $1a$ → Jahr

Dies zeigt, dass Algorithmen mit einer Komplexität von $O(n^3)$ oder höher für große bzw. nicht-triviale Problemgrößen nicht praktikabel sind.

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Iterative Algorithmen

Elementare Kosten als Approximation

Elementare Operation	**Anzahl der Rechenschritte**
elementare Arithmetik: + ,- , * , /, etc.	1
elementare logische Operationen: &&, ||, !, etc.	1
Ein- und Ausgabe	1
Wertzuweisung	1
return, break, continue	1
Kontrollstrukturen	Anzahl der Rechenschritte
Methodenaufruf	1 + Komplexität der Methode
Fallunterscheidung	Komplexität des logischen Ausdrucks + Maximum der Komplexität der Rechenschritte der Zweige

Beispielanalyse von `ist_primzahl`

```
def ist_primzahl(n):
    prim = True                 # Wertzuweisung:            1
    i = 2                       # Wertzuweisung:            1
    if n < 2:                   # Vergleich:                1
        prim = False            # Wertzuweisung:            1
    else:                       # Durchläufe:               n-2 * (
        while prim and i < n:   #   Vergleiche, und:            3
            if n % i == 0:      #   modulo, Vergleich:          2
                prim = False    #       Wertzuweisung:              1
            i += 1              #   Inkrement:                  1
                                #                             )
                                # letzte Bedingungsprüfung  3
    return prim                 # Befehl:                   1
```

Im schlechtesten Fall, d. h. es gilt $i == n$ nach der while-Schleife, werden $7 + (n-2) \cdot 7 = 7 \cdot n - 7$ Rechenschritte benötigt. Die Anzahl der Rechenschritte hängt somit linear vom Eingabewert n ab.

Beachte, dass in keinem Falle alle Instruktionen ausgeführt werden.

Hinweis

Dies kein effizienter Algorithmus zum Feststellen ob eine Zahl Primzahl ist.

Übung

Bestimmung der asymptotischen Laufzeit eines Algorithmus

Die Funktion $p(n)$ hat die Laufzeit $T_p(n) = c_p \cdot n^2$ und $q(n)$ die Laufzeit $T_q(n) = c_q \cdot log(n)$.

```
1 Algorithmus COMPUTE(n)
2 p(n);
3 for j = 1...n do
4     for k = 1...j do
5         q(n);
6     end
7 end
```

Bestimmen Sie die asymptotische Laufzeit des Algorithmus in Abhängigkeit von n durch zeilenweise Analyse.

Bestimmung der asymptotischen Laufzeit eines Algorithmus

Die Funktion $p(n)$ hat die Laufzeit $T_p(n) = c_p \cdot n^2$ und $q(n)$ die Laufzeit $T_q(n) = c_q \cdot log(n)$.

```
1 Algorithmus COMPUTE(n)
2 p(n);
3 for j = 1...n do
4     for k = 1...j do
5         q(n);
6     end
7 end
```

Bestimmen Sie die asymptotische Laufzeit des Algorithmus in Abhängigkeit von n durch zeilenweise Analyse.

Übung

„Naive“ Power Funktion

Bestimmen Sie die algorithmische asymptotische Komplexität des folgenden Algorithmus durch Analyse jeder einzelnen Zeile. Jede Zeile kann für sich mit konstantem Zeitaufwand abgeschätzt werden. Bestimmen Sie die Laufzeitkomplexität für den schlimmstmöglichen Fall in Abhängigkeit von k für eine nicht-negative Ganzzahl n mit k Bits.

(Beispiel: die Zahl $n = 7_d$ benötigt drei Bits $n = 111_b$, die Zahl $4d$ benötigt zwar auch drei Bits 100_b aber dennoch weniger Rechenschritte.).

```
1 Algorithmus Power(x,n)
2     r = 1
3     for i = 1...n do
4         r = r * x
5     return r
```

„Naive“ Power Funktion

Bestimmen Sie die algorithmische asymptotische Komplexität des folgenden Algorithmus durch Analyse jeder einzelnen Zeile. Jede Zeile kann für sich mit konstantem Zeitaufwand abgeschätzt werden. Bestimmen Sie die Laufzeitkomplexität für den schlimmstmöglichen Fall in Abhängigkeit von k für eine nicht-negative Ganzzahl n mit k Bits.

(Beispiel: die Zahl $n = 7_d$ benötigt drei Bits $n = 111_b$, die Zahl $4d$ benötigt zwar auch drei Bits 100_b aber dennoch weniger Rechenschritte.).

```
1 Algorithmus Power(x,n)
2     r = 1
3     for i = 1...n do
4         r = r * x
5     return r
```

Effizientere Power Funktion

Bestimmen Sie die algorithmische asymptotische Komplexität des folgenden Algorithmus durch Analyse jeder einzelnen Zeile. Jede Zeile kann für sich mit konstantem Zeitaufwand abgeschätzt werden. Bestimmen Sie die Laufzeitkomplexität mit Indikator t_i für gesetzte Bits in n für den schlimmstmöglichen Fall in Abhängigkeit von k für eine nicht-negative Ganzzahl n mit k Bits.

(D. h. $t_i = 1$, wenn der i-te Bit von n gesetzt ist, sonst ist $t_i = 0$; sei $n = 5_d = 101_b$ dann ist $t_1 = 1, t_2 = 0, t_3 = 1$).

```
1  Algorithmus BinPower(x,n)
2      r = 1
3      while n > 0 do
4          if n mod 2== 1 then
5              r = r * x
6              n = (n-1)/2
7          else
8              n = n/2
9          x = x *x
10     return r
```

Effizientere Power Funktion

Bestimmen Sie die algorithmische asymptotische Komplexität des folgenden Algorithmus durch Analyse jeder einzelnen Zeile. Jede Zeile kann für sich mit konstantem Zeitaufwand abgeschätzt werden. Bestimmen Sie die Laufzeitkomplexität mit Indikator t_i für gesetzte Bits in n für den schlimmstmöglichen Fall in Abhängigkeit von k für eine nicht-negative Ganzzahl n mit k Bits.

(D. h. $t_i = 1$, wenn der i-te Bit von n gesetzt ist, sonst ist $t_i = 0$; sei $n = 5_d = 101_b$ dann ist $t_1 = 1, t_2 = 0, t_3 = 1$).

```
1  Algorithmus BinPower(x,n)
2      r = 1
3      while n > 0 do
4          if n mod 2== 1 then
5              r = r * x
6              n = (n-1)/2
7          else
8              n = n/2
9          x = x *x
10     return r
```