Algorithmen und Datenstrukturen 1

Sommersemester 2022

Herzlich willkommen!

Wer ist wer?

Wir: Professur für Algorithmen und Komplexität

- Martin Hoefer (Vorlesungen)
  R 115 – RMS 11–15, mhoefer@em.uni-frankfurt.de
- Tim Koglin, Marco Schmalhofer (Übungskoordination)
  R104a, R114 – RMS 11–15, algo122@cs.uni-frankfurt.de
- Timo Eisert, Niklas Fleischer, Lukas Geis, Marius Hagemann, Lukas Harren, Alexander Hengstmann, Melvin Kallmayer, Aura Sofia Lohr, Patrick Raphael Melnic, Julian Mende, Anton Micke, Niklas Oberender, Toprak Saricerci, Gerome Stiler, Jonas Strauch, Tolga Tel, Christine Thomas (Tutoren)

- algo.cs.uni-frankfurt.de
  → Sommer 2022 → Algorithmen und Datenstrukturen 1

Wer sind Sie?

Worum geht's?

Abstrakte Datentypen und Datenstrukturen

Ein **abstrakter Datentyp** ist eine Sammlung von Operationen auf einer Menge von Objekten. Eine **Datenstruktur** ist eine Implementation eines abstrakten Datentyps

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 „**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 sei ein abstrakter Datentyp. Entwerfe eine *möglichst effiziente* Datenstruktur.

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

  $\rightarrow$ Asymptotische Analyse von Laufzeit und Speicherplatzverbrauch

Worum geht's?

Wie können wir **effiziente Algorithmen** entwerfen? Wie können wir dabei *effiziente Datenstrukturen* geschickt einsetzen?

Algorithmen?

- Algorithmen für **Graphen**: Berechne ...
  alle Zusammenhangskomponenten eines Graphen
  einen minimalen Spannbaum eines Graphen
  einen kürzesten Weg von Knoten u nach Knoten v
- Generelle **Entwurfsmethoden** für Algorithmen:
  Greedy (z.B. zur Ablaufplanung)
  Divide & Conquer (z.B. zum Sortieren)
  Dynamische Programmierung (z.B. für Paarweises Alignment)

Effizienz?
$\rightarrow$ Asymptotische Analyse von Laufzeit und Speicherplatzverbrauch

Worauf wird aufgebaut?

(1) Diskrete Modellierung (bzw. Vorkurs):
- 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

(3) Lineare Algebra und Analysis:
- Logarithmen,
- Grenzwerte und
- asymptotische Notation.

Literatur (1/2)

- Skripte und Folien zur Vorlesung finden Sie auf der Webseite der Veranstaltung.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, “Introduction to Algorithms”, 2002.
- 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”, 2014.
- R. Sedgewick, “Algorithmen in C++”, 2002.
- J. Kleinberg, E. Tardos, “Algorithm Design”, 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 von “Algorithmen und Datenstrukturen 1”.

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

Organisatorisches

Übungsbetrieb

- Alle Details auf der **Webseite**!
- **Anmeldung** zu Übungen im AUGE-System bis Mittwoch, 13.04.
- Übungsblätter erscheinen i.d.R. **jeden Dienstag** auf der Webseite.
- Lösungen (als eine PDF-Datei) sind nach **einwöchiger Bearbeitungszeit** online über das SAP-System abzugeben.
- Nach Anmeldung für die Übungen über AUGE bekommen Sie per Email ihren **individuellen Abgabelink** für das SAP-System. Dieser Link ist das ganze Semester lang für Sie gültig.
- Übungsgruppen treffen sich ab übernächste Woche Montag, 25.04.
- Heute Ausgabe **0. Übungsblatt** (ohne Abgabe), Besprechung ab 25.04.
- Ausgabe **1. Übungsblatt**: Dienstag, 19.04. (Abgabe bis 26.04., Besprechung ab 02.05.)
- Übungsaufgaben sollten mit Anderen besprochen werden, aber Lösungen **müssen eigenständig aufgeschrieben** werden!

Klausur

- Hauptklausur: Montag, **08. August 2022**, ab 9:00 Uhr.
- Zweitklausur: Freitag, **14. Oktober 2022**, ab 9:00 Uhr.
- 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 ($y/10$ kaufmännisch gerundet).

- Die Veranstaltung ist **bestanden**, wenn die **Klausur bestanden ist** (i.d.R. wenn $x \geq 50$).
- Wenn die Klausur bestanden ist, hängt die **Note** von der **Gesamtpunktzahl** z ab.

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

- Sprechstunde vereinbaren per Email.
  Oder versuchen Sie es auf gut Glück (Raum 115, R.M.S. 11-15).

Helfen Sie uns durch

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

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

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.

Wiederholung? (1) demogr.

Mathematische Grundlagen:

Gesucht ist eine kurze Formel für $\sum_{i=1}^{n} i$

- (1) $\frac{n(n-1)}{2}$
- (2) $\frac{n(n+1)}{2}$
- (3) $\frac{n^2-1}{2}$
- (4) Hab ich mal irgendwann gelernt...
- (5) Zahlen als Buchstaben!?!?

Auflösung: (2) $\frac{n(n+1)}{2}$ (wie beweist man das?)

Die Summe der ersten n Zahlen

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

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

Wiederholung? (2) demogr.

Mathematische Grundlagen:

Gesucht ist eine kurze Formel für $\sum_{i=0}^{n} a^i$ mit $a \neq 1$.

- (1) $2 \cdot a^n$
- (2) $\frac{a^{n+2}-1}{n}$
- (3) $\frac{a^{n+1}-1}{a-1}$
- (4) $\frac{a \cdot n \cdot (n-1)}{2}$
- (5) Keine Ahnung...

Auflösung: (3) $\frac{a^{n+1}-1}{a-1}$ (wie beweist man das?)

Die geometrische Reihe

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \text{ falls } a \neq 1.$$

- Vollständige Induktion nach n:
  Induktionsanfang für $n = 0$: $\sum_{i=0}^{0} a^i = 1$ und $\frac{a^{0+1}-1}{a-1} = 1$. ✓
  Induktionsschritt von n auf $n+1$:
  - Wir können annehmen, dass $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$ gilt. Dann ist
  - $\sum_{i=0}^{n+1} a^i = \sum_{i=0}^{n} a^i + a^{n+1} = \frac{a^{n+1}-1}{a-1} + a^{n+1} = \frac{a^{n+1}-1+a^{n+2}-a^{n+1}}{a-1} = \frac{a^{n+2}-1}{a-1}$ und das war zu zeigen. ✓
- Ein direkter Beweis:

$$\begin{aligned}(a-1) \cdot \sum_{i=0}^{n} a^i &= a \cdot \sum_{i=0}^{n} a^i - \sum_{i=0}^{n} a^i \\ &= \sum_{i=1}^{n+1} a^i - \sum_{i=0}^{n} a^i = a^{n+1} - a^0 = a^{n+1} - 1\end{aligned}$$

und das war zu zeigen. ✓

Binomialkoeffizienten

$$\binom{n}{k} = \begin{cases} \frac{n\cdot(n-1)\cdots(n-k+1)}{1\cdot 2\cdots k} = \frac{n!}{k!\cdot(n-k)!} & \text{falls } 1 \leq k \leq n-1, \\ 1 & \text{falls } k = 0 \text{ oder } k = n. \end{cases}$$

- Sei S eine Menge von n Elementen. Dann hat S genau $\binom{n}{k}$ Teilmengen der Größe k.
- Binomischer Lehrsatz: Es gilt $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$. Also ist $\sum_{i=0}^{n} \binom{n}{i} = 2^n$: eine n-elementige Menge hat 2^n Teilmengen.
- Sei $Perm(S)$ die Menge aller Permutationen von S. Dann ist $|Perm(S)| = n!$

k-elementige Teilmengen der Menge $\{1, \ldots, n\}$

Wieviele k-elementige Teilmengen von $\{1, \ldots, n\}$ gibt es?
Wir führen eine Induktion nach n.

- Induktionsanfang für $n = 1$:
  - $\binom{1}{0} = \binom{1}{1} = 1$.
  - Es gibt genau eine Teilmenge von $\{1\}$ mit keinem Element sowie genau eine Teilmenge mit einem Element.
- Induktionsschritt: Wir wissen, dass $\{1, \ldots, n\}$ genau $\binom{n}{k}$ Teilmengen mit genau k Elementen hat.
  - Wieviele k-elementige Teilmengen von $\{1, \ldots, n, n+1\}$ besitzen das Element $n+1$ nicht?
    Nach Induktionsannahme genau $\binom{n}{k}$ Teilmengen.
  - Wieviele k-elementige Teilmengen von $\{1, \ldots, n, n+1\}$ besitzen das Element $n+1$?
    Nach Induktionsannahme genau $\binom{n}{k-1}$ Teilmengen.
  - Es ist $\binom{n}{k} + \binom{n}{k-1} = \frac{n!}{k!(n-k)!} + \frac{n!}{(k-1)!(n-(k-1))!} = \frac{n!(k-1)!(n-(k-1))! + n!k!(n-k)!}{k!(k-1)!(n-k)!(n-(k-1))!} = \frac{n!(n-(k-1))+n!k}{k!(n-(k-1))!} = \frac{n!(n+1)}{k!(n+1-k)!} = \frac{(n+1)!}{k!(n+1-k)!} = \binom{n+1}{k}$ und das war zu zeigen.

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: Ein 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

$$d = \mathit{oben} - \mathit{unten} + 1,$$

dass der Aufruf Binärsuche(unten,oben) das korrekte Ergebnis liefert.

- **Induktionsanfang** für $d = 0$: Es ist $d = \mathit{oben} - \mathit{unten} + 1 = 0$ $\Rightarrow \mathit{oben} < \mathit{unten}$.

  Binärsuche($\mathit{unten}, \mathit{oben}$) bricht ab mit der Antwort „x wurde nicht gefunden“. ✓
- **Induktionsschritt** $d \to d + 1$: Es ist $\mathit{oben} - \mathit{unten} + 1 = d + 1$.
  - Wenn $A[\mathit{mitte}] = x$, dann wird mit Erfolgsmeldung abgebrochen. ✓
  - Ansonsten, wenn $x < A[\mathit{mitte}]$, 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. ✓

Wiederholung? (3) demogr.

Wie groß ist die maximale Anzahl inspizierter Zellen in der Binärsuche in einem Array mit $n = 2^k - 1$ Zellen?

- (1) n
- (2) $n \cdot k$
- (3) $\lceil \sqrt{k} \rceil$
- (4) k
- (5) Nur Kleingeister suchen, das Genie beherrscht das Chaos!

Auflösung: (4) k (wie beweist man das?)

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: $\frac{n-1}{2} = \frac{2^k-2}{2} = 2^{k-1} - 1$.
2. Wir zeigen $T(2^k - 1) = k$ mit vollständiger Induktion nach k.
   Induktionsanfang: $T(2^0 - 1) = T(0) = 0$. ✓
   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: Lineare Suche ist exponentiell langsamer als Binärsuche.

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 $a^z = x$.

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

(b) $\log_a(x \cdot y) = \log_a x + \log_a y$.

(c) $\log_a(x^y) = y \cdot \log_a(x)$.

(d) $\log_a x = (\log_a b) \cdot (\log_b x)$. 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 (d).
- $4^{\log_2 n} = 4^{(\log_2 4)\cdot(\log_4 n)} = (4^{\log_4 n})^{\log_2 4} = n^2$ mit (a).

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

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

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

$$\text{die Anzahl } n \text{ der Schlüssel}$$

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!

Laufzeit: Skalierung bei wachsender Eingabelänge

Wie **skaliert** die Laufzeit eines Algorithmus oder einer Datenstruktur wenn die Eingabelänge **wächst**?

- Eine exakte Laufzeitbestimmung für alle Eingaben ist oft schwierig.
- 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

Wachstumsverhalten

der tatsächlichen Laufzeit verlässlich voraussagt.

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: $\text{Additionen}_{A_1}(n) \geq \sum_{i=1}^{n/4} \sum_{j=3n/4+1}^{n} \frac{n}{2}$ (warum??) , also $\text{Additionen}_{A_1}(n) \geq \frac{n}{4} \cdot \frac{n}{4} \cdot \frac{n}{2} = \frac{n^3}{32}$ folgt.

Die worst-case Laufzeit von A_1

- Wieviele Vergleiche werden für die n Zahlen $a_1, \ldots, a_n$ ausgeführt?

$$\begin{aligned} \text{Vergleiche}_{A_1}(n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \sum_{i=1}^{n} (n - i + 1) \\ &= n + (n-1) + \cdots + 1 \\ &= \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}. \end{aligned}$$

- Und die worst-case Laufzeit?
  - $\text{Zeit}_{A_1}(n) = \text{Additionen}_{A_1}(n) + \text{Vergleiche}_{A_1}(n)$.
  - $\frac{n^3}{32} \leq \text{Additionen}_{A_1}(n) \leq n^3$. Also ist

$$\frac{n^3}{32} + \frac{n \cdot (n+1)}{2} \leq \text{Zeit}_{A_1}(n) \leq n^3 + \frac{n \cdot (n+1)}{2}.$$

Die Laufzeit von A_1 ist **kubisch :-((**

Die asymptotische Notation

Die asymptotische Notation (WICHTIG!!!)

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

Die Laufzeit von A_1 ist kubisch

Wir müssen $\text{Zeit}_{A_1}(n) = O(n^3)$ und $n^3 = O(\text{Zeit}_{A_1}(n))$ zeigen.

- Warum gilt $\text{Zeit}_{A_1}(n) = O(n^3)$?
  - Wir wissen $\text{Zeit}_{A_1}(n) \leq n^3 + \frac{n \cdot (n+1)}{2}$.
  - Also ist $\text{Zeit}_{A_1}(n) \leq n^3 + \frac{n \cdot 2n}{2} \leq n^3 + \frac{2n^3}{2} = 2 \cdot n^3$.
  - $\text{Zeit}_{A_1}(n) = O(n^3)$: Setze $c = 2$ und $n_0 = 0$.
- Warum gilt $n^3 = O(\text{Zeit}_{A_1}(n))$?
  - Wir wissen $\frac{n^3}{32} + \frac{n \cdot (n+1)}{2} \leq \text{Zeit}_{A_1}(n)$.
  - Also ist $\frac{n^3}{32} \leq \text{Zeit}_{A_1}(n)$ und deshalb folgt $n^3 \leq 32 \cdot \text{Zeit}_{A_1}(n)$.
  - $n^3 = O(\text{Zeit}_{A_1}(n))$: Setze $c = 32$ und $n_0 = 0$.

$\text{Zeit}_{A_1}(n) = \Theta(n^3)$: Die Laufzeit wächst (ungefähr) um den Faktor 8, wenn wir die Eingabelänge verdoppeln.

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			

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

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

- Also folgt

$$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \log_a(b).$$

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

$\log_a(n)$ und der Logarithmus $\log_2(n)$ zur Basis 2 haben dasselbe Wachstum: Ab jetzt arbeiten wir nur mit $\log_2(n)$.

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

Es gilt

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

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.

Das Integralkriterium

Die Funktion $f : \mathbb{R} \to \mathbb{R}$ sei gegeben.
(a) Wenn f monoton wachsend ist, dann gilt

$$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)dx.$$

(b) Wenn f monoton fallend ist, dann gilt

$$\int_1^{n+1} f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_0^n f(x)dx.$$

Zwei wichtige Konsequenzen:

- $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln(n))$.
- $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$, falls $k \geq 0$.

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 $a > 1$.

Asymptotische Notation (1) demogr.

Seien $f(n) = n^{1.1} / \log n$, $g(n) = n \log^3 n$, $h(n) = 2^{log_2 n}$.

Ordne die Funktionen nach asympt. Wachstum.

- (1) f, g, h
- (2) f, h, g
- (3) g, f, h
- (4) g, h, f
- (5) h, f, g
- (6) h, g, f

Auflösung: (6) h, g, f

Asymptotische Notation (2) multiple choice

Sei $f : \mathbb{N}_{>0} \to \mathbb{N}_{>0}$. Was gilt dann?

- (1) $f(n) + 10 = O(f(n))$
- (2) $f(n + 10) = O(f(n))$

Auflösung: (1) $f(n) + 10 = O(f(n))$.

Asymptotische Notation (3) multiple choice

Es seien $f(n) = 0.3 \cdot \sqrt{n} - \log n$ und $g(n) = 10 \cdot n^{0.2}$. Was gilt dann?

- (1) $f(n) = O(g(n))$
- (2) $f(n) = o(g(n))$
- (3) $f(n) = \Omega(g(n))$
- (4) $f(n) = \omega(g(n))$
- (5) $f(n) = \Theta(g(n))$

Auflösung: (3) & (4)

Asymptotische Notation (4) multiple choice

Es seien $f(n) = n \log n + 4 \cdot n^2$ und $g(n) = \sum_{i=n/2}^{n} i$. Was gilt dann?

- (1) $f(n) = O(g(n))$
- (2) $f(n) = o(g(n))$
- (3) $f(n) = \Omega(g(n))$
- (4) $f(n) = \omega(g(n))$
- (5) $f(n) = \Theta(g(n))$

Auflösung: (1) & (3) & (5)

Asymptotische Notation (5) multiple choice

Es seien $f(n) = |\sin n| + |\cos n|$ und $g(n) = \sqrt{\sum_{i=1}^{n} 1/i}$. Was gilt dann?

- (1) $f(n) = O(g(n))$
- (2) $f(n) = o(g(n))$
- (3) $f(n) = \Omega(g(n))$
- (4) $f(n) = \omega(g(n))$
- (5) $f(n) = \Theta(g(n))$

Auflösung: (1) & (2)

Algorithmenentwurf, Laufzeitmessung,

und asymptotische Analyse.

Wie passt das alles zusammen?

Zurück zum Teilfolgenproblem: Algorithmus A_2

Max $= -\infty$;
`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 $a_i \ldots, a_j$.

(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

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

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:

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

- **Der allgemeinere Fall:**
  Ein rekursives Programm A mit Eingabelänge n besitze **a** rekursive Aufrufe mit Eingabelänge $\frac{\mathbf{n}}{\mathbf{b}}$ 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

Der Baum der Rekursionsgleichung

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.
(kann man mit vollständiger Induktion zeigen).

Expansion und Vermutung

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.

Wir expandieren die Rekursionsgleichung und erhalten

$$\begin{aligned}
T(n) &= a \cdot T\left(\frac{n}{b}\right) + t(n) \\
&= a\left(a \cdot T\left(\frac{n}{b^2}\right) + t\left(\frac{n}{b}\right)\right) + t(n) \\
&= a^2 \cdot T\left(\frac{n}{b^2}\right) + a \cdot t\left(\frac{n}{b}\right) + t(n) \\
&= a^3 \cdot T\left(\frac{n}{b^3}\right) + a^2 \cdot t\left(\frac{n}{b^2}\right) + a \cdot t\left(\frac{n}{b}\right) + t(n) = \cdots \\
&\stackrel{?}{=} a^k \cdot T\left(\frac{n}{b^k}\right) + a^{k-1} \cdot t\left(\frac{n}{b^{k-1}}\right) + \cdots + a \cdot t\left(\frac{n}{b}\right) + t(n).
\end{aligned}$$

Man kann die Vermutung $\stackrel{?}{=}$ mit vollständiger Induktion beweisen.

Eliminierung rekursiver Terme

Setze $k = \log_b n$. Dann ist $b^k = n$ und $T(\frac{n}{b^k}) = T(1) = c$.

$$
\begin{aligned}
T(n) &= a^k \cdot T\left(\frac{n}{b^k}\right) + a^{k-1} \cdot t\left(\frac{n}{b^{k-1}}\right) + \cdots + a \cdot t\left(\frac{n}{b}\right) + t(n) \\
&= a^{\log_b n} \cdot T(1) + \sum_{j=0}^{k-1} a^j \cdot t\left(\frac{n}{b^j}\right) \\
&= a^{\log_b n} \cdot c + \sum_{j=0}^{\log_b n - 1} a^j \cdot t\left(\frac{n}{b^j}\right) = n^{\log_b a} \cdot c + \sum_{j=0}^{\log_b n - 1} a^j \cdot t\left(\frac{n}{b^j}\right).
\end{aligned}
$$

Eine Beobachtung:

Wenn $t(n) \leq d \cdot n^{\log_b(a)-\varepsilon}$ für Konstanten $\varepsilon > 0$ und $d > 0$, dann dominiert der erste Summand $n^{\log_b a} \cdot c$. Warum?

Das Mastertheorem: Formulierung

Die Rekursion

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

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 $T(n) = \Theta(n^{\log_b a})$.
Die Blätter im Rekursionsbaum dominieren.

(b) Wenn $t(n) = \Theta(n^{\log_b a})$, dann ist $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.
Gleichgewicht zwischen den Schichten im Rekursionsbaum.

(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 $T(n) = \Theta(t(n))$.
Die Wurzel im Rekursionsbaum dominiert.

Siehe Beweis des Mastertheorems im **Skript**.

Die Laufzeit von Algorithmus A_3

Wir hatten die Rekursionsgleichungen

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

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_2, denn

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

Und jetzt alle... demogr.

Die Rekursion $T(1) = c$, $T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$ 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 $a \cdot t\left(\frac{n}{b}\right) \leq \alpha \cdot t(n)$ für eine Konstante $\alpha < 1$, dann $T(n) = \Theta(t(n))$.

Lösung für $T(1) = \Theta(1)$, $T(n) = 4 \cdot T(n/2) + \Theta(n)$?

- (1) $T(n) = \Theta(n^2 \cdot \log^4 n)$
- (2) $T(n) = \Theta(n^2)$ $\surd$
- (3) $T(n) = \Theta(n \cdot \log n)$
- (4) $T(n) = \Theta(n \cdot \log^2 n)$
- (5) $T(n) = \Theta(n^4)$

Was kann das Mastertheorem nicht?

Um das Mastertheorem anzuwenden, muss stets

$$\text{dieselbe Anzahl } a \text{ von Teilproblemen}$$

mit

$$\text{derselben Eingabelänge } \frac{n}{b}$$

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.

Das Teilfolgenproblem: Algorithmus A_4

$$\text{Max}_k = \max\{f(i,j) | 1 \leq i \leq j \leq k\}$$

ist der Wert einer optimalen Teilfolge für die ersten k Folgenelemente.

Wie lässt sich Max_{k+1} effizient berechnen?

- Setze $\text{Max}^*_k = \max\{f(i,k) | 1 \leq i \leq k\}$.
- Dann ist $\text{Max}_{k+1} = \max\{\text{Max}_k, \text{Max}^*_{k+1}\}$. Warum?
  - Fall 1: Eine optimale Teilfolge enthält das Folgenelement a_{k+1} **nicht**. Dann ist $\text{Max}_{k+1} = \text{Max}_k$.
  - Fall 2: Eine optimale Teilfolge enthält das Folgenelement a_{k+1}. Dann ist $\text{Max}_{k+1} = \text{Max}^*_{k+1}$.
- $\text{Max}^*_{k+1} = \max\{\text{Max}^*_k + a_{k+1}, a_{k+1}\}$. Warum?
  - Fall 1: Eine optimale, mit dem $k+1$sten Folgenelement endende Folge enthält nicht nur a_{k+1}. Dann ist $\text{Max}^*_{k+1} = \text{Max}^*_k + a_{k+1}$.
  - Fall 2: Sonst ist $\text{Max}^*_{k+1} = a_{k+1}$.

Algorithmus A_4

(1) $\text{Max}_1 = \text{Max}_1^* = a_1$.

(2) Für $k = 1, \ldots, n-1$ setze
$$\text{Max}_{k+1}^* = \max\{\text{Max}_k^* + a_{k+1}, a_{k+1}\} \text{ und}$$
$$\text{Max}_{k+1} = \max\{\text{Max}_k, \text{Max}_{k+1}^*\}.$$

(3) Max_n wird ausgegeben.

- Pro Schleifendurchlauf: zwei Vergleiche und eine Addition.
  Also insgesamt $2(n-1)$ Vergleiche und $n-1$ Additionen.
- $\text{Zeit}_{A_4}(n) = 3(n-1)$ und A_4 hat lineare Laufzeit, da $\text{Zeit}_{A_4} = \Theta(n)$.
- A_4 ist schneller als A_3, denn $\lim\limits_{n\to\infty} \frac{n}{n \log_2 n} = 0$.

Wachstumsverhalten in der Laufzeitanalyse:

Zähle Anweisungen, die die Laufzeit **dominieren**

Analyse von C++ Programmen

- Zähle **ausgeführte elementare Anweisungen** wie etwa Zuweisungen und Auswahl-Anweisungen (if-else und switch).
  - Weder iterative Anweisungen (for, while und do-while) noch Sprunganweisungen (break, continue, goto und return) oder Funktionsaufrufe sind zu zählen.
- Ist der so ermittelte Aufwand denn nicht zu klein?
  - Zähle **mindestens eine** ausgeführte elementare Anweisung in jedem Schleifendurchlauf oder Funktionsaufruf.
  - Für vernünftige Programme wird die asymptotische Laufzeit korrekt ermittelt.
- 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.
- Der Aufwand kann in jedem Durchlauf einer **Schleife** 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;
    Maximal c Anweisungen, die n nicht verändern }
```

- $T(n)$ bezeichne die Laufzeit der while-Schleife für Parameter n. Wir erhalten die Rekursionsgleichung:

$$T(1) \leq 1 + c, \text{ und } T(n) \leq T(n/2) + 1 + c.$$

- Wende das Mastertheorem an:
  - $a = 1$, $b = 2$ und $t(n) \leq 1 + c$.
  - Es ist $t(n) = \Theta(1) = \Theta(n^{\log_b a})$.
  - Fall 2 ist anzuwenden und $T(n) = \Theta(\log_2 n)$ folgt.

- Die Laufzeit ist

logarithmisch.

Warum?

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

While-Schleifen demogr.

```
while (n >= 12)
    { n = √n;
    Maximal c Anweisungen, die n nicht verändern }
```

Wie hoch ist die Laufzeit?

- (1) $\Theta(\sqrt{n}\log n)$
- (2) $\Theta(\sqrt{n})$
- (3) $\Theta(\log n)$
- (4) $\Theta(\sqrt{\log(n)})$
- (5) $\Theta(\log(\log(n))$

Auflösung: (5) $\Theta(\log(\log(n))$

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!