Komplexität und Algorithmen

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 2.0.3

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
Kontrollfragen: https://delors.github.io/theo-algo-komplexitaet/kontrollfragen.de.rst.html
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 häufig 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.

Entwurf von Algorithmen: 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

1.1. 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

$F(0) = 0$ $F(1) = 1$ $F(n) = F(n-1) + F(n-2)$

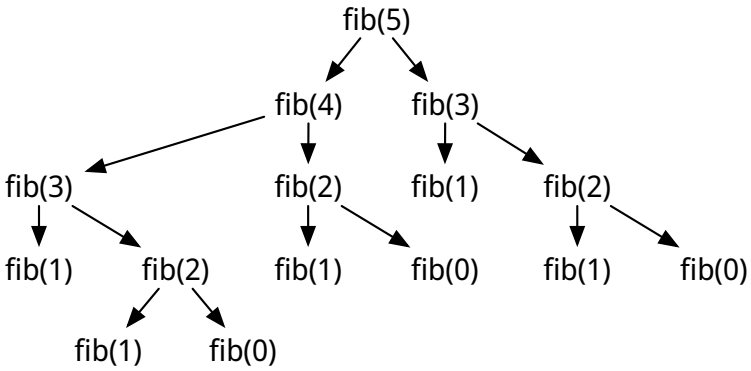
Warnung

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

$F(n)$ als stehende Formel:

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

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 $T_0, \ldots, T_k$.

Beispiel: Berechnung der Fibonacci-Zahlen mit dynamischer Programmierung

Rekursive Definition der Fibonacci-Zahlen nach gegebener Gleichung:

1. $T = f(0), \ldots, f(n-1)$
2. $T_i = f(i), i = 0, \ldots, n–1$
3. Berechnung von $fib(i)$ benötigt von den früheren Problemen nur die zwei letzten Teillösungen $fib(i–1)$ und $fib(i–2)$ für $i \geq 2$.

Lösung mit linearer Laufzeit und konstantem Speicherbedarf

```
1  procedure fib (n : integer) : integer
2      f_n_m2 := 0
3      f_n_m1 := 1
4      for k := 2 to n do
5          f_n := f_n_m1 + f_n_m2
6          f_n_m2 := f_n_m1
7          f_n_m1 := f_n
8      if n ≤ 1 then return n
9      else          return f_n
```

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

Übung

1.2. Fibonacci-Zahl effizient berechnen

Implementieren Sie den Pseudocode der ersten Lösung zur Berechnung der Fibonacci-Zahlen. Verwenden Sie `BigInteger`s.

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

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.

- Definition über 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 $n \in \mathbb{N}$

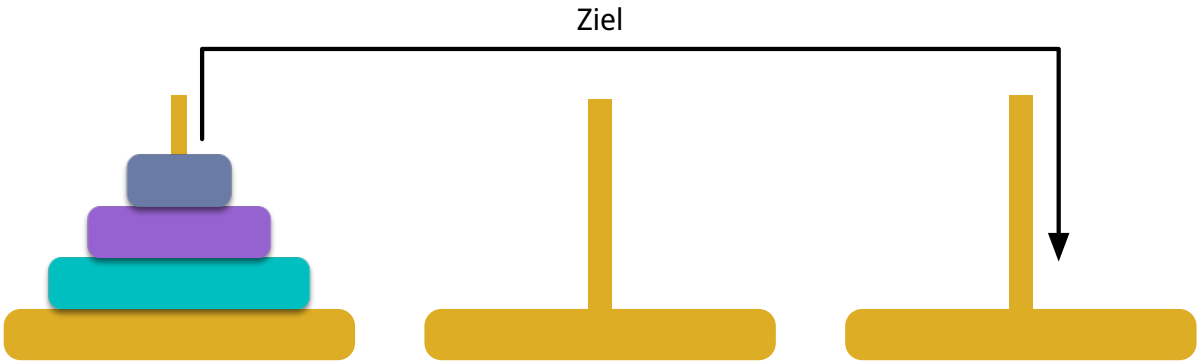
Eine rekursive Definition ist eine Definition, die sich auf sich selbst bezieht. Eine solche Definition ist häufiger schwieriger zu analysieren.

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 a_n 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 $(1, 3, 7, 15, \ldots)$ dem Zusammenhang $a_n = 2^n - 1$ entspricht.

Beweis

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. ∎

Konvergenz von Folgen

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

Beispiel

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

Entwicklung der Folge

$a_1 = -1 + 2 = 1$

$a_2 = 0.5 + 2 = 2.5$

$a_3 = -0.33.. + 2 \approx 1.67$

$a_4 = 0.25 + 2 = 2.25$

...

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.

D. h. $a_n \to 2$ 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

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

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 zur 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.

Anwendung von L'Hôpital

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

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

1.3. Erste Folge - zum Aufwärmen

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

1.4. 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.

1.5. 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.

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

Beispiel

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

1.6. 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.

1.7. Zwei Wurzeln

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

Landau-Notation

Asymptotische Abschätzung

Definition

Landau-Notation

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

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

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

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

Wenn eine Funktion f in der Menge $O(g)$ (d. h. $f \in O(g)$) ist, dann wächst die Funktion g mindestens genauso schnell wie die Funktion f. Wächst $g(n)$ asymptotisch schneller, dann ist $f(n)/g(n)$ für $n \to \infty$ in diesem Falle 0; wachsen beide gleich schnell, dann ist es eine Konstante c.

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

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

Alternative Schreibweisen

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

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

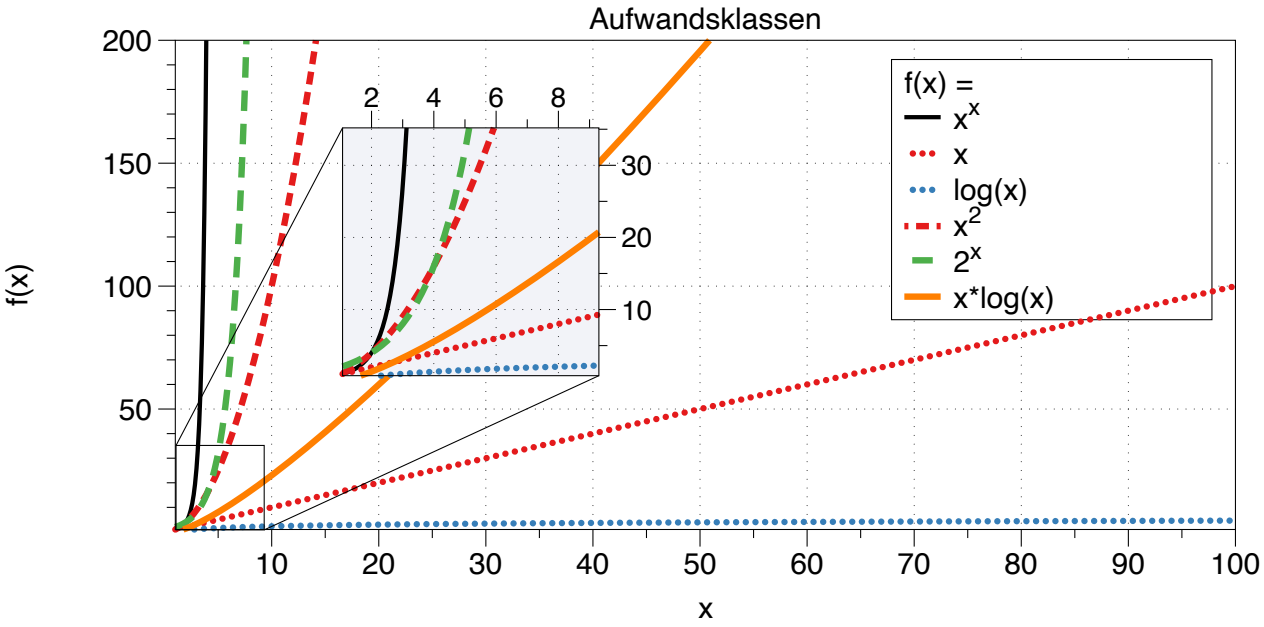
D. h. ab einem Wert n_0 liegt die Funktion f unter dem c_0-fachen der Funktion g.

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

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

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

Verstehen von Aufwandsklassen



Bemerkung

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.

Übung

1.8. 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.

Übung - Asymptotische Abschätzungen

1.9. Gegenseitige asymptotische Abschätzung II

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

1.10. 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:

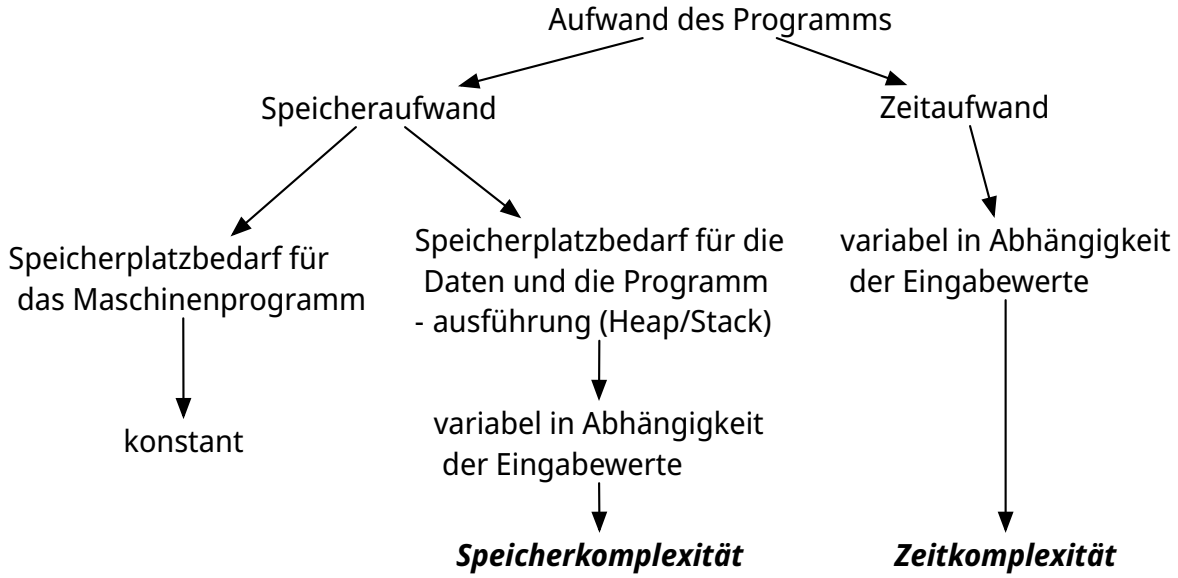
- Daher ist es wichtig, Eigenschaften von Algorithmen zu analysieren!
- Insbesondere z.B.
- Zeitaufwand und
- Speicherbedarf
- in Abhängigkeit von der Problemgröße.

Problemumfang (Problemgröße) n

Konkrete Beispiele für Problemgrößen:

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

Aufwand - Übersicht



Komplexität

Wir unterscheiden:

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

Komplexität eines Problems:
Minimale Komplexität (irgend) eines Algorithmus zur Lösung des Problems.

Algorithmen - Zeitaufwand

Tatsächlicher Zeitaufwand hängt vom ausführenden Rechnersystem ab.

- Beeindruckende Entwicklung der Rechentechnik in den letzten Jahrzehnten
- Größere Probleme können gelöst werden

⚠ Warnung

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

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 dem Algorithmus 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 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 (hier 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 oder die Matrixaddition.

$O(n^3)$

- Häufig bei drei ineinander geschachtelten Schleifen.

Beispiel

Die (naive) 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

Das Rucksackproblem (*Knapsack Problem*)

Ein Rucksack besitzt eine maximale Tragfähigkeit und n Gegenstände unterschiedlichen Gewichts und Wertes 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 in Hinblick auf den Gesamtwert 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

Das 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

◎ Bemerkung

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

Verwendete Abkürzungen:

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

Sei die Problemgröße $n = 128$:

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

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

Iterative Algorithmen

Elementare Kosten als Approximation

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
Schleife	Annahme: m Durchläufe: Komplexität der Initialisierung + m mal die Komplexität des Schleifenkörpers + Komplexität aller Schleifenfortschaltungen

Beispiel Primzahltest: Analyse mit elementaren Kosten

```
1  def ist_primzahl(n):
2      prim = True                  # Wertzuweisung:            1
3      i = 2                        # Wertzuweisung:            1
4      if n < 2:                    # Vergleich:                1
5          prim = False             # Wertzuweisung:            1
6      else:                        # Durchläufe:               n-2 * (
7          while prim and i < n:    #   Vergleiche, und:            3
8              if n % i == 0:       #   modulo, Vergleich:          2
9                  prim = False     #       Wertzuweisung:              1
10             i += 1               #   Inkrement:                  1
11                                  #                             )
12                                  # letzte Bedingungsprüfung  3
13     return prim                  # Befehl:                   1
```

Im schlechtesten Fall, d. h. $i \geq 2$ und 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 ist kein effizienter Algorithmus zum Feststellen ob eine Zahl Primzahl ist.
Dieser Algorithmus ist nur zu Demonstrationszwecken gedacht.

Insertion-Sort: Analyse mit abstrahierten Kosten

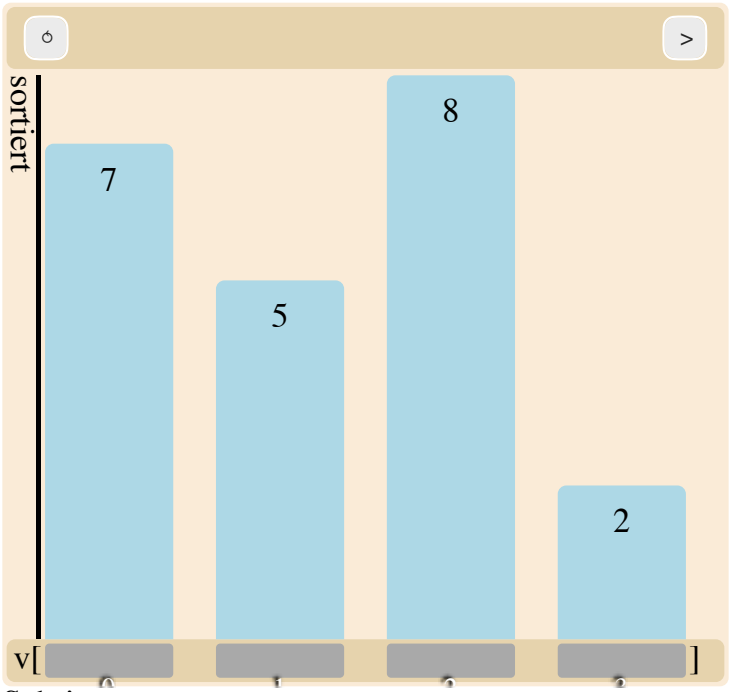
Sortieralgorithmen

Hintergrund

Sortierung basiert (meist) auf paarweisen Vergleichen.

- Vergleichsoperatoren für numerische Werte (`int`, `float`,...) klar
- Andere Datentypen / Klassen benötigen definierten Vergleichsoperator

Vergleichbar zum Ziehen von Karten: die neue Karte wird an der richtigen Stelle eingeschoben.



Schritt:

...

i	sortiert	key	unsortiert (Ausgangssituation)
0			5 3 1 4 2
1	5	3	1 4 2
2	3 5	1	4 2
3	1 3 5	4	2
4	1 3 4 5	2	
	1 2 3 4 5		

```python
def insertion_sort(A):
  for i in range(1, len(A)):
    key = A[i]
    j = i - 1
    while j >= 0 and A[j] > key:
      A[j + 1] = A[j]
      j = j - 1
    A[j + 1] = key
```

Beispiel Insertion-Sort: Detailanalyse

Algorithmus: Insertion-Sort(A, n) [Pseudocode - Index startet bei 1]	Zeit	Anzahl
1: `for i = 2...n do`	c1	n
2: `  key = A[i]`	c2	$n-1$
3: `  j = i - 1`	c3	$n-1$
4: `  while j > 0 and A[j] > key do`	c4	$\sum_{i=2}^{n} t_i$
5: `  A[j + 1] = A[j]`	c5	$\sum_{i=2}^{n} (t_i - 1)$
6: `  j = j - 1`	c6	$\sum_{i=2}^{n} (t_i - 1)$
7: `  A[j + 1] = key`	c7	$n-1$

- c_x sind die konstanten Kosten für die jeweilige Operation. Wir abstrahieren diese als $c = max(c_1, \dots c_7)$.
- t_i ist die Anzahl der Schritte, die für das Einsortieren der i-ten Karte benötigt wird. Dies hängt davon ab, wie die Liste vorliegt.

Abschätzung der Laufzeit $T(n)$ nach oben:

$$T(n) \leq c \cdot \left(n + 3 \cdot (n-1) + \sum_{i=2}^{n} t_i + 2 \cdot \sum_{i=2}^{n} (t_i - 1) \right)$$

$$T(n) \leq c \cdot \left(n + 3 \cdot (n-1) + \sum_{i=2}^{n} t_i + 2 \cdot (\sum_{i=2}^{n} t_i - \sum_{i=2}^{n} 1)) \right)$$

$$= c \cdot \left(4n - 3 + 3 \cdot \sum_{i=2}^{n} t_i - 2 \cdot (n-1) \right)$$

$$= c \cdot \left(2n - 1 + 3 \cdot \sum_{i=2}^{n} t_i \right)$$

Jetzt können drei Fälle unterschieden werden:

- die Liste ist bereits sortiert, d. h. $t_i = 1$
- die Liste ist umgekehrt sortiert, d. h. $t_i = i$
- die Liste ist zufällig sortiert, d. h. $t_i = \frac{i+1}{2}$

Im schlimmsten Fall, d. h. die Liste ist umgekehrt sortiert, ergibt sich:

$$T(n) \leq c \cdot \left(2n - 1 + 3 \cdot \sum_{i=2}^{n} i \right)$$

und nach Anwendung der Summenformel für die natürlichen Zahlen:

$$= c \cdot \left(\frac{3}{2} n^2 + \frac{7}{2} n - 4 \right)$$

Im besten Fall, d. h. die Liste ist bereits sortiert, ergibt sich:

$$T(n) \leq c \cdot \left(2n - 1 + 3 \cdot \sum_{i=2}^{n} 1 \right)$$
$$= c \cdot (5n - 4)$$

Bemerkung

Insertion-Sort hat die gleiche Performance auf Arrays und (doppelt) verketteten Listen

---

※ Hinweis

In Zeile 1 ist die Anzahl n, da $n - 1$ mal hochgezählt wird und dann noch ein Test erfolgt, der fehlschlägt und die Schleife beendet.

Beispiel Insertion-Sort: Ergebnisse

In Hinblick auf den Zeitaufwand gilt:

$$\begin{aligned} T_{worst}(n) &\in \Theta(n^2) \\ T_{average}(n) &\in \Theta(n^2) \\ T_{best}(n) &\in \Theta(n) \end{aligned}$$

Der Insertion-Sort-Algorithmus hat eine quadratische Komplexität, d. h. die Laufzeit wächst quadratisch mit der Problemgröße. Er hat die Komplexität $O(n^2)$.

Übung

2.1. 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

2.2. „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. Die Eingabe ist eine nicht-negative Ganzzahl n mit k Bits. Bestimmen Sie die Laufzeitkomplexität für den schlimmstmöglichen Fall in Abhängigkeit von k!

(Beispiel: die Zahl $n = 7_d$ benötigt drei Bits $n = 111_b$, die Zahl 4_d 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
```

Übung

2.3. Effizientere Power Funktion

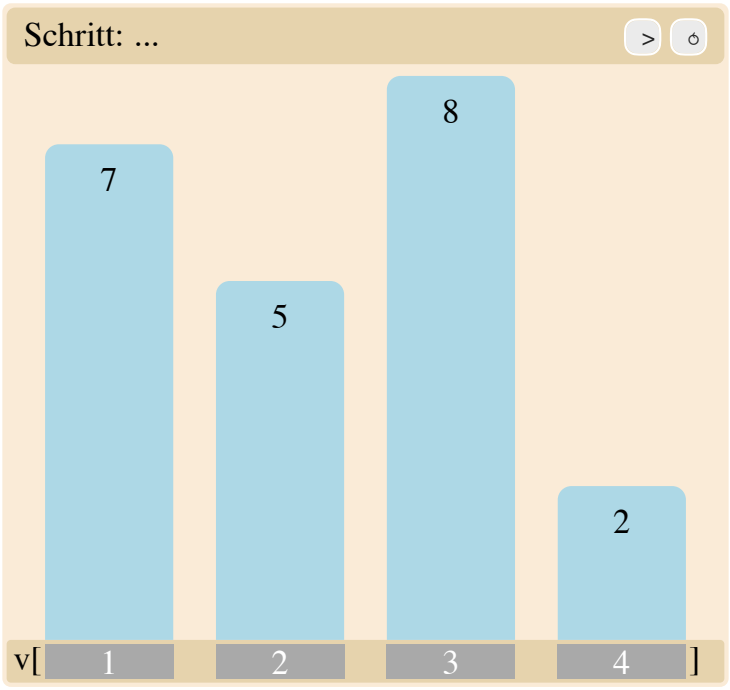
Bestimmen Sie die algo. asymptotische Komplexität durch Analyse jeder einzelnen Zeile. Jede Zeile kann mit konstantem Zeitaufwand abgeschätzt werden. Bestimmen Sie die Laufzeitkomplexität mit Indikator t_i für gesetzte Bits in n für den schlimmstmöglichen Fall - in Abhängigkeit von k für eine nicht-negative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

Übung - Bubble-Sort

Der Algorithmus "Bubble-Sort" sortiert eine Liste von Zahlen in aufsteigender Reihenfolge. Der Algorithmus vergleicht benachbarte Elemente und vertauscht sie, wenn sie in der falschen Reihenfolge sind. Dieser Vorgang wird wiederholt, bis die Liste vollständig sortiert ist.



2.4. Implementieren Sie Bubble-Sort

1. Implementieren Sie den Bubble-Sort Algorithmus in einer Programmiersprache Ihrer Wahl und analysieren Sie die Laufzeitkomplexität des Algorithmus. Der Algorithmus soll ein Array/Liste von Ganzzahlen als Eingabe nehmen und sortiert zurückgeben. Dabei soll das Array/die Liste *in-place* sortiert werden, d. h. ohne zusätzliche Speicherplatznutzung.
2. Bestimmen Sie die Komplexität des Algorithmus in Abhängigkeit von der Anzahl der Elemente (n) im Array/der Liste.

Rucksackproblem (*Knapsack Problem*)

Definition

Das Rucksackproblem

Gegeben seien Wertepaare $\{(g_1, w_1), \ldots, (g_m, w_m)\}$ mit $g_i, w_i \in \mathbb{N}$, die das Gewicht g_i und den Wert w_i eines Teils i darstellen. Gesucht sind die Anzahlen $a_i \in \mathbb{N}_0$ der jeweiligen Teile, so dass

$$\sum_{i=1}^{m} a_i g_i \leq n \quad \text{und} \quad \sum_{i=1}^{m} a_i w_i \quad \text{maximal wird}$$

also für gegebene maximale Last n des Rucksacks der aufsummierte Wert maximal wird.

Beispiel

Verfügbare Objekte $((Gewicht, Wert))$: $A = \{(1,1), (3,4), (5,8), (2,3)\}$.

- Bei einer maximalen Traglast von 5 ist der maximale Wert 8 ($1\times$ Objekt 3).
- Gesucht ist die maximale Wertsumme bei einer maximalen Traglast von 13.

1. Versuch: bei Einhaltung der Traglast ($n = 13$):

   $\overset{\#}{1} \cdot \overset{g}{1} + \overset{\#}{4} \cdot \overset{g}{3} = 13 \leq 13 \quad \Rightarrow \quad \overset{\#}{1} \cdot \overset{w}{1} + \overset{\#}{4} \cdot \overset{w}{4} = 17$ (Wert)

2. Versuch: bei Einhaltung der Traglast ($n = 13$):

   $1 \cdot 1 + 2 \cdot 5 + 1 \cdot 2 = 13 \leq 13 \quad \Rightarrow \quad 1 \cdot 1 + 2 \cdot 8 + 1 \cdot 3 = 20$ (Wert)

Rucksackproblem - rekursive Lösung

```
1  gW = [ (1, 1), (3, 4), (5, 8), (2, 3) ] # [(Gewicht, Wert)...]
2
3  def bestWertRekursiv(n): # n = max. Traglast
4      best = 0
5      for i in range(len(gW)):
6          (gewt,wert) = gW[i]
7          if n >= gewt:
8              test = wert + bestWertRekursiv(n - gewt)
9              if test > best:
10                 best = test
11     return best
12
13 print(bestWertRekursiv(5)) # max. Traglast ist hier zu Beginn n = 5
```

Für die Bestimmung der Komplexität nehmen wir jetzt die häufigste Aktion her; hier die Additionen.

Bei der Rekursion ergibt sich (m = Anzahl der verschiedenen Objekte):

- Im schlimmsten Fall sind alle $g_i = 1$ (d. h. die Gewichte).
- Pro Aufruf m weitere Aufrufe.

  (D. h. auf erster Ebene haben wir m Additionen, auf der zweiten Ebene m^2 Additionen, usw.)

$$
\begin{aligned}
c_{Add}^{Rek}(n) &= m + m^2 + \ldots + m^n \quad |\text{Anw. der Summenformel für geo. Reihen} \\
&= m \cdot \frac{m^n - 1}{m - 1} = \frac{m}{m-1}(m^n - 1) \\
&= \frac{4}{3}(4^n - 1) \quad \text{hier mit } m = 4 \quad (\text{Anzahl der Objekte})
\end{aligned}
$$

Erklärungen

Grobe Idee: Wir gehen in der Methode `bestWertRekursiv` über alle Elemente und probieren aus ob wir diese einmal in den Rucksack packen können, d. h. die (verbleibende) Traglast ausreicht. Falls ja, dann führen wir einen rekursiven Aufruf durch bei dem wir die Traglast entsprechende reduziert haben.

Details: Für jedes Element entscheiden wir, ob es noch in den Rucksack passt (Zeile 7). Falls ja, dann wird der Wert des Elements addiert und die Traglast um das Gewicht des Elements reduziert (Zeile 8: `n - gewt`). Anschließend wird rekursiv der bester Wert für den kleineren Rucksacks berechnet.

Rucksackproblem - iterative Lösung

Grundsätzliche Idee der iterativen Lösung

Gehe über alle Objekte. Berechne in jedem Schleifendurchlauf `i` bei Hinzunahme von Teil `i` das jeweils beste Ergebnis für alle Kapazitäten `j` bis inklusive `n`.

Beispiel

Verfügbare Objekte $((Gewicht, Wert))$: $A = \{(1,1),(3,4),(5,8),(2,3)\}$. Sei die maximale Traglast $n = 7$:

i\j	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	0	1	2	4	5	6	8	9
2	0	1	2	4	5	8	9	10
3	0	1	3	4	6	8	9	11

Implementierung

```
1  gW = [ (1, 1), (3, 4), (5, 8), (2, 3) ] # (Gewicht, Wert)
2
3  def bestWertIterativ(n):    # n = max. Traglast
4      best = [0] * (n + 1)    # best[i] = bester Wert für Traglast i (initial 0)
5      for i in range(len(gW)):
6          (gewt, wert) = gW[i]
7          for j in range(gewt, n + 1): # range(gewt, n + 1): [gewt, ..., n]
8              test = best[j - gewt] + wert
9              if test > best[j]:
10                 best[j] = test
11
12     return best[n]
13
14 print(bestWertIterativ(5))  # max. Traglast ist hier zu Beginn n = 5
```

Komplexitätsanalyse

Bei den Iterationen ergibt sich:

Zwei Schleifen über m und n:

$$\begin{aligned} c_{Add}^{Ite}(n) &= m \cdot n \\ &= 4n \quad \text{hier mit } m = 4 \end{aligned}$$

Erklärungen

Grobe Idee: Wir gehen in der Methode `bestWertIterativ` über alle Objekte (Zeile 5). In der inneren Schleife (Zeile 7) iterieren wir über die Traglasten, die das Objekt — ggf. auch mehrfach — aufnehmen könnten (`range(gewt, n + 1)`). Für jede dieser

Traglasten prüfen wir ob es vorteilhaft ist das Objekt in den Rucksack zu packen. Falls ja, dann wird der aktuell beste Wert für die Traglast aktualisiert.

D. h. wir legen zum Beispiel ein Objekt mit dem Gewicht 2 bei einer verbleibenden Traglast von 5 ggf. (implizit) dadurch mehrfach in den Rucksack, dass wir bereits den besten Wert für die kleineren Traglasten kennen.

Rucksackproblem - Vergleich

$$
\begin{aligned}
c_{Add}^{Rek}(n) &= \frac{m}{m-1}(m^n - 1) \\
&= \frac{4}{3}(4^n - 1)
\end{aligned}
\qquad
\begin{aligned}
c_{Add}^{Ite}(n) &= m \cdot n \\
&= 4n
\end{aligned}
$$

Zusammenfassung

Die iterative Variante ist wegen der vermiedenen Berechnung gleicher Werte – aufgrund der Verwendung von dynamischer Programmierung – praktisch immer schneller. Dies könnte bei Rekursion ggf. mit Caching erreicht werden.

? Frage

Wieso ist das Rucksackproblem dann aber als NP-vollständig klassifiziert?

Die Analyse erfolgte nicht über die Wortlänge (als Eingabegröße); d. h. n (Kapazität bzw. Tragkraft) entspricht nicht der Wortlänge. Ein Binärwort n mit k Zeichen (Bits) hat bis zu $2^k - 1$ Werte.

$$c_{Add}^{Rek}(2^k - 1) = \frac{4}{3}(4^{2^k - 1} - 1) \in O(4^{2 \cdot k})$$

$$c_{Add}^{Ite}(2^k - 1) = 4(2^k - 1) \in \Theta(2^k)$$

!! Wichtig

Der erste Vergleich der Algorithmen ist valide in Hinblick auf die relative Laufzeit beider Varianten. Für die Komplexitätsklassifizierung ist jedoch die Wortlänge entscheidend.

Es ist immer genau zu prüfen was die Wortlänge ist!

Die Wortlänge eines Problems bezeichnet hier die Anzahl der Bits, die benötigt werden, um die Eingabe eines Problems darzustellen. Sie ist ein Maß dafür, wie groß oder komplex die Darstellung der Eingabedaten ist.

Die iterative Variante mit dynamischer Programmierung hat eine Laufzeit von $O(m \cdot n)$ wobei n hier die Kapazität in Gewichtseinheiten ist, nicht die Wortlänge. Wenn n exponentiell groß ist, wird der Algorithmus ineffizient, da die Eingabegröße $\lceil log_2 N \rceil$ viel kleiner ist als N selbst. (D. h. wenn die Kapazität 10 ist, dann brauchen wir 4 Bits, um die Kapazität darzustellen, wenn die Kapazität jedoch 1000 (100 mal größer) ist, dann brauchen wir 10 Bits (d. h. nur 2,5 mal so viele Bits.)

Rekursiv teilende Algorithmen

Standardvorgehensweise bei der Analyse

Standardverfahren zur Analyse rekursiver Algorithmen:

1. Anwendung der Verfahren zur Analyse iterativer Algorithmen, um die Rekurrenzgleichung zu bestimmen.
2. Eine Anzahl von Werten ausrechnen und auf sinnvollen Zusammenhang schließen.
3. Beweis des Zusammenhangs mit vollständiger Induktion.

> ⚠ Achtung!
>
> Das Finden eines sinnvollen Zusammenhangs und der Beweis ist nicht immer einfach.

Dieses Verfahren haben wir bei den Türmen von Hanoi angewandt.

Rekurrenzgleichungen sind Gleichungen, die eine Folge a_n durch ihre vorherigen Elemente definieren. Sie beschreiben die Entwicklung eines Wertes a_n in Abhängigkeit von vorhergehenden Werten.

Beobachtung bzgl. rekursiv teilender Algorithmen

Teilende Verfahren, *bzw. Divide-and-Conquer-Algorithmen*, sind typischerweise sehr effizient.

Beispiel

Wird beispielsweise das Problem immer halbiert, ist also $a_{2n} = a_n + 1$ und ist $a_1 = 1$, dann würde für die Folgenglieder gelten $a_1 = 1, a_2 = 2, a_4 = 3, a_8 = 4, a_{16} = 5, \ldots$.

Verallgemeinert: $a_n = \log_2(n) + 1$.

Herleitung:

$a_1 = \log_2(1) + 1 = 0 + 1$

$a_{2n} = a_n + 1 = \log_2(n) + 1 + 1 = \log_2(n) + \log_2(2) + 1 = \log_2(2n) + 1$

Ein konkretes Beispiel ist die binäre Suche nach einem Namen im Telefonbuch oder nach einer zu erratenden Zahl.

Bei der Herleitung wurde (wieder) vollständige Induktion angewandt und die Logarithmusgesetze genutzt: $\log(a) + \log(b) = \log(a \cdot b)$ sowie $\log_b b = 1$.

Rekurrenzgleichung für rekursiv teilende Algorithmen

- In vielen Fällen geben rekursiv teilende Algorithmen Grund zur Hoffnung, dass die Laufzeit einen relevanten logarithmischen Anteil hat.
- Häufig können die Rekurrenz-Gleichungen rekursiv teilender Algorithmen in folgende Form gebracht werden:

  Sei:

  - a: die Anzahl der rekursiven Aufrufe,
  - $\frac{n}{b}$: die Größe jedes rekursiven Unterproblems wobei b die Anzahl der Teile ist in die das Problem geteilt wird,
  - $f(n)$: der Aufwand während der Ausführung (z. B. der Aufwand für das Teilen der Eingabedaten und das Zusammenführen der Teillösungen).

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

In diesem Fall können drei Fälle unterschieden identifiziert werden:

1. Ist der Aufwand $f(n)$ vernachlässigbar gegenüber dem Aufwand der weiteren Aufrufe, so ist ein rein durch die Rekursion bestimmtes Verhalten zu erwarten.
2. Entspricht der Aufwand $f(n)$ genau dem Aufwand der weiteren Aufrufe, so vervielfältigt sich der Aufwand gegenüber dem 1. Fall, bleibt aber in der gleichen Größenordnung.
3. Ist der Aufwand $f(n)$ größer als der Aufwand der verbleibenden Aufrufe, so wird der Aufwand asymptotisch von $f(n)$ dominiert.

Beispiel für den 1. Fall

Bei $a = 1$ und $b = 2$ — wie bei der binären Suche — ist somit logarithmisches Verhalten zu erwarten. Wird hingegen ein $b = 2$ halbiertes Feld $a = 4$ viermal aufgerufen, so ist ein quadratisches Verhalten zu erwarten.

Lösen von Rekurrenzgleichungen mit dem Master-Theorem

Das Master-Theorem ist ein Werkzeug zur Analyse der Zeitkomplexität von rekursiven Algorithmen, die mit Hilfe von Rekurrenzgleichungen der Form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ beschrieben werden können.

Anwendungsgebiet sind insbesondere Teile-und-Herrsche Algorithmen.

Das Master-Theorem hat drei Fälle, die auf dem Vergleich zwischen $f(n)$ und $n^{\log_b a}$ basieren und die asymptotische Komplexität von $T(n)$ bestimmen. Wobei $n^{\log_b a}$ die Laufzeit für die Rekursion selbst beschreibt:

Seien $a > 0$ und $b > 1$ Konstanten und $f : \mathbb{N} \to \mathbb{N}$:

1. Wenn $f(n) \in O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$ gilt – d. h. wenn $f(n)$ langsamer wächst als $n^{\log_b a}$ – dann dominiert die Rekursion, und es gilt: $T(n) \in \Theta(n^{\log_b a})$.
2. Wenn $f(n) \in \Theta(n^{\log_b a} \cdot (\log n)^k)$ für ein $k \geq 0$ gilt – d. h. wenn $f(n)$ und $n^{\log_b a} \cdot (\log n)^k$ gleich schnell wachsen – dann tragen beide Teile zur Gesamtkomplexität bei, und es gilt: $T(n) \in \Theta(n^{\log_b a} \cdot (\log n)^{k+1})$.
3. Wenn $f(n) \in \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ gilt und weiterhin gilt $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und ein hinreichend großes n – d. h. wenn also $f(n)$ schneller wächst als $n^{\log_b a}$ – dann dominiert $f(n)$ die Komplexität, und es gilt: $T(n) \in \Theta(f(n))$.

Viele Sortieralgorithmen sind zum Beispiel Teile-und-Herrsche Algorithmen.

※ Hinweis

Nicht immer kann das Master-Theorem angewandt werden, da es nur für spezielle Rekurrenzgleichungen gilt.

Im Mastertheorem erfolgt der Vergleich ggf. mit $n^{(\log_b a) - \epsilon}$ und nicht mit $n^{\log_b (a - \epsilon)}$.

Anwendung des Master-Theorems: 1. Beispiel

Gegeben sei: $T(n) = 2T(n/2) + n \log_2 n$

Somit gilt: $a = 2$, $b = 2$ und $n^{\log_2 2} = n$

Analyse: Es liegt Fall 2 vor, da $f(n) = n \cdot (\log_2 n)^{k=1} \in \Theta(n^{\log_b a} \cdot (\log n))$.

Ergebnis: Die Laufzeit beträgt somit $T(n) = \Theta(n \cdot (\log_2 n)^2)$.

Der Wechsel der Basis des Logarithmus ist möglich, da sich die Basis nur um einen konstanten Faktor unterscheidet:

$\log_{a} x = \frac{1}{\log_b a} \cdot \log_b x$

Anwendung des Master-Theorems: 2. Beispiel

Gegeben sei: $T(n) = 9T(n/3) + 2n$

Somit gilt: $a = 9$, $b = 3$ und $n^{\log_3 9} = n^2$

Analyse: Es liegt Fall 1 vor, da $f(n) = 2n \in O(n^{\log_3 9 - \epsilon})$.

Ergebnis: Die Laufzeit beträgt somit $T(n) = \Theta(n^2)$.

Anwendung des Master-Theorems: 3. Beispiel

Gegeben sei: $T(n) = 2T(n/3) + n$

Somit gilt: $a = 2, b = 3$ und $n^{\log_3 2}$, $log_3 2 \approx 0,63 < 1$

Analyse: Es liegt Fall 3 vor, da $f(n) = n \in \Omega(n^{\log_3 2+\epsilon})$ und $af(n/b) = 2n/3 \leq c \cdot n$ für $1 > c \geq 2/3$.

Ergebnis: Die Laufzeit beträgt somit $T(n) = \Theta(n)$.

Zusammenfassung

Das Master-Theorem hilft also, die asymptotische Komplexität von Algorithmen schnell zu bestimmen, ohne dass eine detaillierte Analyse der Rekurrenz erforderlich ist.

Übung

2.5. f(n) ist konstant

Gegeben sei: $T(n) = 2T(n/4) + 1$

- Bestimmen Sie die Laufzeit des Algorithmus mit Hilfe des Master-Theorems.

2.6. f(n) ist die Quadratwurzel

Gegeben sei: $T(n) = 3T(n/9) + \sqrt{n}$

- Bestimmen Sie die Laufzeit des Algorithmus mit Hilfe des Master-Theorems.

2.7. a=1 und f(n) sind konstant

Gegeben sei: $T(n) = T(n/2) + 1$

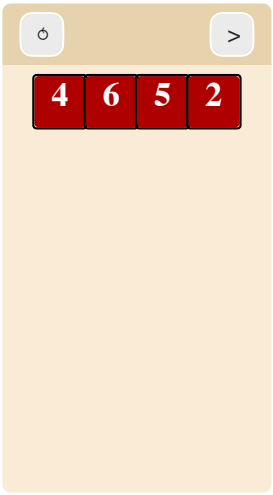
- Bestimmen Sie die Laufzeit des Algorithmus mit Hilfe des Master-Theorems.

Übung

Mergesort

Der Mergesort-Algorithmus ist ein rekursiver Algorithmus, der ein Array solange in zwei Hälften teilt (Teile und Herrsche (*divide and conquer*)) bis diese trivial sind. Danach werden jeweils zwei sortierte Hälften zusammengeführt bis das ganze Array wieder zusammengeführt wurde. Das Zusammenführen der Hälften hat einen Aufwand von n und das Teilen des Arrays hat einen konstanten Aufwand.

Mergesort (Visualisierung)



Schritt:
...

2.8. Anwendung des Master-Theorems auf Mergesort

- Bestimmen Sie die Rekurrenzgleichung für den Mergesort-Algorithmus.
- Bestimmen Sie die Laufzeit des Mergesort-Algorithmus mit Hilfe des Master-Theorems.