Datenstrukturen[1]

Sommersemester 2017

Herzlich willkommen!

[1] Basierend auf Folien von Prof. Georg Schnitger, Prof. Ulrich Meyer, Dr. Iganz Rutter

Wer ist wer?

Wir:

- Martin Hoefer (Vorlesungen)
  R 115 – RMS 11–15, mhoefer@cs.uni-frankfurt.de
- Niklas Hahn (Übungskoordination)
  R106 – RMS 11–15, ds17-support@cs.uni-frankfurt.de
- Pirmin Braun, Ngoc Minh Do, Stefan Franke, Sebastian Kriege, Helen Naumann, Martin Parnet, Conrad Schecker, Tim Schön, Joshua Sole (Tutoren)

Wer sind Sie?

Worum geht's?

Abstrakte Datentypen und Datenstrukturen

Ein **abstrakten Datentyp** ist eine

Sammlung von Operationen (auf einer Menge von Objekten).

Implementiere den abstrakten Datentyp durch eine **Datenstruktur**.

Ein Beispiel für häufig auftretende Operationen:

das Einfügen und Entfernen von Schlüsseln nach ihrem Alter.

- Wenn der jüngste Schlüssel zu entfernen ist:
  Wähle die Datenstruktur „**Keller**“ (engl. „**Stack**“).
- Wenn der älteste Schlüssel zu entfernen ist:
  Wähle die Datenstruktur „**Schlange**“ (engl. „**Queue**“).
- Wenn Schlüssel Prioritäten besitzen und der Schlüssel mit höchster Priorität zu entfernen ist: Wähle die Datenstruktur **Heap**.

Worum geht's?

- Gegeben ist ein *abstrakter Datentyp*.
- Entwerfe eine *möglichst effiziente* Datenstruktur.

- Welche abstrakten Datentypen?
  - **Keller**:
    Einfügen und Entfernen des jüngsten Schlüssels.
  - **Warteschlangen**:
    Einfügen und Entfernen des ältesten Schlüssels.
  - **Prioritätswarteschlangen**:
    Einfügen und Entfernen des wichtigsten Schlüssels.
  - **Wörterbücher**:
    Einfügen, Entfernen (eines beliebigen Schlüssels) und Suche.
- Was heißt Effizienz?
  - Minimiere **Laufzeit** und **Speicherplatzverbrauch** der einzelnen Operationen.
  - Wie **skalieren** die Ressourcen mit wachsender Eingabelänge?

  $\rightarrow$ Asymptotische Analyse von Laufzeit und Speicherplatzverbrauch

Worauf wird aufgebaut?

(1) Diskrete Modellierung (bzw. Vorkurs):

- Mengen, Relationen, Worte und Tupel
- Vollständige Induktion und allgemeine Beweismethoden.
- Siehe hierzu auch die Seite des Vorkurs Informatik mit Folien zu Beweistechniken, Induktion und Rekursion und Asymptotik und Laufzeitanalyse.

(2) Grundlagen der Programmierung 1:

- Kenntnis der elementaren Datenstrukturen ist hilfreich, aber nicht notwendig.

(3) Mathematik 1 (bzw. Vorkurs):

- Logarithmen,
- Grenzwerte und
- asymptotische Notation.

Webseite der Veranstaltung

Konsultieren Sie **regelmäßig** die Webseite

http://tinygu.de/ds17

- Organisatorische Details zum Übungsbetrieb und zur Notengebung werden beschrieben.
- Unter **Aktuelles** finden Sie zum Beispiel:
  - Anmerkungen zum Übungsbetrieb (wie melde ich mich an?)
  - Gegebenenfalls Anmerkungen zu aktuellen Übungsaufgaben.
- Nicht angemeldet? Sonstige organisatorische Fragen/Probleme?
  → Mail an `ds17-support@cs.uni-frankfurt.de`
  oder `ds17-support@dlist.server.uni-frankfurt.de`
- Im **Logbuch** finden Sie Informationen,
  - Beamer-Folien, Videos, weitere Referenzen

  zu den einzelnen Vorlesungsstunden.

Literatur (1/2)

- Skript und Folien zur Vorlesung finden Sie auf der Webseite der Veranstaltung.
- Kapitel 3-5, 10-12 und 18 in T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, “Introduction to Algorithms”, second edition, 1990.
- M. T. Goodrich, R. Tamassia, D. M. Mount, “Data Structures and Algorithms in C++”, 2003.
- M. Dietzfelbinger, K. Mehlhorn, P. Sanders, “Algorithmen und Datenstrukturen, die Grundwerkzeuge”, Springer Vieweg, 2014.
- R. Sedgewick, “Algorithmen in C++”, Kapitel 1-7 im Grundlagenteil.
- Kapitel 1-3 in J. Kleinberg, E. Tardos, “Algorithm Design” (GL-1 Textbuch), Pearson New International Edition, 2013.
- Für mathematische Grundlagen:
  - N. Schweikardt, Diskrete Modellierung, eine Einführung in grundlegende Begriffe und Methoden der Theoretischen Informatik, 2013. (Kapitel 2,5)
  - D. Grieser, Mathematisches Problemlösen und Beweisen, eine Entdeckungsreise in die Mathematik, Springer Vieweg 2012.

Literatur (2/2)

Sämtliche Textbücher befinden sich auch in der Bibliothek im

Semesterapparat der Veranstaltung “Datenstrukturen”.

Die **Bibliothek** befindet sich im 1. Stock in der Robert-Mayer Strasse 11-15.

Organisatorisches

Klausur

- Hauptklausur: Donnerstag, **03. August 2017**, ab 9:00 Uhr im H VI.
- Zweitklausur: Montag, **09. Oktober 2017**, ab 9:00 Uhr im H VI.
- Die in den Übungen erreichten Punkte werden mit einem Maximalgewicht von **10%** zu den Klausurpunkten hinzugezählt:

  Werden in der Klausur x% und in den Übungen y% erzielt, dann ist $z = x + (y/10)$ die Gesamtpunktzahl.

  Die Note hängt nur von der Gesamtpunktzahl z ab. Die Veranstaltung ist bestanden, wenn $z \geq 50$.

Übungsbetrieb

- Übungsblätter werden im 2-Wochen Rhythmus auf die Webseite gestellt:

  Übungskalender auf der Webseite!

  Lösungen (zusammengetackert, mit vollem Namen, Matrikelnummer, und Nummer der Übungsgruppe versehen) sind nach **2-wöchiger Bearbeitungszeit** abzugeben:

  - Entweder im H VI **vor Beginn** der Vorlesung oder
  - in den Briefkasten einwerfen zwischen Büros 114 und 115, Robert-Mayer-Str. 11-15.

- Übungsgruppen treffen sich ab nächste Woche Dienstag, 25.4., um das **umfangreiche Präsenzblatt** zu besprechen.
- Erstes Übungsblatt erscheint heute, Abgabe am 02.05.2017
- Übungsaufgaben sollten mit Anderen besprochen werden, aber Lösungen **müssen eigenständig aufgeschrieben** werden!
  - 1. Verstoß: Nicht-Anrechnung aller Punkte des Blatts für ALLE Beteiligten.
  - 2. Verstoß: ALLE Übungspunkte gestrichen.

Bevor es losgeht . . .

- Vor- und Nachbereitung der Vorlesungen
- **Unbedingt** am Übungsbetrieb teilnehmen und Aufgaben bearbeiten!

  Von den Teilnehmern, die 50% der Übungspunkte erreichen, bestehen nahezu 100% auch die Klausur!

- Teamarbeit vs. Einzelkämpfer
- Studieren lernen – der richtige Umgang mit den Freiheiten

- Meine Sprechstunde: Montags 14:00–16:00 Uhr.
  Oder versuchen Sie es auf gut Glück (Raum 115).

Helfen Sie uns durch

- ihre **Fragen**,
- **Kommentare**
- und **Antworten**!

Die Veranstaltung kann nur durch **Interaktion** interessant werden.

Weitere Links und nützliche Informationen

- Im

  Ingo-Wegener Lernzentrum

  treffen Sie ihre KommilitonInnen.

  Frau Düffel hilft mit Rat in allen Lebenslagen, nicht nur bei Fragen zu Datenstrukturen.

- Informationen nur für Erstsemester: http://www.informatik.uni-frankfurt.de/index.php/de/studierende-informationen-fur-erstsemester.html

Mathematische Grundlagen

Vollständige Induktion

Ziel: Zeige die Aussage $A(n)$ für alle natürlichen Zahlen $n \geq n_0$.

Die Methode der **vollständigen Induktion**

1. Zeige im **INDUKTIONSANFANG** die Aussage $A(n_0)$ und
2. im **INDUKTIONSSCHRITT** die Implikation

$$(A(n_0) \wedge A(n_0 + 1) \wedge \cdots \wedge A(n)) \Rightarrow A(n + 1)$$

   für jede Zahl n mit $n \geq n_0$.

   $A(n_0) \wedge A(n_0 + 1) \wedge \cdots \wedge A(n)$ heißt *Induktionsannahme*.

Häufig nimmt man „nur“ die Aussage $A(n)$ in der *Induktionsannahme* an.

Beachte aber das folgende Beispiel der Fibonacci-Zahlen.

Die Summe der ersten n Zahlen

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n \cdot (n+1)}{2} \; .$$

- Vollständige Induktion nach n:
  - Induktionsanfang für $n = 1$: $\sum_{i=1}^{1} i = 1$ und $\frac{1 \cdot (1+1)}{2} = 1$. ✓
  - Induktionsschritt von n auf $n + 1$:
    - Wir nehmen an, dass $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$ gilt.
    - $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n \cdot (n+1)}{2} + (n+1) = \frac{n \cdot (n+1) + 2 \cdot (n+1)}{2} = \frac{(n+2) \cdot (n+1)}{2} = \frac{(n+1) \cdot (n+2)}{2}$ und das war zu zeigen. ✓
- Ein direkter Beweis:
  - Betrachte ein Gitter mit n Zeilen und n Spalten: n^2 Gitterpunkte.
  - Wir müssen die Gitterpunkte unterhalb der Hauptdiagonale und auf der Hauptdiagonale zählen.
  - Die Hauptdiagonale besitzt n Gitterpunkte und unterhalb der Hauptdiagonale befindet sich die Hälfte der verbleibenden $n^2 - n$ Gitterpunkte. Also folgt $\sum_{i=1}^{n} i = n + \frac{n^2 - n}{2} = \frac{n \cdot (n+1)}{2}$.

Karnickel (1/3)

Ein Bauer züchtet Kaninchen. Jedes weibliche Kaninchen bringt im Alter von zwei Monaten ein weibliches Kaninchen zur Welt und danach jeden Monat ein weiteres. Wie groß ist die Anzahl

$$\mathbf{fib}(\mathbf{n})$$

der weiblichen Kaninchen am Ende des n-ten Monats, wenn der Bauer mit einem neu geborenen weiblichen Kaninchen im ersten Monat startet?

Antwort: $\text{fib}(n)$ ist rekursiv wie folgt definiert:

- **Rekursionsanfang**: $\text{fib}(1) \; := \; 1$ und $\text{fib}(2) \; := \; 1$,
- **Rekursionsschritt:** $\text{fib}(n+1) \; := \; \text{fib}(n) + \text{fib}(n-1)$ für alle $n \in \mathbb{N}$ mit $n \geq 2$.

Die Funktion fib wird **Fibonacci-Folge**[a] genannt, die Zahl $\text{fib}(n)$ heißt n-te Fibonacci-Zahl.

[a] Zu Ehren von Leonardo Fibonacci (13. Jh.), einem italienischen Mathematiker

Zur Erinnerung: Es ist

- $\mathrm{fib}(1) := 1$ und $\mathrm{fib}(2) := 1$ und
- $\mathrm{fib}(n+1) := \mathrm{fib}(n) + \mathrm{fib}(n-1)$ f.a. $n \in \mathbb{N}$ mit $n \geq 2$.

Somit gilt:

n	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{fib}(n)$	1	1	2	3	5	8	13	21	34	55	89	144

Wie stark wächst die Folge $\mathrm{fib}(n)$?

- Zeige,
$$\mathrm{fib}(n) \leq 2^n$$
mit der erweiterten Induktionsannahme und $n_0 = 1$.
- Zeige
$$2^{n/2} \leq \mathrm{fib}(n) \text{ gilt für } n \geq 6$$
mit der erweiterten Induktionsannahme und $n_0 = 6$.

Es gibt auch einen expliziten Ausdruck für die n-te Fibonacci-Zahl.

F.a. $n \in \mathbb{N}_{>0}$ gilt nämlich:

$$\mathrm{fib(n)} = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right).$$

- Man zeigt die Bemerkung durch vollständige Induktion nach n.
- Die zentrale Beobachtung:
  Sowohl $\frac{1+\sqrt{5}}{2}$ wie auch $\frac{1-\sqrt{5}}{2}$ erfüllen die quadratische Gleichung

$$x^2 = x + 1.$$

Analyse rekursiver Programme

Rekursive Programme und ihre Rekursionsbäume

Sei x eine Eingabe für ein rekursives Programm $R(\vec{p})$ (mit den Parametern $\vec{p}$). Wir führen einen **Rekursionsbaum** nach den folgenden Regeln ein.

1. Beschrifte die Wurzel von B mit den Parametern $\vec{p}$ des Erstaufrufs.
   - Innerhalb von $R(\vec{p})$ keine rekursiven Aufrufe getätigt $\Rightarrow$ Wurzel wird zu Blatt.
   - Sonst: Wurzel erhält Kind für jeden rekursiven Aufruf innerhalb von $R(\vec{p})$.
     Kind wird mit Parametern des rekursiven Aufrufs beschriftet.
2. Behandle Knoten v von B, beschriftet mit Parametern $\vec{q}$, genau wie Wurzel.
   - Innerhalb von $R(\vec{q})$ keine rekursiven Aufrufe getätigt $\Rightarrow$ v wird zu Blatt.
   - Sonst: Knoten v erhält Kind für jeden rekursiven Aufruf in $R(\vec{q})$.
     Kind wird mit Parametern des rekursiven Aufrufs beschriftet.

Was erzählt uns der Rekursionsbaum?
Zum Beispiel ist die *Anzahl der Knoten* von B gleich der *Anzahl aller rekursiven Aufrufe* für Eingabe x (und Parameter $\vec{p}$).

Eine Berechnung der Fibonacci-Zahlen auf die grausame Art

Eine rekursive Berechnung der Fibonacci-Zahlen

Algo(n):

1. *Falls* $n = 1$, *dann "return"* $\textit{Algo}(1) := 1$.
2. *Falls* $n = 2$, *dann "return"* $\textit{Algo}(2) := 1$.
3. *Falls* $n \geq 3$, *dann "return"* $\textit{Algo}(n) := \textit{Algo}(n-1) + \textit{Algo}(n-2)$.

Zähle jede Addition, jeden Vergleich, und jede Ergebnis-Rückgabe als einen Schritt. Für die Anzahl $g(n)$ benötigter Schritte gilt:

$$\begin{aligned} g(1) &= 2, \quad g(2) = 3 \quad \text{und} \\ g(n) &= 3 + g(n-1) + g(n-2) + 2 = 5 + g(n-1) + g(n-2) \end{aligned}$$

Um Himmels willen, für $n \geq 6$ ist

$$g(n) \geq \mathrm{fib}(n) \geq 2^{n/2},$$

die Laufzeit ist **exponentiell** ↯ **Das geht doch viel, viel schneller, oder!?!**

Der Rekursionsbaum (1/2)

Die Rekursionsbäume B_n für `Algo`(n) sind rekursiv definiert:

- **Rekursionsanfang**: Die Bäume B_1 und B_2 bestehen nur aus einem einzigen Knoten der jeweils mit „1", bzw. „2" markiert ist.
- **Rekursionsschritt**: Die Wurzel r von B_n ist mit n markiert, die Wurzel von B_{n-2} ist linkes Kind von r und die Wurzel von B_{n-1} rechtes Kind von r.

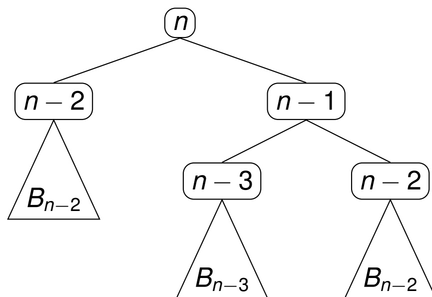

Abbildung : Die rekursive Definition des Baums B_n

Der Rekursionsbaum (2/2)

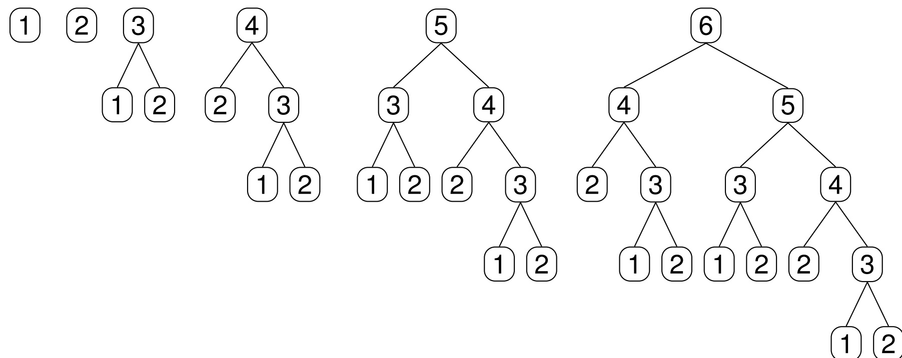

Abbildung : Die „Fibonacci-Bäume“ B_1, B_2, B_3, B_4, B_5 und B_6.

1. Die Wurzel von B_n berechnet die n-te Fibonacci-Zahl.
2. Die Fibonacci-Zahl fib(4) wird 2-mal in B_6 berechnet, fib(3) wird 3-mal berechnet.

WARUM?

Binärsuche

Binärsuche

Ein Array

$$A = (A[1], \ldots, A[n])$$

von n Zahlen und eine Zahl

$$x$$

ist gegeben: Wir möchten wissen, ob und wenn ja wo die Zahl x in A vorkommt.

- Wenn A nicht sortiert ist, dann wird die **lineare Suche** bis zu n Zellen des Arrays inspizieren.
- Wenn das Array aber aufsteigend sortiert ist, dann können wir Binärsuche anwenden.

Binärsuche: Das Programm

```
void Binärsuche( int unten, int oben){
  if (oben < unten)
    std::cout « x « “ wurde nicht gefunden.“
     « std::endl;
  int mitte = (unten+oben)/2;
  if (A[mitte] == x)
    std::cout « x « “ wurde in Position “
     « mitte « “ gefunden.“ « std::endl;
    else {
      if (x < A[mitte])
        Binärsuche(unten, mitte-1);
      else
        Binärsuche(mitte+1,oben);}}
```

Wir sagen, dass Zelle

`A[mitte]`

im ersten Aufruf *inspiziert* wird.

Binärsuche: Korrektheit

Wir zeigen durch vollständige Induktion nach

$$n = oben - unten + 1,$$

dass `Binärsuche(unten,oben)` korrekt ist.

(a) INDUKTIONSANFANG für $n = 0$: Es ist $n = oben - unten + 1 = 0$ $\Rightarrow oben < unten$.

`Binärsuche`$(unten, oben)$ bricht mit der Antwort „x wurde nicht gefunden“ ab. ✓

(b) INDUKTIONSSCHRITT $n \to n + 1$: Es ist $oben - unten + 1 = n + 1$.

- Wenn $A(mitte) = x$, dann wird mit Erfolgsmeldung abgebrochen. ✓
- Ansonsten, wenn $x < A[mitte]$, sucht `Binärsuche` richtigerweise in der linken Hälfte und sonst in der rechten Hälfte.
- Die rekursive Suche in der linken bzw. rechten Hälfte verläuft aber nach Induktionsannahme korrekt. ✓

Binärsuche: Die Laufzeit

Wie groß ist

$$T(n) := \text{maximale Anzahl inspizierter Zellen für ein Array mit } n \text{ Zellen,}$$

für

$$n = 2^k - 1?$$

Hier ist eine rekursive Definition von $T(n)$:

- **Rekursionsanfang**: $T(0) := 0$.
- **Rekursionsschritt:** $T(n) := T(\frac{n-1}{2}) + 1$.

1. Wir haben $n = 2^k - 1$ gefordert. Beachte, dass $\frac{n-1}{2} = \frac{2^k-2}{2} = 2^{k-1} - 1$ gilt.
2. Wir zeigen $T(2^k - 1) = k$ mit vollständiger Induktion nach k.
   (a) **INDUKTIONSANFANG:** $T(2^0 - 1) = T(0) = 0$. ✓
   (b) **INDUKTIONSSCHRITT:** $T(2^{k+1} - 1) = T(2^k - 1) + 1 \overset{\text{Induktionsannahme}}{=} k + 1$. ✓

Binärsuche muss höchstens k Zahlen inspizieren gegenüber $2^k - 1$ Zahlen für die lineare Suche: Die lineare Suche ist exponentiell langsamer als Binärsuche.

Binärsuche: Der Rekursionsbaum

Wie sieht der Rekursionsbaum B für ein Array A und einen Schlüssel y aus, der nicht in A vorkommt?

1. Der Rekursionsbaum B besitzt den **Aus-Grad Eins**: Ein Knoten von B ist entweder ein Blatt oder verursacht `genau einen` rekursiven Aufruf.

2. Das Array A besitze genau $n = 2^k - 1$ Schlüssel.
   Wir wissen: Im schlimmsten Fall werden genau k Zellen inspiziert.

   - y kommt nicht in A vor: Der schlimmste Fall tritt ein.
   - In jedem rekursiven Aufruf wird nur eine Zelle des Arrays inspiziert: Der Rekursionsbaum B ist also ein Weg der Länge k.

Binärsuche ist schnell, weil stets **höchstens ein rekursiver Aufruf** getätigt wird und weil die Länge des Array-Abschnitts in dem gesucht wird, **mindestens halbiert** wird.

Unser Freund, der Logarithmus

Rechnen mit Logarithmen

Seien $a > 1$ und $x > 0$ reelle Zahlen. $\log_a(x)$ ist der Logarithmus von x zur Basis a und stimmt mit z genau dann überein, wenn

$$a^z = x.$$

Insbesondere ist $a^{\log_a(x)} = x$.

(a) $\log_a(x \cdot y) = \log_a x + \log_a y$.
(b) $\log_a(x^y) = y \cdot \log_a(x)$.
(c) $\log_a x = (\log_a b) \cdot (\log_b x)$. Wir „übersetzen" von Basis a in Basis b.

Was ist $4^{\log_2 n}$?

- $\log_2 n = (\log_2 4) \cdot (\log_4 n)$ mit (c).
- $4^{\log_2 n} = 4^{(\log_2 4)\cdot(\log_4 n)} = (4^{\log_4 n})^{\log_2 4} = n^2$.

Was ist $b^{\log_a x}$?

- $\log_a x = (\log_a b) \cdot (\log_b x)$ mit (c).
- $b^{\log_a x} = b^{(\log_a b)\cdot(\log_b x)} = (b^{\log_b x})^{\log_a b} = x^{\log_a b}$.

Laufzeitmessung

Eingabelänge

Wie gut *skaliert* ein Programm, d.h.
wie stark nimmt die Laufzeit zu, wenn die Eingabelänge vergrößert wird?

Was ist denn überhaupt die **Länge einer Eingabe**?

$\rightarrow$ Definiere Eingabelänge abhängig von der Problemstellung. Zum Beispiel:

(a) Wenn ein Array mit n „Schlüsseln“ zu sortieren ist, dann ist

die Anzahl n der Schlüssel

ein vernünftiges Maß für die Eingabelänge.

(b) Wenn ein algorithmisches Problem für einen Graphen mit Knotenmenge V und Kantenmenge E zu lösen ist, dann ist die Summe

$$|V| + |E|$$

der Knoten- und Kantenzahl sinnvoll.

Die worst-case Laufzeit

Sei P ein Programm. Für jede Eingabelänge n setze

$$\text{Zeit}_P(n) = \textbf{größte Laufzeit} \text{ von } P \text{ auf einer Eingabe der Länge } n.$$

Wenn also die Funktion

$$\text{Länge} : \text{Menge aller möglichen Eingaben} \to \mathbb{N}$$

einer Eingabe x die Eingabelänge „Länge(x)" zuweist, dann ist

$$\text{Zeit}_P(n) = \max_{\substack{\text{Eingabe } x \\ \text{Länge}(x)=n}} \text{Anzahl der Schritte von Programm } P \text{ für Eingabe } x$$

die „**worst-case Laufzeit**" von P für Eingaben der Länge n.

Laufzeitmessung

Für welchen Rechner sollen wir die Laufzeit berechnen?

- Analysiere die Laufzeit auf einem abstrakten Rechner:
  - Der Speicher besteht aus Registern, die eine ganze Zahl speichern können.
  - Eine CPU führt einfache boolesche und arithmetische Operationen aus.
  - Daten werden zwischen CPU und Speicher durch (indirekte) Lade- und Speicherbefehle ausgetauscht.
- Damit ist die Laufzeitberechnung für jeden modernen sequentiellen Rechner gültig, aber wir erhalten für einen **speziellen Rechner** nur eine

  bis auf einen konstanten Faktor

  exakte Schätzung.

Laufzeitmessung für welche Programmiersprache und welchen Compiler?

- Wir „sollten“ mit einer Assemblersprache arbeiten: Wir sind vom Compiler unabhängig und die Anzahl der Anweisungen ist relativ klein.
- Aber die Programmierung ist viel zu umständlich und wir wählen deshalb

  C++ bzw. **Pseudocode**.

  - Die Anzahl ausgeführter C++ Befehle ist nur eine Schätzung der tatsächlichen Anzahl ausgeführter Assembler Anweisungen.
  - Unsere Schätzung ist auch hier nur

    bis auf einen konstanten Faktor

    exakt.

Pseudocode

1. Was ist ein **Algorithmus**?

   Eine präzise definierte Rechenvorschrift, formuliert in „Pseudocode“.

2. Und was ist **Pseudocode**?

   Eine Mischung von Umgangssprache, Programmiersprache und mathematischer Notation.

   Typischerweise ist Pseudocode kompakter, aber gleichzeitig auch leichter zu verstehen als entsprechender Code einer Programmiersprache.

> Pseudocode folgt keinen klaren Regeln, aber natürlich muss eine nachfolgende Implementierung trivial sein!

Beispiele und Aufgaben zu Pseudocode auf der Vorlesungshomepage!

Laufzeitmessung und Asymptotik

Wie **skaliert** die Laufzeit unter **wachsender** Eingabelänge?

- Wir betrachten die **worst-case** Laufzeit in Abhängigkeit von der **Eingabelänge**.
  - Unsere Laufzeitbestimmung ist letztlich nur eine, bis auf einen konstanten Faktor exakte Schätzung.
  - Wir interessieren uns vor allen Dingen für die Laufzeit „großer“ Eingaben und „unterschlagen“ konstante Faktoren.

Wir erhalten eine von der jeweiligen Rechnerumgebung unabhängige Laufzeitanalyse, die das

Wachstumverhalten

der tatsächlichen Laufzeit verlässlich voraussagt.

Die asymptotische Notation

Wie schnell dominiert die Asymptotik?

Annahme: Ein einfacher Befehl benötigt 10^{-9} Sekunden.

n	n^2	n^3	n^{10}	2^n	$n!$
16	256	4.096	$\geq 10^{12}$	65536	$\geq 10^{13}$
32	1.024	32.768	$\geq 10^{15}$	$\geq 4 \cdot 10^9$	$\geq 10^{31}$
64	4.096	262.144	$\geq 10^{18}$	$\geq 6 \cdot 10^{19}$	
					mehr als
128	16.384	2.097.152	mehr als	mehr als	10^{14} Jahre
256	65.536	16.777.216	10 Jahre	600 Jahre	
512	262.144	134.217.728			
1024	1.048.576	$\geq 10^9$			
Million	$\geq 10^{12}$	$\geq 10^{18}$			
	mehr als	mehr als			
	15 Minuten	10 Jahre			

Die asymptotische Notation

$f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ seien Funktionen, die einer Eingabelänge $n \in \mathbb{N}$ eine nicht-negative Laufzeit $f(n)$, bzw. $g(n)$ zuweisen.

- Die Groß-Oh Notation: $f = O(g) \Leftrightarrow$ Es gibt eine positive Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass

  $$f(n) \leq c \cdot g(n)$$

  für alle $n \geq n_0$ gilt: f wächst höchstens so schnell wie g.
- $f = \Omega(g) \Leftrightarrow g = O(f)$: f wächst mindestens so schnell wie g.
- $f = \Theta(g) \Leftrightarrow f = O(g)$ und $g = O(f)$: f und g wachsen gleich schnell.
- Klein-Oh Notation: $f = o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$: f wächst langsamer als g.
- Klein-Omega: $f = \omega(g) \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$: f wächst schneller als g.

Grenzwerte und die Asymptotik

Grenzwerte

Grenzwerte sollten das Wachstum vorausssagen!

Der Grenzwert der Folge $\frac{f(n)}{g(n)}$ existiere und es sei $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.

- Wenn $c = 0$, dann ist $f = o(g)$. (f wächst langsamer als g.)
- Wenn $0 < c < \infty$, dann ist $f = \Theta(g)$. (f und g wachsen gleich schnell.)
- Wenn $c = \infty$, dann ist $f = \omega(g)$. (f wächst schneller als g.)
- Wenn $0 \leq c < \infty$, dann ist $f = O(g)$.
  (f wächst höchstens so schnell wie g.)
- Wenn $0 < c \leq \infty$, dann ist $f = \Omega(g)$.
  (f wächst mindestens so schnell wie g.)

Wachstum des Logarithmus

Für alle reelle Zahlen $a > 1$ und $b > 1$ gilt

$$\log_a n = \Theta(\log_b n).$$

Warum? Es gilt

$$\log_a(n) = (\log_b(a)) \cdot \log_b(n).$$

- Also folgt
$$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \log_b(a).$$
- Aber $a, b > 1$ und deshalb $0 < \log_b(a) < \infty$.
- Die Behauptung $\log_a n = \Theta(\log_b n)$ folgt.

$\log_a(n)$ und der natürliche Logarithmus $\ln(n)$ haben dasselbe Wachstum:
Ab jetzt arbeiten wir nur mit $\ln(n)$.

Rechnen mit Grenzwerten

$f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ seien Funktionen.

Die Operation $\circ$ bezeichne eine der Operationen $+, -$ oder $*$. Dann gilt

$$\lim_{n\to\infty} (f(n) \circ g(n)) = \lim_{n\to\infty} f(n) \circ \lim_{n\to\infty} g(n),$$

falls die beiden Grenzwerte auf der rechten Seite existieren und endlich sind. Und

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\lim_{n\to\infty} f(n)}{\lim_{n\to\infty} g(n)},$$

gilt, falls die beiden Grenzwerte auf der rechten Seite existieren, endlich sind und $\lim_{n\to\infty} g(n) \neq 0$ gilt.

- $\lim_{n\to\infty} \frac{n^3+n\cdot(n+1)/2}{n^3} = \lim_{n\to\infty} \frac{n^3}{n^3} + \lim_{n\to\infty} \frac{n\cdot(n+1)/2}{n^3} = 1.$
- Also ist $n^3 + \frac{n\cdot(n+1)}{2} = \Theta(n^3)$ und $n^3 + \frac{n\cdot(n+1)}{2}$ ist **kubisch**.

Die Regel von de l’Hospital

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$, falls der letzte Grenzwert existiert und falls $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) \in \{0, \infty\}$.

- $\lim_{n\to\infty} \ln(n) = \lim_{n\to\infty} n = \infty$ und $\lim_{n\to\infty} \frac{\ln'(n)}{n'} = \lim_{n\to\infty} \frac{1/n}{1} = 0$.
- $\lim_{n\to\infty} \frac{\ln(n)}{n} = 0$ und damit $\log_a(n) = o(n)$ für jedes $a > 1$.
- Der Logarithmus wächst langsamer als jede noch so kleine Potenz n^b (für $0 < b$): Siehe Tafel.

Unbeschränkt wachsende Funktionen

Die Funktionen $f : \mathbb{N} \to \mathbb{R}$ und $g : \mathbb{R} \to \mathbb{R}$ seien gegeben.

(a) f sei **monoton wachsend**. Dann gibt es eine Konstante $K \in \mathbb{N}$ mit $f(n) \leq K$ oder aber $\lim_{n\to\infty} f(n) = \infty$ gilt. Insbesondere ist $\mathbf{f = O(1)}$ oder $\mathbf{1 = o(f)}$.

(b) Für jedes $a > 1$ ist $1 = o(\log_a(n))$.

(c) Wenn $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, dann gilt $\lim_{n\to\infty} g(f(n)) = \infty$.

(d) Wenn $g = o(n)$ und $1 = o(f)$, dann ist

$$\lim_{n\to\infty} \frac{g(f(n))}{f(n)} = 0,$$

also ist $g(f(n)) = o(f(n))$. Wenn g „sublinear" ist und f unbeschränkt wächst, dann nimmt das Wachstum von $g \circ f$ ab.

Anwendungen: (Siehe Tafel.)

- $1 = o(\log_2 \log_2(n))$.
- $\log_2 \log_2(n) = o(\log_2(n))$.
- $1 = o(\log_2^{(k+1)}(n))$ und $\log_2^{(k+1)}(n) = o(\log_2^{(k)}(n))$ für jede natürliche Zahl k.

Eine Wachstums-Hierarchie

- $f_1(n) = 1$, dann ist $f_1 = o(\log_2 \log_2 n)$,

  haben wir schon eingesehen.
- $\log_2 \log_2 n = o(\log_2 n)$,

  haben wir schon eingesehen.
- $\log_2 n = \Theta(\log_a n)$ für jedes $a > 1$,

  haben wir schon eingesehen.
- $\log_2 n = o(n^b)$ für jedes $b > 0$,

  haben wir schon eingesehen.
- $n^b = o(n)$ und $n = o(n \cdot \log_a n)$ für jedes b mit $0 < b < 1$ und jedes $a > 1$,

  $\lim_{n\to\infty} \frac{n^b}{n} = \lim_{n\to\infty} \frac{1}{n^{1-b}} = 0$ und $\lim_{n\to\infty} \frac{n}{n \cdot \log_a n} = \lim_{n\to\infty} \frac{1}{\log_a n} = 0$.
- $n \cdot \log_a n = o(n^k)$ für jede reelle Zahl $k > 1$ und jedes $a > 1$.
- $n^k = o(a^n)$ für jedes $a > 1$,

  $n^k = a^{k \cdot \log_a n}$ und $\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} a^{k \cdot \log_a n - n} = 0$.
- und $a^n = o(n!)$ für jedes $a > 1$. (Siehe Tafel)

Was ist gut, was ist schlecht?

1. Die konstante Funktion wächst am langsamsten, gefolgt von $\log_a^{(k)}(n)$.
2. Selbst für sehr, sehr kleine Potenzen $0 < b < 1$ wächst der Logarithmus viel, viel langsamer als n^b.
   - Häufig auftretende Anfragen an eine Datenstruktur sollten in Zeit $O(\log_2 n)$ und besser noch in Zeit $O(1)$ laufen.
3. Dann kommen $f(n) = n$ und $g(n) = n \cdot \log_2 n$.
   - Jedes Programm, dass sich alle n Eingabedaten anschauen muss, muss mindestens $\Omega(n)$ Schritte investieren.
   - Die Laufzeitfunktion $n \cdot \log_2 n$ taucht für schnelle Divide & Conquer Algorithmen auf.
4. Laufzeiten n^k für $k > 1$ sind teuer.
   (Schon quadratische Laufzeiten tun weh).
5. Die Laufzeiten $h(n) = a^n$ sind **unbezahlbar** und für $h(n) = n!$ mehr als **unverschämt**: Die Funktionen a^n und $n!$ sind Stars in jedem Gruselkabinett und selbst kubische Laufzeiten lassen einen kalten Schauer über den Rücken laufen.

Laufzeitanalyse von

C++ Programmen und Pseudocode:

Zähle Anweisungen, die die Laufzeit **dominieren**

Analyse von C++ Programmen

Bestimme und zähle „**dominierende**“ Anweisungen.

- Die Anzahl ausgeführter dominierender Anweisungen sollte
  - **einfach zu bestimmen** sein und
  - die **asymptotische Anzahl aller ausgeführten Anweisungen** nicht verfälschen.
- Zum Beispiel: Zähle die Anzahl bestimmter ausgeführter elementarer Anweisungen wie etwa Zuweisungen und Auswahl-Anweisungen (if-else und switch).
  - Zähle weder iterative Anweisungen (for, while und do-while), noch Sprunganweisungen (break, continue, goto und return) oder Funktionsaufrufe.
  - Ist der so ermittelte Aufwand denn nicht zu klein?
    - Zähle **mindestens eine** ausgeführte elementare Anweisung in jedem Schleifendurchlauf oder Funktionsaufruf.
- In den meisten Fällen sind wir an der **worst-case Laufzeit** interessiert:
  - Bestimme für jede Eingabelänge n die **maximale** Laufzeit für eine Eingabe der Länge n.

Zuweisungen, Auswahl-Anweisungen und Schleifen

- Eine **Zuweisung** zu einer „einfachen“ Variablen ist einfach zu zählen, eine Zuweisung zu einer Array-Variablen ist mit der Länge des Arrays zu gewichten.
- Die Auswertung der Bedingung in einer **Auswahl-Anweisungen** ist nicht notwendigerweise zu zählen.
  - Aber die Laufzeit hängt jetzt vom Wert der Bedingung ab!
  - Häufig genügt die Bestimmung des Gesamtaufwands für den **längsten** der alternativen Anweisungsblöcke.
- **Schleifen**: Der Aufwand kann in jedem Schleifendurchlauf variieren!
  - Häufig genügt die Bestimmung der maximalen Anzahl ausführbarer Anweisungen innerhalb einer Schleife sowie die Anzahl der Schleifendurchläufe.

Beispiel: For-Schleifen

Die Matrizenmultiplikation $A \cdot B = C$:

```
for (i=0; i < n; i++)
    for (j=0; j < n; j++)
        {C[i][j] = 0;
        for (k=0; k < n ; k++)
            C[i][j] += A[i][k] * B[k][j]; }
```

- Zähle die Anzahl der Zuweisungen.
- Analysiere die geschachtelten for-Schleifen durch geschachtelte Summen:

$$\textbf{Laufzeit} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (1 + \sum_{k=0}^{n-1} 1) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (1 + n) = n^2 \cdot (1 + n).$$

Das Programm besitzt

kubische Laufzeit.

Beispiel: While Schleifen (1/2)

```
while (n >= 1)
    {n = n/2;
    Eine Menge von maximal c Anweisungen, die n nicht verändern }
```

- Zähle die höchstens $1 + c$ Anweisungen pro Schleifendurchlauf.
- $T(n)$ bezeichne die Laufzeit der while-Schleife für Parameter n. Wir erhalten die

  Rekursionsgleichung: $T(1) \leq 1 + c, \;\; T(n) \leq T(n/2) + 1 + c.$

- Die Laufzeit ist

  logarithmisch.

  Warum?
  - Höchstens $1 + c$ Anweisungen pro Schleifendurchlauf.
  - Und wieviele Iterationen? Na klar, $1 + \lfloor \log_2 n \rfloor$ viele.

Beispiel: While Schleifen (2/2)

```
while (n > 1)
    n = ( n & 1 ) ? 3*n + 1 : n/2;
```

- Was „tut“ diese while Schleife?
  - Wenn n ungerade ist, dann ersetze n durch $3 \cdot n + 1$.
  - Wenn n gerade ist, dann ersetze n durch $\frac{n}{2}$.
- Was ist die asymptotische Laufzeit in Abhängigkeit von der „Eingabelänge“ n?
  - Es ist bis heute nicht bekannt, ob die Schleife für jede Eingabe terminiert!
  - Die Laufzeit ist deshalb erst recht nicht bekannt.

Die Analyse der Laufzeit kann äußerst schwierig sein!

Algorithmenentwurf, Laufzeitmessung,

und asymptotische Analyse.

Wie passt das alles zusammen?

Ein Beispiel: Das Teilfolgenproblem

- Die Eingabe besteht aus n ganzen Zahlen $a_1, \ldots, a_n$.
- Definiere $f(i,j) = a_i + a_{i+1} + \cdots + a_j$ für $1 \leq i \leq j \leq n$.
- Das Ziel ist die Berechnung von

$$\max\{f(i,j) | 1 \leq i \leq j \leq n\},$$

  also die maximale Teilfolgensumme.

Anwendungbeispiel für Aktien: Berechne den größten Gewinn für ein Zeitintervall.

Algorithmen für das Teilfolgenproblem

Wir betrachten nur Algorithmen A, die ausschließlich

Additionen und/oder Vergleichsoperationen

auf den Daten ausführen.

- $\text{Additionen}_A(n) = \max_{a_1,\ldots a_n \in \mathbb{Z}}\{$ Anzahl Additionen von A für Eingabe $(a_1, \ldots, a_n)\}$
- $\text{Vergleiche}_A(n) = \max_{a_1,\ldots,a_n \in \mathbb{Z}}\{$ Anzahl Vergleiche von A für Eingabe $(a_1, \ldots, a_n)\}$.
- $\text{Zeit}_A(n) = \text{Additionen}_A(n) + \text{Vergleiche}_A(n)$ ist die worst-case Rechenzeit von Algorithmus A.

Das Teilfolgenproblem: Algorithmus A_1

```
Max = −∞;
for (i=1 ; i <= n; i++)
    for (j=i ; j <= n; j++)
        {Berechne f(i,j) mit j − i Additionen;
        Max = max{f(i,j), Max}; }
```

Wir benötigen $j - i$ Additionen zur Berechnung von $f(i,j)$. Also ist

$$\text{Additionen}_{A_1}(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} (j - i).$$

- Eine obere Schranke: $\text{Additionen}_{A_1}(n) \leq \sum_{i=1}^{n} \sum_{j=1}^{n} n = n^3$.
- Eine untere Schranke: $\text{Additionen}_{A_1}(n) \geq \sum_{i=1}^{n/4} \sum_{j=3n/4+1}^{n} \frac{n}{2}$ und $\text{Additionen}_{A_1}(n) \geq \frac{n}{4} \cdot \frac{n}{4} \cdot \frac{n}{2} = \frac{n^3}{32}$ folgt.

Die Laufzeit von A_1 ist **kubisch**:
Die Laufzeit nimmt um den Faktor acht zu, wenn sich die Eingabelänge verdoppelt. **:-((**

Das Teilfolgenproblem: Algorithmus A_2

```
Max = −∞;
for (i=1 ; i <= n; i++)
    { f(i, i − 1) = 0;
    for (j=i ; j <= n; j++)
        { f(i, j) = f(i, j − 1) + a_j;
        Max = max{f(i, j), Max}; }}
```

Wir benötigen genau so viele Additionen wie Vergleiche. Also ist

$$\begin{aligned} \text{Zeit}_{A_2(n)} &= \text{Additionen}_{A_2}(n) + \text{Vergleiche}_{A_2}(n) \\ &= \frac{n \cdot (n+1)}{2} + \frac{n \cdot (n+1)}{2} = n \cdot (n+1). \end{aligned}$$

$\lim_{n \to \infty} \frac{n \cdot (n+1)}{n^2} = 1$ und deshalb folgt $\text{Zeit}_{A_2}(n) = \Theta(n^2)$.

Die Laufzeit von A_2 ist **quadratisch** und wächst damit um den Faktor vier, wenn sich die Eingabelänge verdoppelt **:-((**

Zwischenfazit

- Mit Hilfe der asymptotischen Notation können wir sagen, dass
  - A_1 eine kubische Laufzeit und
  - A_2 eine quadratische Laufzeit besitzt.
- Es ist $\lim_{n\to\infty} \frac{n^2}{n^3} = 0$ und damit folgt $n^2 = o(n^3)$:
  A_2 ist wesentlich schneller als A_1.
- Aber es geht noch deutlich schneller!

Ein Divide & Conquer Ansatz

- Angenommen, wir kennen den maximalen $f(i,j)-$Wert $\text{opt}_{\text{links}}$ für $1 \leq i \leq j \leq \lceil \frac{n}{2} \rceil$ sowie den maximalen $f(i,j)$-Wert $\text{opt}_{\text{rechts}}$ für $\lceil \frac{n}{2} \rceil + 1 \leq i \leq j \leq n$.
- Welche Möglichkeiten gibt es für den maximalen $f(i,j)$-Wert opt für $1 \leq i \leq j \leq n$?

1. $\text{opt} = \text{opt}_{\text{links}}$ und die maximale Teilfolge liegt in linken Hälfte.
2. $\text{opt} = \text{opt}_{\text{rechts}}$ und die maximale Teilfolge liegt in rechten Hälfte.
3. $\text{opt} = \max\{f(i,j) \mid 1 \leq i \leq \lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil + 1 \leq j \leq n\}$:
   eine maximale Teilfolge beginnt in der linken und endet in der rechten Hälfte.

Das Teilfolgenproblem: Algorithmus A_3

$A_3(i,j)$ bestimmt den Wert einer maximalen Teilfolge für $a_i \dots, a_j$.

(1) Wenn $i = j$, dann gib a_i als Antwort aus.

(2) Ansonsten setzte mitte $= \lfloor \frac{i+j}{2} \rfloor$;

(3) $\text{opt}_1 = \max\{f(k, \text{mitte}) \mid i \leq k \leq \text{mitte}\}$;
$\text{opt}_2 = \max\{f(\text{mitte}+1, l) \mid \text{mitte}+1 \leq l \leq j\}$;

(4) $\text{opt} = \max\{A_3(i, \text{mitte}), A_3(\text{mitte}+1, j), \text{opt}_1 + \text{opt}_2\}$ wird als Antwort ausgegeben.

- $n = j - i + 1$ ist die Eingabelänge. n sei eine Zweierpotenz.
- Wie bestimmt man die Laufzeit $\text{Zeit}_{A_3}(n)$?
  - $\text{Zeit}_{A_3}(1) = 1$,
  - In Schritt (4) werden zwei **rekursive** Aufrufe für Eingabelänge $\frac{n}{2}$ mit Laufzeit jeweils $\text{Zeit}_{A_3}(\frac{n}{2})$ ausgeführt. Dazu kommen $t(n)$ weitere „**nicht-rekursive**“ Operationen aus (3) und (4). Also

$$\text{Zeit}_{A_3}(n) = 2 \cdot \text{Zeit}_{A_3}(\frac{n}{2}) + t(n).$$

Wie löst man Rekursionsgleichungen?

- Wie groß ist der „Overhead“ $t(n)$?
  - Um opt_1 und opt_2 zu berechnen genügen $\frac{n}{2} - 1 + \frac{n}{2} - 1 = n - 2$ Additionen.
  - Dieselbe Anzahl von Vergleichen wird benötigt.
  - Eine weitere Addition wird für $\text{opt}_1 + \text{opt}_2$ benötigt, zwei weitere Vergleiche fallen in Schritt (4) an.
  - $t(n) = 2 \cdot (n - 2) + 3 = 2n - 1$.
- Wir erhalten die Rekursionsgleichnungen:

$$\text{Zeit}_{A_3}(1) = 1$$
$$\text{Zeit}_{A_3}(n) = 2 \cdot Zeit_{A_3}\left(\frac{n}{2}\right) + 2n - 1.$$

- **Der allgemeinere Fall:**
  Ein rekursives Programm A mit Eingabelänge n besitze $\mathbf{a}$ rekursive Aufrufe mit Eingabelänge $\mathbf{\frac{n}{b}}$ und es werden $\mathbf{t(n)}$ nicht-rekursive Operationen ausgeführt.
  Die Rekursionsgleichung:

$$\text{Zeit}_A(n) = a \cdot \text{Zeit}_A\left(\frac{n}{b}\right) + t(n).$$

Das Mastertheorem

Die Rekursionsgleichung

$$T(1) = c, T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$$

ist zu lösen. n sei eine Potenz der Zahl $b > 1$ und $a \geq 1, c > 0$ gelte.

Der Baum der Rekursionsgleichung:

1. Wir sagen, dass die Wurzel `root` die Eingabelänge n hat.
   - Wenn $n = 1$, dann markiere `root` mit c und `root` wird zu einem Blatt.
   - Wenn $n > 1$, dann markiere `root` mit $t(n)$. `root` erhält a Kinder mit Eingabelänge n/b.
2. Sei v ein Knoten des Baums mit Eingabelänge m.
   - Wenn $m = 1$, dann markiere v mit c und v wird zu einem Blatt.
   - Wenn $m > 1$, dann markiere v mit $t(m)$ und v erhält a Kinder mit Eingabelänge m/b.

$T(n)$ stimmt überein mit der Summe aller Knotenmarkierungen

Wende vollständige Induktion an.

Der Baum der Rekursionsgleichung (2/2)

Die Rekursionsgleichung

$$T(1) = c, T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$$

ist zu lösen. n sei eine Potenz der Zahl $b > 1$ und $a \geq 1, c > 0$ gelte.

Die **Tiefe** eines Knotens v ist die Länge des Weges von der Wurzel zu v.

(a) Mit *vollständiger Induktion*: Der Baum hat in Tiefe d genau a^d Knoten.

(b) Die Tiefe des Baums ist $\log_b n$. Die Anzahl seiner Blätter ist

$$a^{\log_b n} = a^{(\log_b a) \cdot (\log_a n)} = a^{(\log_a n) \cdot (\log_b a)} = n^{\log_b a}.$$

Das Mastertheorem: Sei $\varepsilon > 0$.

1. Wenn $t(n) = O(n^{(\log_b a) - \varepsilon})$, dann ist $T(n) = \Theta(n^{\log_b a})$:
   Die Anzahl der Blätter dominiert in $T(n)$.
2. Wenn $t(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$:
   Gleichgewicht zwischen den $\log_b n$ Schichten des Baums.
3. Wenn $t(n) = \Omega(n^{(\log_b a) + \varepsilon})$, dann $T(n) = \Theta(t(n))$: Die Wurzel dominiert.

Das Mastertheorem: Die korrekte Formulierung

Die Rekursion

$$\mathbf{T(1) = c, T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)}$$

sei zu lösen. n ist eine Potenz der Zahl $b > 1$ und $a \geq 1$, $c > 0$ gelte.

(a) Wenn $t(n) = O\left(n^{(\log_b a) - \varepsilon}\right)$ für eine Konstante $\varepsilon > 0$, dann ist $T(n) = \Theta(n^{\log_b a})$.

(b) Wenn $t(n) = \Theta(n^{\log_b a})$, dann ist $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.

(c) Wenn $t(n) = \Omega\left(n^{(\log_b a) + \varepsilon}\right)$ für eine Konstante $\varepsilon > 0$ und $a \cdot t\left(\frac{n}{b}\right) \leq \alpha t(n)$ für eine Konstante $\alpha < 1$, dann $T(n) = \Theta(t(n))$.

Siehe Beweis des Mastertheorems im **Skript**.

Die Laufzeit von Algorithmus A_3

Wir hatten die Rekursionsgleichungen

$$\text{Zeit}_{A_3}(1) = 1, \text{Zeit}_{A_3}(n) = 2 \cdot \mathit{Zeit}_{A_3}\left(\frac{n}{2}\right) + 2n - 1$$

erhalten.

- Um das Mastertheorem anzuwenden, müssen wir $c = 1, a = 2, b = 2$ und $t(n) = 2n - 1$ setzen.
- In welchem Fall befinden wir uns?
  - Es ist $\log_b a = \log_2 2 = 1$ und $t(n) = \Theta(n^{\log_b a})$.
  - Fall (b) ist anzuwenden und liefert $\text{Zeit}_{A_3}(n) = \Theta(n \log_2 n)$.
- Algorithmus A_3 ist schneller als Algorithmus A_1, denn

$$\lim_{n\to\infty} \frac{\text{Zeit}_{A_3}(n)}{\text{Zeit}_{A_2}(n)} = \lim_{n\to\infty} \frac{n \log_2 n}{n^2} = \lim_{n\to\infty} \frac{\log_2 n}{n} = 0.$$

Was kann das Mastertheorem nicht?

Um das Mastertheorem anzuwenden, muss stets

dieselbe Anzahl a von Teilproblemen

mit

derselben Eingabelänge $\frac{n}{b}$

rekursiv aufgerufen werden: b darf sich während der Rekursion nicht ändern.

Das Mastertheorem ist nicht auf die Rekursion

$$T(1) = 1, T(n) = T(n-1) + n$$

anwendbar, weil b sich ändert. Gleiches gilt für die Rekursion

$$T(1) = 1, T(n) = 2 \cdot T(n-1) + 1,$$

die die Anzahl der Ringbewegungen in den Türmen von Hanoi zählt.