Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2012 / 2013

Vorlesung 2, Dienstag 30. Oktober 2012
(Laufzeitanalyse MinSort / HeapSort, Induktion)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü1 (HeapSort + Progammieren)
  - HowTo: Fragen auf dem Forum
- Laufzeitanalyse
  - MinSort ... "quadratische" Laufzeit
  - HeapSort ... besser, aber trotzdem nicht ganz "linear"
  - Tiefe eines Baumes
  - Vollständige Induktion
  - Rechnen mit dem Logarithmus

Erfahrungen mit dem Ü1 (HeapSort)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 30. Oktober 16:00
  - Viel Zeit für das Drumherum gebraucht (SVN, Gtest, ...)
  - Erst mal wieder ins Programmieren reinkommen
  - Manche sehr wenig oder sogar keine Programmierpraxis

    Sollten Sie ja eigentlich aus der Einf. in die Progr. haben !
  - Keine Codebeispiele für C++

    Siehe Vorlesung AlgDatEse WS1011 oder Progr. C++ SS2012

  - Wer will: noch eine Woche mehr, um am Ü1 zu arbeiten !

    Es gibt aber auch neue Aufgaben (Mathe), also bitte nicht erst nächsten Montag weitermachen ...

Fragen auf dem Forum

UNI FREIBURG

- Wie gesagt: nur keine Hemmungen
  - ABER fragen Sie richtig = **konkret** und **prägnant** !
  - Nicht sagen "Mein Code funktioniert nicht"
  - Sondern genaue Angabe von Fehlermeldung und den entsprechenden Zeilen im Code
  - Bei Fehlern in der Funktion: ein möglichst **minimales Programm** entwerfen bei dem der Fehler auftritt
    - dabei löst sich das Problem manchmal von selber
    - und wenn nicht, ist es für andere dann oft leicht
    - 100+ Zeilen will sich dagegen keiner angucken

MinSort — Laufzeit 1/4

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 2/4

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 3/4

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)

MinSort — Laufzeit 4/4

- Beweis der Behauptung $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$

Sei T_i die Anzahl der Operationen in Runde i

Dann ist $T(n) = T_1 + T_2 + \cdots + T_n = \sum_{i=1}^{n} T_i$

$T_1 \leq C_2' \cdot n$ für irgendeine Konstante C_2'

$T_2 \leq C_2' \cdot (n-1)$

$T_3 \leq C_2' \cdot (n-2)$

$T_4 \leq C_2' \cdot (n-3)$

$\vdots$

$$\sum_{i=1}^{n} T_i \leq C_2' \cdot \underbrace{\sum_{i=1}^{n} i}_{= \frac{n(n+1)}{2} \leq 2n^2} \leq n^2$$

$$\boxed{T(n) \leq C_2' \cdot n^2}$$

$T_1 \geq C_1' \cdot n$ für irgend ein C_1'

$T_2 \geq C_1' \cdot (n-1)$

$T_3 \geq C_1' \cdot (n-2$

$\vdots$

$$\sum_{i=1}^{n} T_i \geq C_1' \cdot \underbrace{\sum_{i=1}^{n} i}_{= \frac{n(n+1)}{2} \geq n^2/2}$$

$$\boxed{T(n) \geq \frac{C_1'}{2} \cdot n^2}$$

Quadratische Laufzeit

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 $(2n)^2 = 4 \cdot n^2$
  - Doppelt so große Eingabe → viermal so große Laufzeit
  - 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

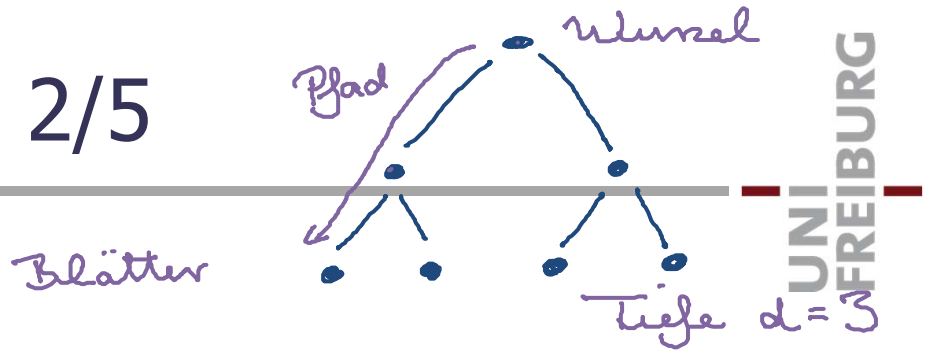
Quadr. Laufzeit = "große" Probleme unlösbar

HeapSort — Laufzeit 1/5

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 2/5



- Tiefe eines Baumes
  - Die Tiefe eines Baumes ist definiert als die maximale Anzahl Knoten auf einem Pfad Wurzel → Blatt
  - Ein vollst. binärer Baum der Tiefe d hat $2^d - 1$ Knoten
  - Beweis über vollständige Induktion:

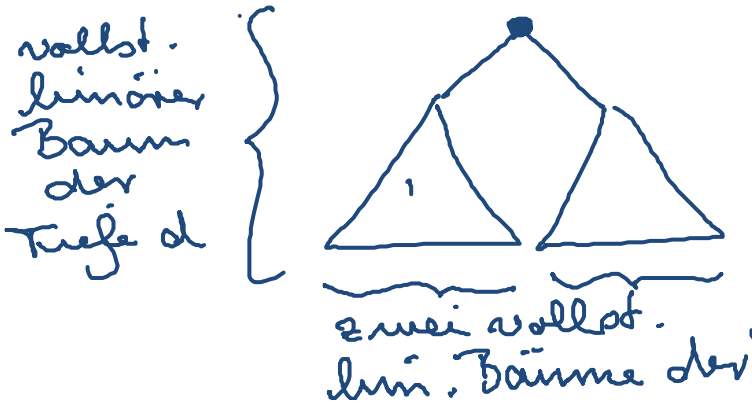
#Knoten = n
$= 2^3 - 1 = 7$

Induktionsanfang: $d = 1$ • $n = 1 = 2^1 - 1$ ✓

Induktionsschritt:

Annahme: Formel gilt für $1, \dots, d-1$

Dann zeigen wir jetzt, dass sie auch für d gilt.



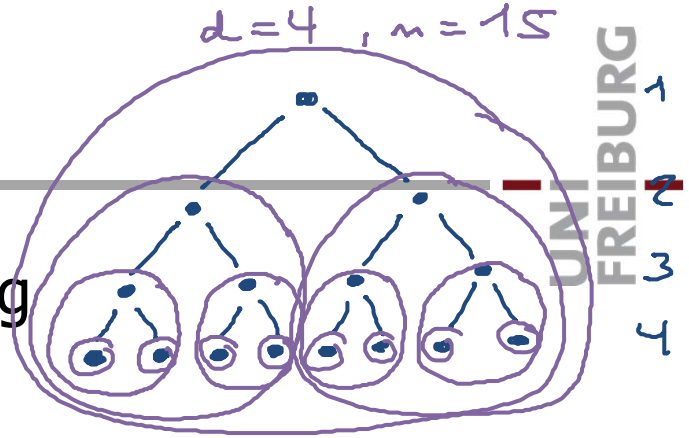
n = #Knoten
$= 1 + 2 \cdot$ #Knoten in Baum der Tiefe $d-1$
$2^{d-1} - 1$ nach Induktionsannahme
$= 1 + 2(2^{d-1} - 1)$
$= 1 + 2^d - 2 = 2^d - 1$ ∎

Einschub: Induktionsbeweis

UNI FREIBURG

- 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 $A(1), \ldots, A(n-1)$ 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

HeapSort — Laufzeit 3/5



- Laufzeit von dem heapify am Anfang
  - Sei d die Tiefe des Heaps
  - Keine Kosten für die $\leq 2^{d-1}$ Knoten in Tiefe d
  - Kosten $\leq C \cdot 2$ für jede der 2^{d-2} Knoten in Tiefe $d-1$
  - Kosten $\leq C \cdot 3$ für jede der 2^{d-3} Knoten in Tiefe $d-2$
  - Kosten $\leq C \cdot 4$ für jede der 2^{d-4} Knoten in Tiefe $d-3$
  - Und so weiter ...
  - Es gilt $\sum_{i=1,\ldots,d} 2^{d-i} \cdot i \leq 2^{d+1}$ ... Beweis nächste Folie
  - Ein Baum der Tiefe d hat $n \geq 2^{d-1}$ Knoten
  - Also $2^{d+1} \leq 4 \cdot n$, also Kosten für heapify maximal $4C \cdot n$

HeapSort — Laufzeit 4/5

- Beweis von $\sum_{i=1,\ldots,d} 2^{d-i} \cdot i \leq 2^{d+1}$

Induktion über d

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

Induktionsschritt:

Nehmen wir an, die Formel gilt für $1, \ldots, d-1$.

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

$$= 2 \cdot \underbrace{\sum_{i=1}^{d-1} 2^{d-1-i} \cdot i}_{\leq 2^{d-1+1} = 2^d \text{ nach Induktionsvoraussetzung}} + d$$

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

wir hätte gerne $\leq 2^{d+1}$

geht aber nicht $\Rightarrow$ DOOF.

die Aussage aber stimmt aber

LÖSUNG:

man zeigt eine etwas stärkere Behauptung!

(Vorteil: dann hat man auch eine stärkere Induktionsvoraussetzung)

HeapSort — Laufzeit 5/5

wir hatten schon gesehen, dass $n > 2^{d-1}$

$d-1 < \log_2 n$

$d < 1 + \log_2 n$

stimmt exakt nur falls n eine Zweierpotenz ist

- Laufzeit für den Rest
  - Die Tiefe des Heaps am Anfang ist $d \le \log_2 n$
  - Die Tiefe wird im Laufe der Zeit höchstens weniger
  - Also $\le C \cdot n \cdot \log_2 n$ Operationen

    Übungsaufgabe: auch $\ge C_1 \cdot n \cdot \log_2 n$ Operationen
  - Mit den $\le 4C \cdot n$ von heapify sind das $\le 5C \cdot n \cdot \log_2 n$
  - Sei $T(n)$ die Gesamtlaufzeit von HeapSort, dann gilt

    $C_1 \cdot n \cdot \log_2 n \le T(n) \le C_2 \cdot n \cdot \log_2 n$

    für irgendwelche Konstanten C_1 und C_2 und $n \ge 2$
  - Anmerkung: das $n \ge 2$ brauchen wir, weil $\log_2 1 = 0$

Einschub: Basis des Logarithmus

UNI FREIBURG

- 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

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