# Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

# Übersicht

- Administratives
- Einleitung
- Ein einführendes Beispiel

# Administratives

- Dozent
  - Peppo Brambilla
  - Systemadministrator INF
- Vorlesungsassistent
  - Tiziano Portenier
  - Doktorand in der Computer Graphics Group
- Hilfsassitenten
  - Ramona Beck
  - Raffael Hertle

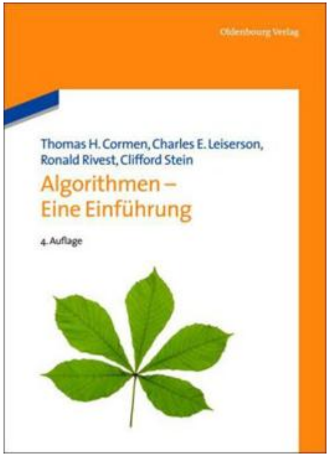
# Ilias

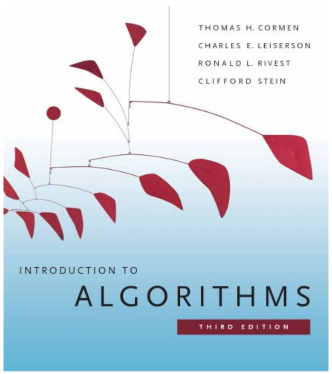
- Unter Phil. nat., Informatik, 2. Semester
  Repository > Philosophisch-naturwissenschaftliche Fakultät > Informatik > FS2018 > Vorlesung > 2409-FS2018-0: Datenstrukturen und Algorithmen

- Vorlesungsfolien
- Übungsaufgaben
  - Werden nicht gedruckt abgegeben
- Scans des Lehrbuchs (PDF)
- Forum für alle Fragen bezüglich Organisation, Stoff, Übungen

# Lehrbuch

- Obligatorisches Lehrbuch „Algorithmen- Eine Einführung“, Cormen et al.
  - Oldenbourg, 4. Auflage 2013
  - ISBN: 978-3-486-74861-1
  - Erhältlich bei Bugeno http://www.bugeno-unibe.ch
  - Scans (PDF) auf Ilias verfügbar
- Englische Version „Introduction to algorithms“, Third Edition
  - ISBN: 978-0-262-53305-8
- Lesematerial zu jeder Vorlesung auf der Vorlesungswebpage
  - Material im Buch gilt als Prüfungsstoff, ausser explizit erwähnte Ausnahmen



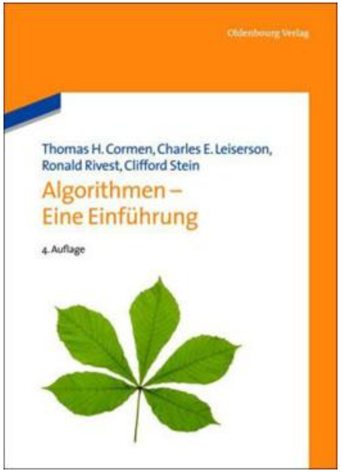



# Vorkenntnisse

- Grundlagen von Java
  - Auf der Stufe der Vorlesung Programmieren 1
  - Programmkonstrukte (Schleifen, Bedingungen, Datentypen, Arrays, Klassen)
  - Rekursion
  - Handhabung einer Programmierumgebung (Kommandozeilen Werkzeuge, Eclipse)
- Resourcen: P1 Lehrbuch (Java Software Solutions: Foundations of Program Design)

# Vorkenntnisse

- Mathematik
  - Funktionsbegriff
  - Polynome
  - Logarithmus
  - Exponentialfunktion
  - Summenzeichen, Summenformeln
  - Produktzeichen
  - Beweise mit vollständiger Induktion
- Resourcen: DA Lehrbuch, Kapitel 3.2, Anhänge



# Übungen

- Übungsstunde: Donnerstag, 16:15-17:00
- Wöchentliche Übungen
  - Abgabe jeweils in der Übungsstunde (bis 16:15)
  - Jede Übung gibt 10 Punkte
  - Programmierübungen bitte elektronisch und in Papierform abgeben
  - Verspätete Abgabe wird nicht akzeptiert
- **Zulassung zur Prüfung: Durchschnittlich 70% der maximalen Punkte erreicht.**

# Übungen

- Abgabe einzeln oder in Zweiergruppen
- Kein Kopieren der Abgaben
  - Jeder Studierende muss seine Lösung selber aufschreiben und erklären können
  - Wir behalten uns stichprobenweise individuelle Gespräche zu Übungen vor
- Bei Verstoss gegen diese Regeln wird die Übung als nicht gelöst gewertet

# Pool Stunde

- Wenn praktische Übungen anstehen
- Montag, 17:15-18:00, ExWi A94
- Primär für Studierende, die bei den Programmierübungen noch Mühe haben

# Prüfung

- Donnerstag, 7. Juni 2018, 16-18 Uhr
- Fragen ähnlich wie in Übungen
  - Bearbeiten der Übungen ist beste Prüfungs-vorbereitung
- Wiederholungsmöglichkeiten
  - Prüfung innerhalb von acht Monaten nach Vorlesung wiederholen
  - Ganze Vorlesung inkl. Übungen im nächsten Jahr wiederholen

# Übersicht

- Administratives
- Einleitung
- Ein einführendes Beispiel

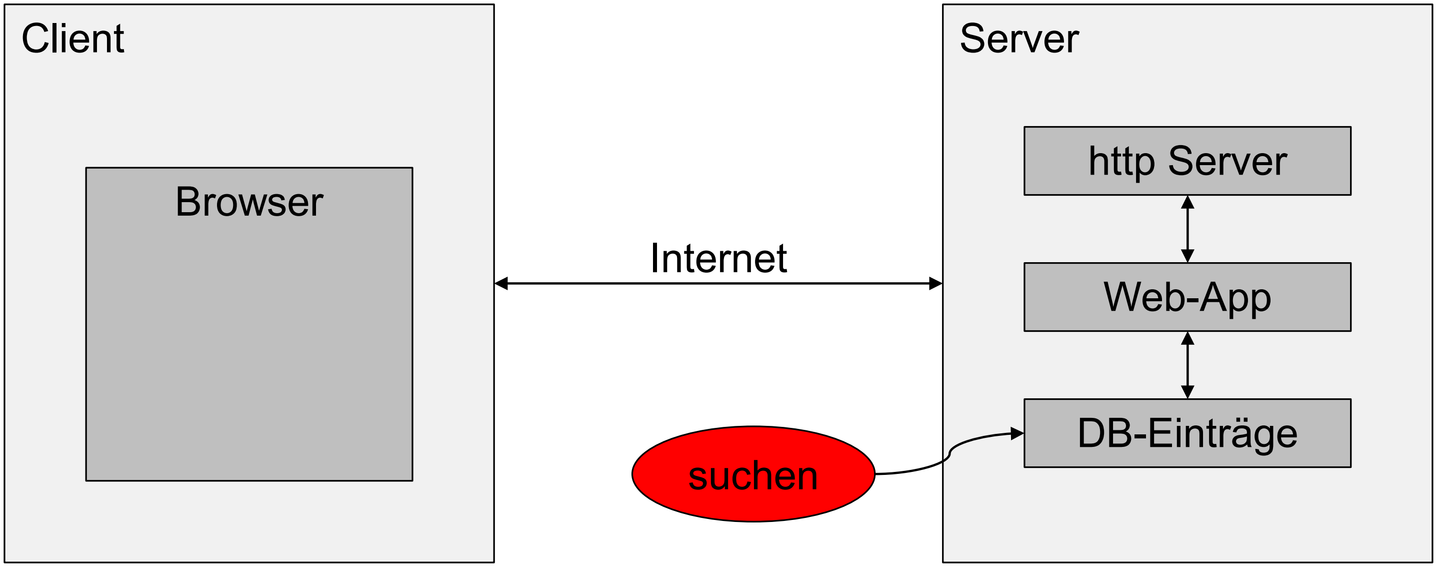
# Einleitung

- Positionierung von Datenstrukturen & Algorithmen in der Informatik
- Algorithmus-Begriff
- Analyse von Algorithmen
- Lernziele

# Datenstrukturen & Algorithmen

- Programmieren „im Kleinen“
  - Fokus auf funktionelle Bausteine, die in grösseren Systemen universell einsetzbar sind

**Beispiel: Internet Telefonverzeichnis**



# „Kleine, wohl definierte Probleme"

## Beispiele und Anwendungen

**Anwendungen**

- Telefonbuch
- Google Maps
- MP3-Player
- E-Banking
- DNA Sequenz
- 3D Grafik

**Algorithmus**

- Sortieren / Suchen
- Kürzester Pfad
- Datenkomprimierung
- Verschlüsselung
- String Matching
- Dreiecke Zeichnen

# Algorithmus

- „Präzise Beschreibung eines Lösungsverfahrens für ein Problem“
- Wort kommt von Al-Chwarizmi, persischer Astronom und Mathematiker, ca. 783-850
  http://en.wikipedia.org/wiki/Muhammad_ibn_Mūsā_al-Khwārizmī

*Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt. [Wikipedia]*

# Algorithmus

- Beschreibung durch
  - Programm in beliebiger Programmiersprache
  - Pseudo-Code
  - Text
- Beschreibung muss präzis genug sein, so dass sie einfach in ein Computerprogramm umformuliert werden kann
- Fragestellung: Analyse von Algorithmen

  Wie kann Effizienz und Korrektheit von Computerprogrammen analysiert werden?

# Andere Eigenschaften

- Speicherplatz
- Anwenderfreundlichkeit
- Ausführzeit
- Sicherheit
- Kosten
- Erweiterbarkeit
- Modularität

# Effizienz

- Effizienz entscheidet oft, ob ein Problem praktisch lösbar ist oder wo Grenzen der Lösbarkeit liegen
- Effizienz ist oft nötig um alle anderen gewünschten Eigenschaften zu erreichen
  - Effizienz ist „Währung“, mit der man für andere Eigenschaften bezahlt

# Lernziele

- Verstehen und Analysieren einiger der wichtigsten Algorithmen in der Informatik
- Verstehen der Grundbegriffe der Analyse von Algorithmen (Effizienz, Korrektheit)
- Verstehen und Anwenden von Entwurfsstrategien für Algorithmen
- Bewusstsein der Relevanz effizienter Algorithmen für Entwicklung von Softwaresystemen in der Praxis

# Übersicht

- Administratives
- Einleitung
- Ein einführendes Beispiel

# Ein einführendes Beispiel

**Sortieren**

- Eines der wichtigsten Probleme, das effizient von Computer gelöst werden kann
- 25% aller Rechenzeit im kommerziellen Bereich entfällt auf das Sortieren von Daten

# Ein einführendes Beispiel

- Einfacher Algorithmus: Sortieren durch Einfügen

  → Insertion Sort
- Asymptotische Analyse
- Verbesserung: Sortieren durch Mischen

  → Merge Sort
- Analyse mittels Rekursionsgleichungen

# Sortieren

- Eingabe: Zahlen $\langle a_1, a_2, \dots, a_n \rangle$
- Ausgabe: Permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Beispiel
  - Eingabe: 8 2 4 9 3 6
  - Ausgabe: 2 3 4 6 8 9

# Sortieren durch Einfügen (insertion sort)

key = Zwischenspeichern

- Pseudocode

Array

INSERTION-SORT(A) (Anfang bei 1 statt 0)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

# Beispiel

i j key

8 2 4 9 3 6 2

i j key

2 8 4 9 3 6 2

i i i j ... key

# Laufzeitanalyse

- Laufzeit: „Anzahl Schritte (Operationen), die der Algorithmus abarbeitet"
- Laufzeit hängt von der Eingabe ab
  - Sortieren einer vorsortierten Eingabe ist schneller, braucht weniger Operationen

    Bei einer vorsortierten Menge wird die while Schleife nie ausgeführt also deutlich schneller
- Wir interessieren uns für Laufzeit abhängig von der Länge $n$ der Eingabe
  - Kürzere Eingaben werden schneller sortiert als längere

# Laufzeitanalyse

- Bezeichnen Laufzeit für Eingabe der Länge $n$ mit $T(n)$
- Worst-case (sehr nützliche Analyse)
  - $T(n) =$ längstmögliche Laufzeit für Eingabelänge $n$
- Average-case (manchmal nützlich)
  - $T(n) =$ erwartete Laufzeit für Eingabelänge $n$
  - Braucht Annahme der statistischen Verteilung der Eingaben
- Best-case (irreführende Analyse)

# Worst-case Analyse

- Wie lange dauert Sortieren durch Einfügen einer Sequenz der Länge $n$ höchstens?
- Von Computer zu Computer verschieden
- Beobachtung: Laufzeit verschiedener Algorithmen variiert stark, viel stärker als Geschwindigkeitsunterschied verschiedener Computer

# Worst-case Analyse

**Ansatz**

- Ignoriere maschinenabhängige Faktoren
- Analysiere Laufzeitverhalten wenn Länge der Eingabe gross wird
- Asymptotische Analyse $T(n), n \to \infty$
  - Wie schnell wächst $T(n)$, wenn $n$ gegen unendlich geht?
  - Proportional zu $n$ (linear)?
  - Oder schneller, langsamer?

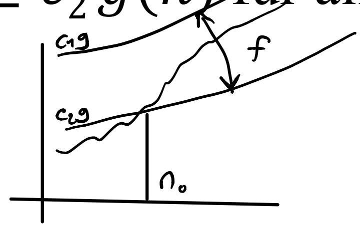
# Vereinfachtes Maschinenmodell

- RAM Modell (random-access machine)
- Instruktionen ähnlich wie in realen Computern
  - Arithmetik (addieren, subtrahieren, etc.)
  - Datenzugriff (laden, speichern, kopieren)
  - Kontrollinstruktionen (bedingte Ausführung, Prozeduraufrufe)
- Vereinfachende Annahme: jede Operation dauert eine konstante Zeit
  - Im Allgemeinen: jede Linie Pseudocode braucht konstante Zeit

# Θ-Notation

- Mathematische Beschreibung für asymptotisches Verhalten von Funktionen
- Intuitiv: $f$ wächst gleich schnell wie $g$

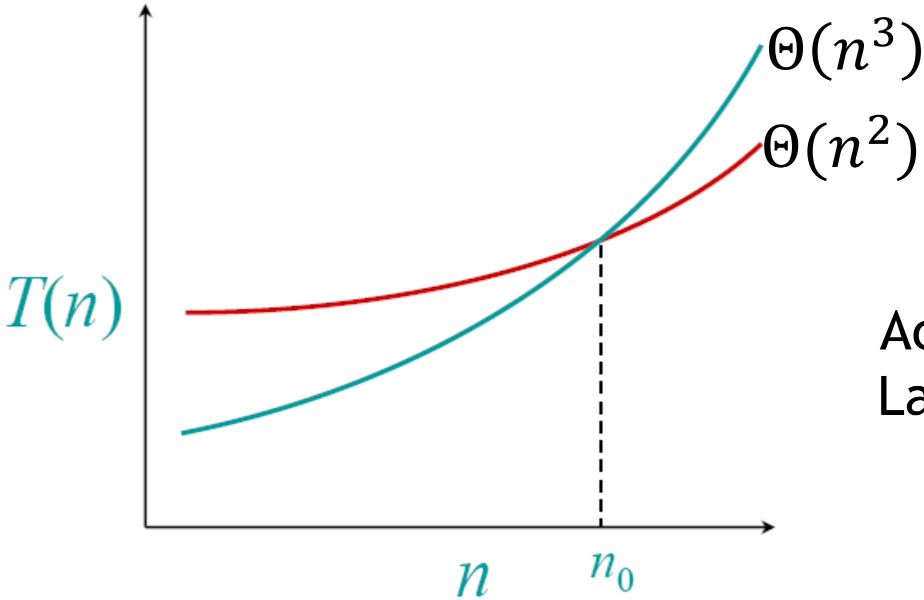
$$\Theta(g(n)) = \{f(n) : \text{es existieren positive Konstanten } c_1, c_2 \text{ und } n_0, \text{ so dass } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ für alle } n \geq n_0\}$$




- Praktisch
  - Ignoriere Terme niedriger Ordnung und Konstanten
  - Schreibweise: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$
  - „Wächst gleich schnell wie $n^3$“

# Asymptotisches Verhalten

- Für grosse $n$ (grösser als $n_0$), Laufzeit von $\Theta(n^2)$ ist immer schneller als Laufzeit von $\Theta(n^3)$
- Unabhängig von Computer, Programmiersprache, etc.




Achtung: Asymptotische Laufzeit ist nicht immer einziges Kriterium in der Praxis

# Sortieren durch Einfügen

- Worst case: Eingabe umgekehrt sortiert

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

$\left( \sum_{j=2}^{n} i = \frac{n(n+1)}{2} \right.$ ist etwas im Quadrat$\left. \right)$

- Average case: alle Permutationen gleich wahrscheinlich

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

$\frac{n(n+1)}{4}$ immer noch quadratisch

- Gilt Sortieren durch Einfügen als schneller Sortieralgorithmus?
  - Akzeptabel für kleine $n$ (wenige hundert)
  - Nicht akzeptabel für grosse $n$

# Divide-and-conquer Strategie

- „Teile und beherrsche“
  - Rekursive Strategie
  - Führt zu effizienten Algorithmen für viele Probleme
- Divide: Aufteilen des Problems in kleinere Probleme
- Conquer: Löse die Teilprobleme rekursiv. Falls die Probleme klein genug sind, löse sie direkt.
- Combine: Füge die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems zusammen

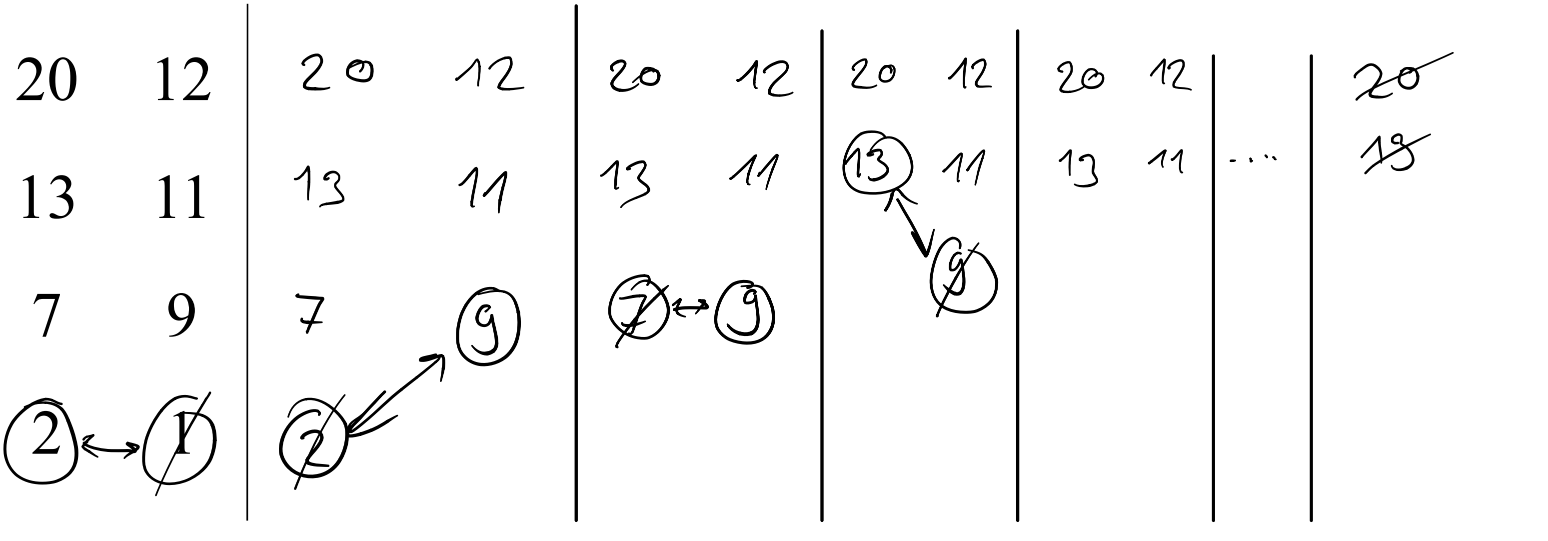
# Sortieren durch Mischen (merge sort)

**Merge−Sort** $A[1 \ldots n]$

1. If $n = 1$, done
2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$
   and $A[\lceil n/2 \rceil + 1 \ldots n]$
3. ***"Merge"*** the $2$ sorted lists

- Schlüssel zum Erfolg: „Merge“ Prozedur
- Zusammenfügen zweier sortierter Reihen

# Zusammenfügen




1 2 7 9 ... 13 20

# Laufzeitanalyse

| | |
|---|---|
| $T(n)$ | **Merge−Sort** $A[1 \dots n]$ |
| $\Theta(1)$ | 1. If $n = 1$, done |
| $2T(n/2)$ | 2. Recursively sort $A[1 \dots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \dots n]$ |
| $\Theta(n)$ | 3. *"Merge"* the $2$ sorted lists |

Genauer wäre $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$
Macht asymptotisch keinen Unterschied

# Rekursionsgleichung

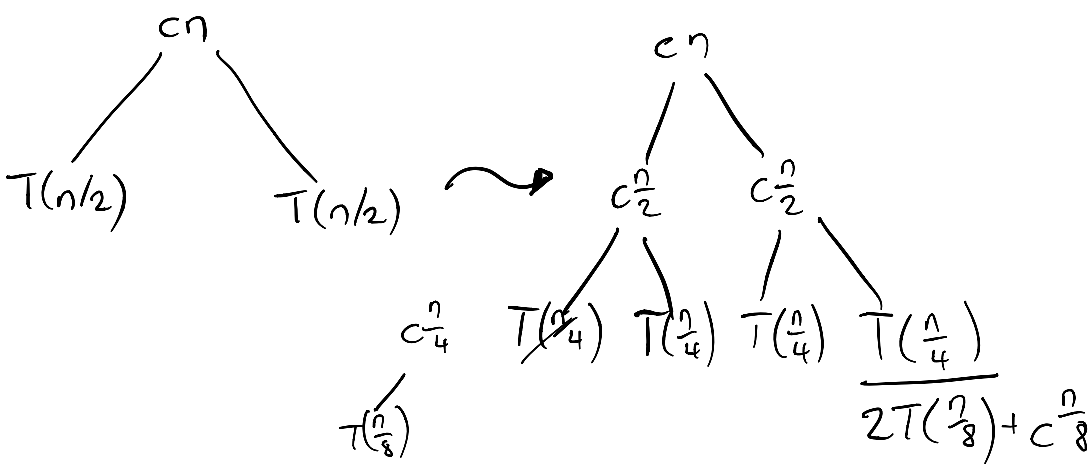
$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(n/2) + \Theta(n), & n > 1 \end{cases}$$

- Notation: Wir lassen den Fall $T(1) = \Theta(1)$ häufig weg
- Detaillierte Erklärung siehe Buch

# Rekursionsbaum

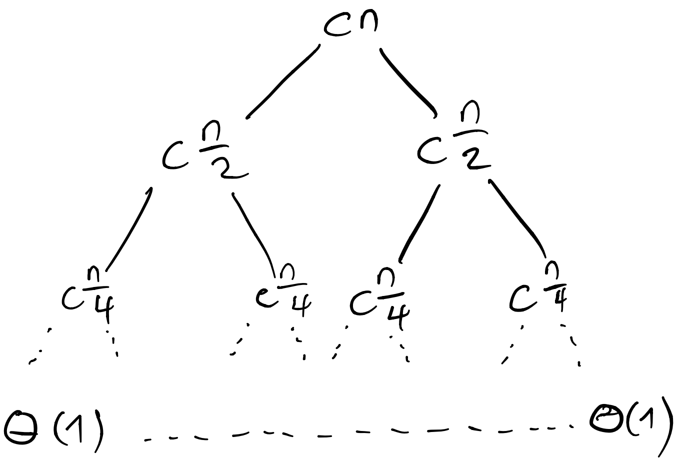
- Löse $T(n) = 2T(n/2) + cn$, $c > 0$ konstant

$$T(n) = 2\underbrace{T(n/2)}_{=2T(n/4)+cn} + cn$$



# Rekursionsbaum

- Löse $T(n) = 2T(n/2) + cn$, $c > 0$ konstant




| # Therme | Σ Therme | Tiefe Baum | Σ |
|---|---|---|---|
| 1 | $cn$ | 0 | $cn \cdot \log_2(n)$ |
| 2 | $cn$ | 1 | $+ \Theta(n)$ |
| 4 | $cn$ | 2 | $= \Theta(n \cdot \log n)$ |
| ⋮ | ⋮ | | |
| $n$ | $\Theta(n)$ | $\log_2(n)$ | |

# Vergleich

- $\Theta(n \lg n)$ wächst langsamer als $\Theta(n^2)$
- Merge sort ist asymptotisch schneller als Sortieren durch Einfügen im worst case
- In praktischen Implementationen ist merge sort schneller ab etwa $n > 30$
- Testen Sie selbst!

# Zusammenfassung

- Laufzeitanalyse
  - $T(n)$: Laufzeit abhängig von Eingabegrösse $n$
  - Worst-case, Average-case, (Best-case)
  - Asymptotisches Analyse: $T(n)$, $n \to \infty$
- Insertion Sort:
  - Idee: Einordnen von oben/unten her
  - Worst- und Average-case: $\Theta(n^2)$
- Merge Sort:
  - Divide-and-conquer Strategie
  - Laufzeitanalyse: $\Theta(n \log(n))$

# Nächste Vorlesung

- Mathematische Hilfsmittel
- Kapitel 3 und 4, ohne Abschnitte 4.2 und 4.6