Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2014 / 2015

Vorlesung 2, Donnerstag 30. Oktober 2014
(Laufzeitanalyse MinSort / HeapSort, Induktion)

Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

[basiert auf der Veranstaltung von Prof. Dr. Hannah Bast, Lehrstuhl für Algorithmen und Datenstrukturen, mit Folien aus „Informatik 2" von Prof. Dr. Matthias Teschner]

UNI FREIBURG

Überblick über die Vorlesung heute

UNI FREIBURG

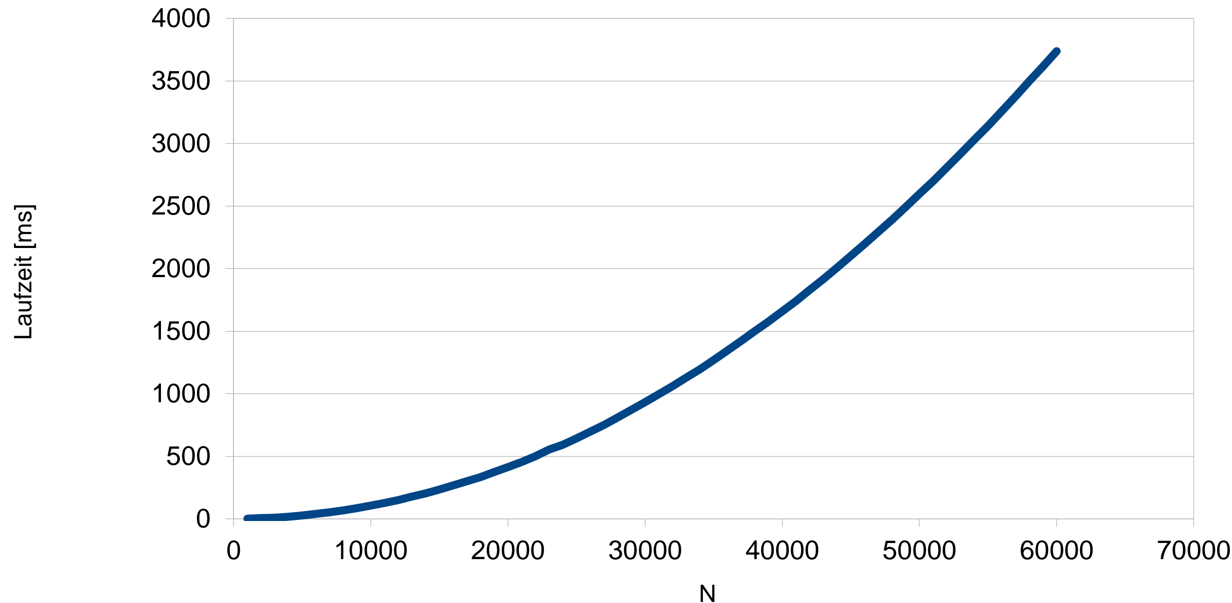
- Organisatorisches
  - Rückmeldungen zur Ü1 (HeapSort + Progammieren)
- Laufzeitanalyse
  - MinSort ... "quadratische" Laufzeit
  - HeapSort ... besser, aber trotzdem nicht ganz "linear"
  - Tiefe eines Baumes
  - Vollständige Induktion
  - Rechnen mit dem Logarithmus

Rückmeldungen

- Übung
  - einfacher Einstieg, keine wesentlichen Probleme, hat Spaß gemacht
  - Einrichtung Entwicklungsumgebung, etc. hat z.T. sehr viel Zeit gekostet (vor allem unter Windows)
  - Zeitbedarf sehr unterschiedlich (30 Minuten bis kompletter Tag), meistens im Rahmen der 4 vorgesehenen Stunden
  - Code „gegoogelt", „Wikipedia", „Musterlösung vom letzten Jahr"? **(nicht Sinn der Sache!)**
- Vorlesung
  - interessant, gut verständlich
  - etwas langsamer sprechen
  - Vorlesungs-PDF mit 1 Folie pro Blatt

MinSort — Laufzeit 1/7

UNI FREIBURG



- Wie lange läuft unser Programm?
  - In der letzten Vorlesung hatten wir dazu ein Schaubild
  - **Beobachtung:** es wird "unproportional" langsamer, je mehr Zahlen sortiert werden
  - Wie können wir präziser fassen, was da passiert?

MinSort — Laufzeit 2/7

UNI FREIBURG

- Wie analysieren wir die Laufzeit?
  - Idealerweise hätten wir gerne eine Formel, die uns für eine bestimme Eingabe sagt, wie lange das Programm dann läuft
  - **Problem:** Laufzeit hängt auch noch von vielen anderen Umständen ab, insbesondere
    - auf was für einem Rechner wir den Code ausführen
    - was sonst gerade noch auf dem Rechner läuft
    - welchen Compiler wir benutzt haben
    - und natürlich von der Mondphase
  - **Abstraktion 1:** Deshalb analysieren wir nicht die Laufzeit, sondern die Anzahl der (Grund-)Operationen

Grundoperationen

UNI FREIBURG

- Unvollständige Liste von Grundoperationen
  - Eine arithmetische Operation, z.B. a + b
  - Variablenzuweisung, z.B. x = y
  - Funktionsaufruf, z.B. Sorter.minSort(array)
    - das meint natürlich nur das Springen zu der Funktion
  - **Intuitiv:** eine Zeile Code
  - **Genauer wäre:** eine Zeile Maschinencode
  - **Noch genauer wäre:** ein Prozessorzyklus
  - Wir sehen später noch, dass es nicht so wichtig ist, wie genau wir die Grundoperationen definieren
    - Wichtig ist nur, dass die tatsächliche Laufzeit ungefähr **proportional** zur Anzahl Operationen ist

MinSort — Laufzeit 3/7

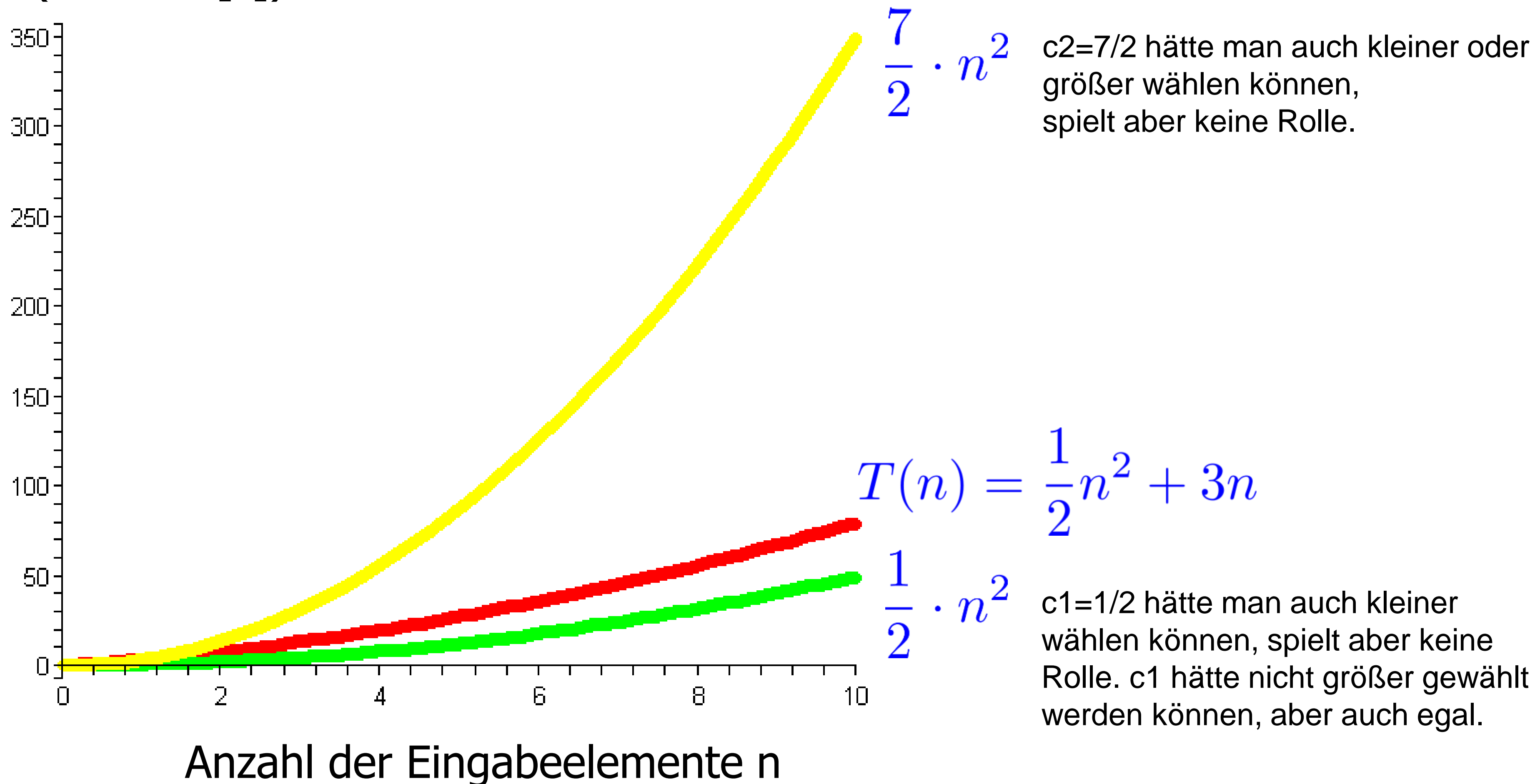
UNI FREIBURG

- Wieviele Operationen braucht MinSort?
  - **Abstraktion 2:** Wir zählen die Operationen nicht genau, sondern berechnen obere (und selten auch untere) Schranken

    Grund: das erleichtert die Sache und wir haben ja eh abstrahiert von exakter Laufzeit zu Anzahl Operationen
  - Sei n die Größe der Eingabe (= des Eingabearrays)
  - **Beobachtung:** Die Anzahl Operationen hängt nur von n ab, nicht davon, welche n Zahlen das sind ... das ist häufig so!
  - Sei $T(n)$ die Anzahl der Operationen bei Eingabegröße n
  - **Behauptung:** Es gilt $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$

    wobei C_1 und C_2 irgendwelche Konstanten sind
  - Das nennt man "quadratische Laufzeit" (wegen dem n^2)

Beispiel quadratische Laufzeit

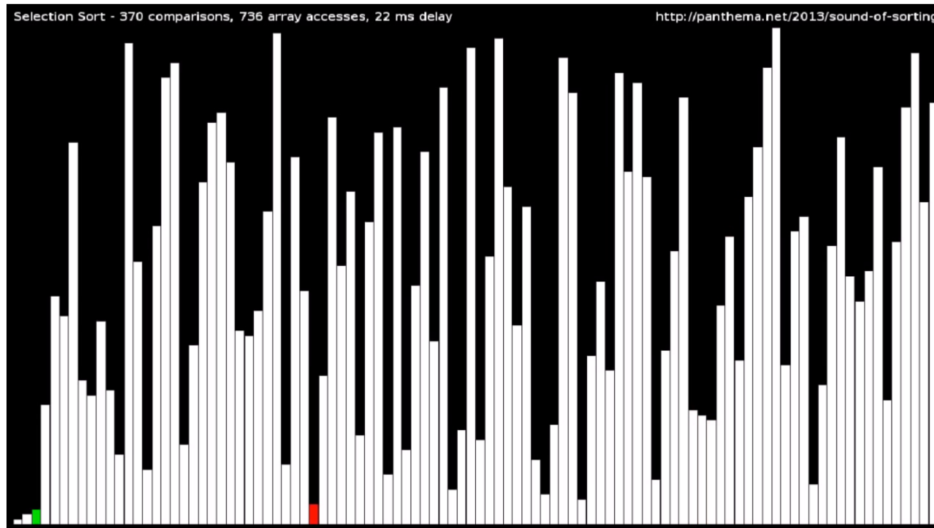


MinSort — Laufzeit 4/7

UNI FREIBURG

- Analyse der Anzahl der Operationen:

Sei n die Anzahl der zu sortierenden Elemente und T_i die Anzahl der Operationen in Runde i, dann ist



$$
\begin{aligned}
T_1 &\leq C'_2 \cdot n \\
T_2 &\leq C'_2 \cdot (n-1) \\
T_3 &\leq C'_2 \cdot (n-2) \\
T_4 &\leq C'_2 \cdot (n-3) \\
&\vdots \\
T_{n-1} &\leq C'_2 \cdot 2 \\
T_n &\leq C'_2 \cdot 1
\end{aligned}
$$

Beispiel: In Runde 4 müssen wir n-3 Elemente durchsuchen, um das Minimum zu finden

$$T(n) = T_1 + T_2 + \cdots + T_n \leq \boxed{\sum_{i=1}^{n} C'_2 \cdot i}$$

MinSort — Laufzeit 5/7

- Alternativ: Analyse des Codes:

```
public static void minSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
      int min = array[i];
      int minIndex = i;
      for (int j = i + 1; j < n; j++) {
        if (array[j] < min) {
          min = array[j];
          minIndex = j;
        }
      }
      int tmp = array[i];
      array[i] = array[minIndex];
      array[minIndex] = tmp;
    }
}
```

konst. Laufzeit

n-i-1 mal

n-1 mal

$$T(n) \leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C_2' = \sum_{i=0}^{n-2} (n-i-1) \cdot C_2' = \sum_{i=1}^{n-1} (n-i) \cdot C_2' \leq \sum_{i=1}^{n} C_2' \cdot i$$

MinSort — Laufzeit 6/7

UNI FREIBURG

- Beweis der Behauptung $T(n) \leq C_2 \cdot n^2$

$$
\begin{aligned}
T(n) &\leq \sum_{i=1}^{n} C_2' \cdot i \\
&= C_2' \cdot \sum_{i=1}^{n} i \\
&= C_2' \cdot \frac{n(n+1)}{2} \\
&\leq C_2' \cdot \frac{n(n+n)}{2} \\
&= C_2' \cdot \frac{2n^2}{2} \\
&= C_2' \cdot n^2
\end{aligned}
$$

Wie weiter vereinfachen?

Gaußsche Summenformel

$1 \leq n$

MinSort — Laufzeit 7/7

- Beweis der Behauptung $C_1 \cdot n^2 \leq T(n)$
  - Analog zur oberen Schranke können wir auch für die untere Schranke zeigen, dass es ein C_1' gibt, für das gilt:

$$T(n) \geq \sum_{i=1}^{n-1} (n-i) \cdot C_1' = \sum_{i=1}^{n-1} C_1' \cdot i$$

$$T(n) \geq C_1' \cdot \frac{(n-1) \cdot n}{2}$$

Wie kommen wir hier auf n^2 ?

$$n - 1 > \frac{n}{2} \text{ für } n > 2$$

$$> C_1' \cdot \frac{n \cdot n}{2 \cdot 2}$$

$$= \frac{C_1'}{4} \cdot n^2$$

Alles zusammen: $\frac{C_1'}{4} \cdot n^2 \leq T(n) \leq C_2' \cdot n^2$

Quadratische Laufzeit bewiesen: $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$

UNI FREIBURG

Quadratische Laufzeit in der Praxis

UNI FREIBURG

- Definition
  - Die Laufzeit T hängt von der Eingabegröße n ab
  - Es gibt Konstanten C_1 und C_2 mit $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$
- Betrachtungen dazu
  - Doppelt so große Eingabe → viermal so große Laufzeit

    $C_2 \cdot (2 \cdot n)^2 = C_2 \cdot 4 \cdot n^2$
  - Unabhängig von den Konstanten wird das schnell sehr teuer
    - $C = 1$ ns (1 einfache Anweisung ≈ 1 Nanosekunde)
    - $n = 10^6$ (1 Millionen Zahlen = 4 MB) [bei 4 Bytes/Zahl]
      - $C \cdot n^2 = 10^{-9} \cdot 10^{12} = 10^3$ s = 16.7 Minuten
    - $n = 10^9$ (1 Milliarde Zahlen = 4 GB)
      - $C \cdot n^2 = 10^{-9} \cdot 10^{18} = 10^9$ s = 31.7 Jahre

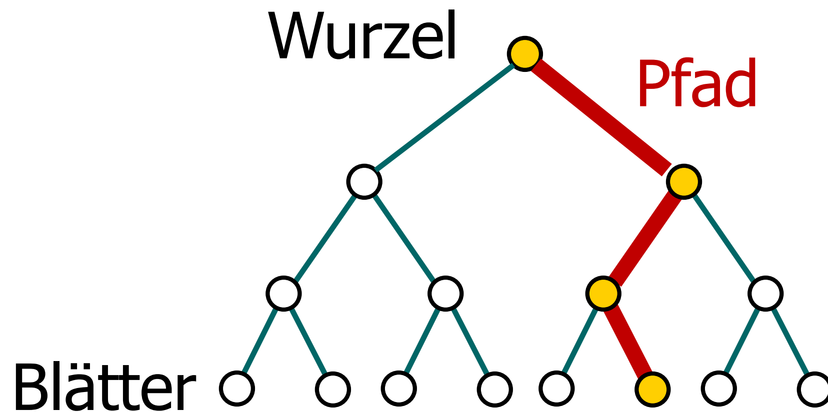
Quadr. Laufzeit = "große" Probleme unlösbar

HeapSort — Laufzeit 1/6

UNI FREIBURG

- Intuitiv
  - Zur Bestimmung des Minimums musste man bei MinSort in jeder Runde die ganzen übrigen Elemente durchgehen
  - Bei HeapSort ist es einfach immer die Wurzel vom Heap
  - Um den Heap zu reparieren, müssen wir aber nur einen Teil des Baumes durchgehen, nicht alle Element darin
- Formal
  - Sei $T(n)$ die Laufzeit von HeapSort für n Elemente
  - Auf den nächsten Folien zeigen wir $T(n) \leq C \cdot n \cdot \log_2 n$

HeapSort — Laufzeit 2/6



- Tiefe eines Baumes:
  - Die Tiefe d eines Baumes ist definiert als die maximale Anzahl Knoten auf einem Pfad Wurzel → Blatt
  - Ein vollst. binärer Baum der Tiefe d hat $n = 2^d - 1$ Knoten
  - Das wollen wir auf den nächsten Folien beweisen
  - Im Beispiel hier:

$$d = 4 \quad \Rightarrow \quad n = 2^4 - 1 = 15$$

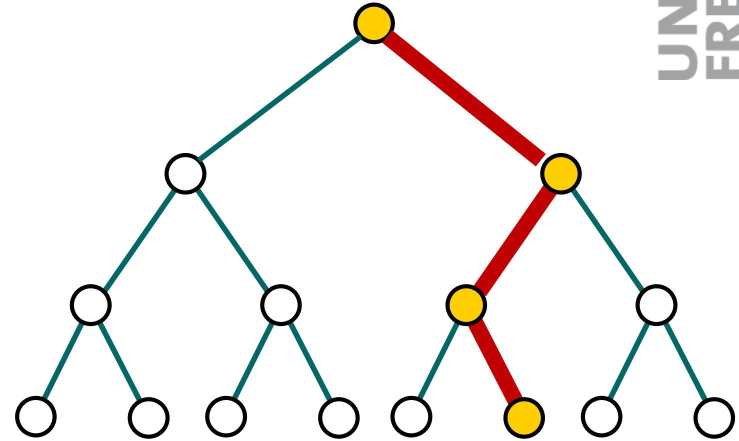
UNI FREIBURG

Einschub: Induktionsbeweis

- Prinzip
  - Man möchte beweisen, dass eine Aussage für alle natürlichen Zahlen gilt, also: $A(n)$ gilt für alle $n \in N$
  - Dann hat ein Induktionsbeweis zwei Schritte
  - Wir zeigen, dass $A(1)$ gilt (Induktionsanfang)
  - Wir nehmen an, dass $A(1), ..., A(n-1)$ gelten, für $n > 1$, und zeigen, dass dann auch $A(n)$ gilt (Induktionsschritt)
  - Wenn wir die beiden Sachen gezeigt haben, haben wir nach dem Prinzip der **vollständigen Induktion** gezeigt, dass $A(n)$ für alle natürlichen Zahlen n gilt

Beweis durch vollständige Induktion

Behauptung: Ein vollständiger Binärbaum der Tiefe d hat $n(d) = 2^d - 1$ Knoten

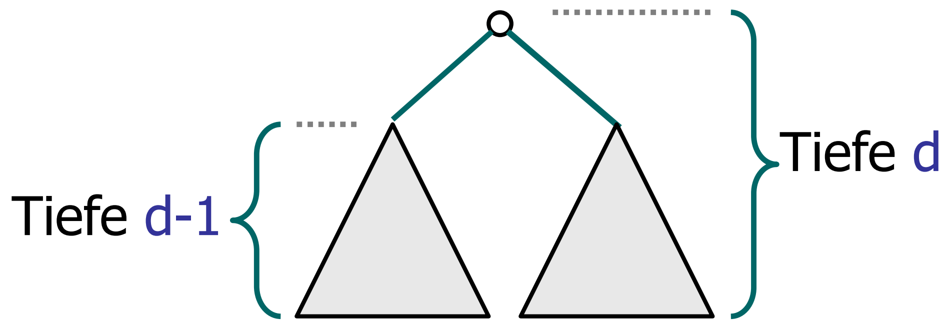


- **Induktionsanfang**: Formel gilt für d=1

$$n(1) = 2^1 - 1 = 1$$ Stimmt!

Beweis durch vollständige Induktion

- **Induktionsschritt**: Wenn die Formel für {1, ..., d-1} gilt, dann gilt sie auch für d



Ein vollst. Binärbaum der Tiefe d besteht aus 2 vollst. Binärbäumen der Tiefe d-1 und einem zusätzlichen Knoten

$$n(d) = 1 + 2 \cdot n(d-1)$$

Behauptung einsetzen

$$2^d - 1 = 1 + 2 \cdot (2^{d-1} - 1)$$

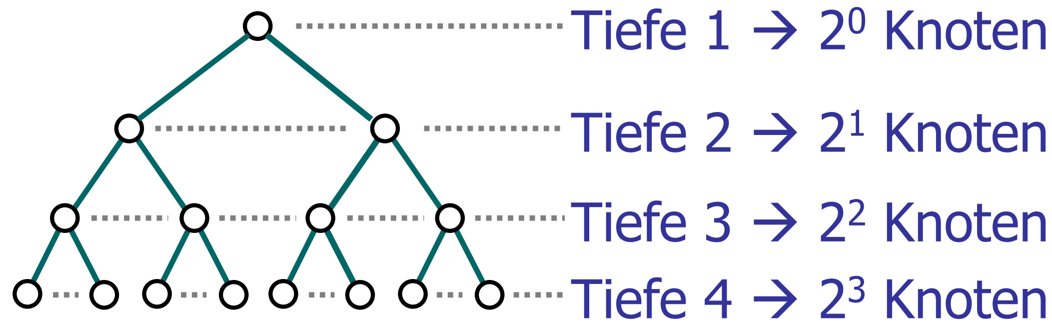
$$2^d - 1 = 1 + 2 \cdot 2^{d-1} - 2$$

$$2^d - 1 = 2^d - 1$$

Stimmt! Per vollständiger Induktion folgt, dass $n(d) = 2^d - 1$ für alle $d \in \mathbb{N}$ gilt.

UNI FREIBURG

HeapSort — Laufzeit 3/6



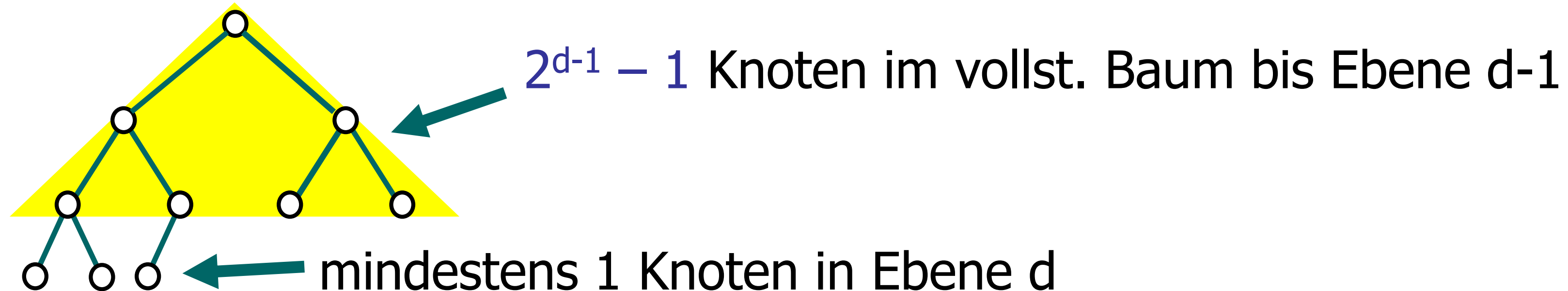
Allgemein: Tiefe d → 2^{d-1} Knoten

- Laufzeit von dem **heapify** am Anfang: Sei d die Tiefe des Heaps
  - in Tiefe d gibt es 2^{d-1} (oder weniger) Knoten aber keine Kosten
  - in Tiefe $d - 1$ gibt es 2^{d-2} Knoten. An jedem Knoten Kosten für Durchsickern auf einem Pfad der Länge 1, also Kosten $\leq C \leq C \cdot 2$
    (2 statt 1, um nachher eine Standardformel zu benutzen)
  - Tiefe $d - 1$; 2^{d-2} Knoten; Pfadlänge 1; Kosten pro Knoten $\leq C \cdot 2$
  - Tiefe $d - 2$; 2^{d-3} Knoten; Pfadlänge 2; Kosten pro Knoten $\leq C \cdot 3$
  - Tiefe $d - 3$; 2^{d-4} Knoten; Pfadlänge 3; Kosten pro Knoten $\leq C \cdot 4$
  - usw.
  - Gesamtkosten: $T(d) \leq \sum_{i=2}^{d} C \cdot i \cdot 2^{d-i}$ $\leq \sum_{i=1}^{d} C \cdot i \cdot 2^{d-1}$

HeapSort — Laufzeit 4/6

UNI FREIBURG

- Laufzeit von dem **heapify** am Anfang (Forts.)
  - Es gilt $\sum_{i=1}^{d} i \cdot 2^{d-i} \leq 2^{d+1}$ (siehe nächste Folie)
  - Also sind die Kosten für heapify: $T(d) \leq C \cdot 2^{d+1}$
  - Wir wollen die Kosten aber in Abhängigkeit von n
  - Ein Binärbaum der Tiefe d hat $n \geq 2^{d-1}$ Knoten ...warum?



  - (Gleichung mal 4) ➔ $4 \cdot n \geq 2^{d+1}$, also Kosten für heapify:

$$T(n) \leq C \cdot 4 \cdot n$$

Induktionsbeweis

UNI FREIBURG

Wir wollen beweisen: $$\sum_{i=1}^{d} i \cdot 2^{d-i} \leq 2^{d+1}$$

$$A(d) \leq B(d)$$

Die linke Seite nennen wir A(d), die rechte Seite B(d)

Induktionsanfang (d=1): $$A(1) \leq B(1)$$

$$\sum_{i=1}^{1} i \cdot 2^{1-i} \leq 2^{1+1}$$

$$2^0 \leq 2^2 \qquad \square$$

Induktionsbeweis $\sum_{i=1}^{d} i \cdot 2^{d-i} \leq 2^{d+1}$

UNI FREIBURG

Induktionsschritt: wir wollen zeigen: Wenn die Behauptung für {1,…,d} gilt, dann gilt sie auch für d+1

$$A(d) \leq B(d) \quad \Rightarrow \quad A(d+1) \leq B(d+1)$$

Rechte Formel ausschreiben und versuchen, A(d) und B(d) herauszuziehen:

$$\sum_{i=1}^{d+1} i \cdot 2^{d+1-i} \leq 2^{d+1+1}$$

$$\sum_{i=1}^{d+1} i \cdot 2 \cdot 2^{d-i} \leq 2 \cdot 2^{d+1}$$

$$2 \cdot \sum_{i=1}^{d+1} i \cdot 2^{d-i} \leq 2 \cdot B(d)$$

$$2 \cdot \sum_{i=1}^{d} i \cdot 2^{d-i} + 2 \cdot (d+1) \cdot 2^{d-(d+1)} \leq 2 \cdot B(d)$$

$$2 \cdot A(d) \boxed{+ d + 1} \leq 2 \cdot B(d)$$

Geht so nicht! Mist!

Die Behauptung stimmt aber trotzdem!

Induktionsbeweis $\sum_{i=1}^{d} i \cdot 2^{d-i} \leq 2^{d+1}$

UNI FREIBURG

- **Funktionierender Beweis**: Wir zeigen eine etwas stärkere Behauptung!

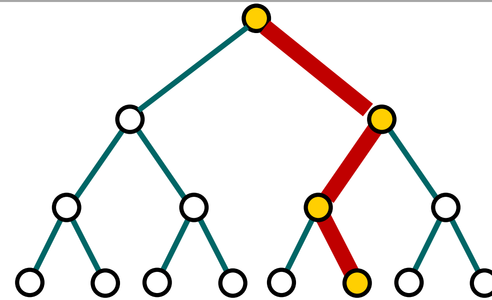
$$\sum_{i=1}^{d} i \cdot 2^{d-i} \leq 2^{d+1} - d - 2 \leq 2^{d+1}$$

Vorteil: dann hat man auch eine stärkere Induktionsvoraussetzung

→ Übungsaufgabe

HeapSort — Laufzeit 5/6

UNI FREIBURG



- Laufzeit für den Rest
  - n mal Maximum herausnehmen: jeweils konstante Kosten
  - n mal Heap reparieren („Durchsickern"): jeweils maximal d Schritte
  - Die Tiefe des Heaps am Anfang ist $d \leq 1 + \log_2 n$ ...warum?
  - → Aus $n \geq 2^{d-1}$ ➔ $\log_2 n \geq d - 1$ ➔ $d \leq 1 + \log_2 n$
  - Die Tiefe wird im Laufe der Zeit höchstens weniger
  - Also $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$ Operationen
  - Das Plus ist hässlich, also: $T(n) \leq C \cdot n \cdot 2 \cdot \log_2 n$ (gilt ab $n>2$)

HeapSort — Laufzeit 6/6

- Gesamtlaufzeit
  - Laufzeit heapify: $T(n) \leq 4 \cdot C \cdot n$
  - Laufzeit Rest: $T(n) \leq 2 \cdot C \cdot n \cdot \log_2 n$
  - Gesamtlaufzeit hätte wieder ein hässliches Plus, also erweitern wir die Schranke bei heapify noch etwas: $T(n) \leq 4 \cdot C \cdot n \leq 4 \cdot C \cdot n \cdot \log_2 n$
  - … Und erhalten insgesamt:
    $$T(n) \leq 6C \cdot n \cdot \log_2 n$$
  - Für die untere Schranke gilt das auch (Beweis heute nicht), also:
    $$C_1 \cdot n \cdot \log_2 n \leq T(n) \leq C_2 \cdot n \cdot \log_2 n$$
    für irgendwelche Konstanten C_1 und C_2 und $n \geq 2$

UNI FREIBURG

Einschub: Basis des Logarithmus

- Logarithmen zu verschiedenen Basem
  - Es gilt $\log_b n = \log_2 n / \log_2 b$
  - Das heißt, für zwei verschiedene Basen b und c

    unterscheiden sich $\log_b n$ und $\log_c n$ nur durch einen konstanten Faktor

    nämlich $\log_b c$ bzw. $\log_c b$
  - Zum Beispiel $\log_2 10 = 3.321928...$

Laufzeit proportional zu n · log n in der Praxis

UNI FREIBURG

- Schauen wir uns wieder Zahlenbeispiele an
  - Nehmen wir also an, es gibt Konstanten C_1 und C_2 mit

    $C_1 \cdot n \cdot \log_2 n \leq T(n) \leq C_2 \cdot n \cdot \log_2 n$ für $n \geq 2$
  - Dann dauert es bei doppelt so großer Eingabe nur geringfügig mehr als doppelt so lange
  - $C = 1$ ns (1 einfache Anweisung ≈ 1 Nanosekunde)
  - $n = 2^{20}$ (≈ 1 Millionen Zahlen = 4 MB) [bei 4 Bytes/Zahl]
    - $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{20} \cdot 20$ s = 21 Millisekunden
  - $n = 2^{30}$ (≈ 1 Milliarde Zahlen = 4 GB)
    - $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{30} \cdot 30$ s = 32 Sekunden

Laufzeit n · log n ist also fast so gut wie linear!

Literatur / Links

UNI FREIBURG

- Analyse von HeapSort

  In Mehlhorn/Sanders: 5. Sorting and Selection

  In Cormen/Leiserson/Rivest: II.7.1 HeapSort

  Wikipedia Artikel zu MinSort und HeapSort

- Vollständige Induktion

  http://de.wikipedia.org/wiki/Vollständige_Induktion

Neues Übungsblatt:

UNI
FREIBURG

- Diesmal 2 einfache Beweise und eine Programmieraufgabe