Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 2a, Dienstag, 2. Mai 2017
(Laufzeitanalyse MinSort und MergeSort)

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

UNI
FREIBURG

Blick über die Vorlesung heute

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü1 (Drumherum + Sortieren)
- Laufzeitanalyse
  - Allgemein — wie fasst man Laufzeit mathematisch?
  - MinSort — "quadratische" Laufzeit
  - MergeSort — besser, aber auch nicht ganz "linear"
  - Auffrischung — vollständige Induktion, Logarithmus
  - ÜB2: vier wunderschöne Theorieaufgaben zum Einüben dieser Grundtechniken und Konzepte

Erfahrungen mit dem ÜB1 1/2

- Zusammenfassung / Auszüge
  - Wie üblich beim ÜB1 sehr große Unterschiede in den Bearbeitungszeiten: bei manchen war es schnell gemacht, andere haben ewig daran gesessen
  - Einige fanden das iterative MergeSort kompliziert
  - Einige kamen mit rekursiv vs. iterativ durcheinander

    Das Bild von Vorlesung 1b, Folie 6 ist aber sehr klar
  - Es konnte viel Code aus der VL übernommen werden
  - Viele Fragen auf dem Forum zum "Drumherum"

    Das ist auch normal für das erste Übungsblatt

Erfahrungen mit dem ÜB1 2/2

- Ergebnisse
  - MergeSort ist **viel** schneller als MinSort
  - Die Laufzeit auf dem Schaubild sieht "linear" aus

Laufzeitanalyse allgemein 1/5

- Wie lange laufen unsere bisherigen Programme?
  - Für MinSort und MergeSort hatten wir dazu bisher zwei Schaubilder und Folgendes beobachtet

    MinSort: Laufzeit wird "unproportional" langsamer, und damit schon bei mittleren Eingabegrößen sehr langsam

    Doppelt so große Eingabe → mehr als doppelt so langsam

    MergeSort: Laufzeit wird "proportional" langsamer und bei mittleren Eingabegrößen viel schneller als MinSort

    Doppelt so große Eingabe → ca. doppelt so langsam

Laufzeitanalyse allgemein 2/5

- Wie können wir das präziser fassen
  - Idealerweise: eine Formel, die uns für eine bestimmte Eingabe sagt, wie lange das Programm darauf läuft
  - Problem: Laufzeit hängt auch noch von vielen anderen Umständen ab, insbesondere
    - auf was für einem Rechner wird den Code ausführen
    - was sonst gerade noch auf dem Rechner läuft
    - was für eine Programmiersprache benutzt wurde
    - welchen Compiler wir benutzt haben
    - Jahreszeit, Mondphase, Raum-Zeit-Verkrümmung, …

Laufzeitanalyse allgemein 3/5

- Abstraktion 1: Anzahl Operationen statt Laufzeit
  - Intuitiv: eine Operation = eine Zeile Code, zum Beispiel:

Arithmetische Operationen	a + b
Variablenzuweisungen	x = y
Verzweigungen	if … else …
Sprung zu einer Funktion	min_sort(…)

  - Genauer wäre: eine Zeile Maschinencode … oder noch genauer: ein Prozessorzyklus

    Wir sehen später noch, dass es nicht so wichtig ist, wie genau wir diese Operationen definieren

    Wichtig ist, dass die Anzahl Operationen ungefähr **proportional** zur tatsächlichen Laufzeit ist

Laufzeitanalyse allgemein 4/5

- Abstraktion 2: Abschätzung statt genau zählen
  - Meistens Abschätzung nach oben ("obere Schranke")

    Dann wissen wir, wie lange ein Programm höchstens braucht
  - Seltener Abschätzung nach unten ("untere Schranke")

    Dann wissen wir, wie lange ein Programm mindestens braucht

    Schätzen satt genau zählen erleichtert die Aufgabe

    Und wir haben ja eh schon abstrahiert von der exakten Laufzeit zu der Anzahl Operationen

Laufzeitanalyse allgemein 5/5

- Abstraktion 3: Schranken pro Eingabegröße n
  - Oft hängt die Laufzeit vor allem von der Größe der Eingabe ab, und nur wenig davon, wie die Eingabe genau aussieht

    Zum Beispiel hängt die Laufzeit von MinSort für eine Eingabegröße n nur minimal von der genauen Eingabe ab
  - Wir schreiben deswegen für die Laufzeit oft **T(n)**

    Wenn wir obere Schranken berechnen wollen, ist damit die **maximale** Laufzeit für eine Eingabe der Größe n gemeint

    Wenn wir untere Schranken berechnen wollen, ist damit die **minimale** Laufzeit für eine Eingabe der Größe n gemeint

    Diese Notation ist mathematisch etwas unpräzise, aber es ist in aller Regel klar, was gemeint ist

Laufzeitanalyse MinSort 1/4

für irgendwelche Konstanten A und B

- Es gilt: $T(n) \le C_1 \cdot n^2$ … für irgendeine Konstante C_1
  - MinSort hat eine äußere und eine innere Schleife
  - Für jede Iteration der äußeren Schleife, schätzen wir die Anzahl Operationen der inneren Schleife ab

Iteration 1: $\le A \cdot (n-1) + B$

Iteration 2: $\le A \cdot (n-2) + B$

...

Iteration n−1: $\le A + B$

Insgesamt: $T(n) \le A \cdot (\underbrace{n-1+n-2+\ldots+1}_{\sum_{i=1}^{n-1} i = \frac{1}{2} n(n-1)}) + B \cdot (n-1)$

$$\le \frac{1}{2} A \cdot n \underbrace{(n-1)}_{\le n} + B \cdot \underbrace{(n-1)}_{\le n^2}$$

$$\le \frac{1}{2} A \cdot n^2 + B \cdot n^2$$

$$\le \underbrace{(A/2 + B)}_{=: C_1} \cdot n^2$$

Laufzeitanalyse MinSort 2/4

für irgendwelche Konstanten A' und B'

- Es gilt auch: $T(n) \geq C_2 \cdot n^2$ ... für eine Konst. $C_2 < C_1$

für $n \geq 2$

Iteration 1 : $\geq A' \cdot (n-1) + B'$

Iteration 2 : $\geq A' \cdot (n-2) + B'$

...

Iteration $n-1$: $\geq A' + B'$

Insgesamt : $T(n) \geq A' \cdot \underbrace{(n-1 + n-2 + \dots + 1)}_{= \frac{1}{2} n (n-1)} + B' \cdot (n-1)$

$$\geq A'/2 \cdot n \cdot \underbrace{(n-1)}_{\geq \frac{n}{2}} + \underbrace{B' \cdot (n-1)}_{\geq 0}$$

$$\geq \underbrace{A'/4}_{=: C_2} \cdot n^2$$

Laufzeitanalyse MinSort 3/4

- Quadratische Laufzeit
  - Es gibt Konstanten C_1 und C_2 , so dass gilt

    $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$

  - Dann gilt insbesondere

    Doppelt so große Eingabe → viermal so große Laufzeit

    $T(2n) = C \cdot (2n)^2 = 4 \cdot C \cdot n^2 = 4 \cdot T(n)$

Laufzeitanalyse MinSort 4/4

- Quadratische Laufzeit, Rechenbeispiel
  - Unabhängig von den Konstanten wird das schnell sehr teuer
  - Annahme: $C = 1$ ns (10^{-9}) (1 einfache Anweisung ≈ 1 Nanosekunde)
  - Beispiel 1: $n = 10^6$ (1 Millionen Zahlen = 4 MB, 4 Bytes/Zahl)

    … dann $C \cdot n^2 = \underline{10^{-9}} \cdot 10^{12}$ s $= 10^3$ s $= 16.7$ Minuten
  - Beispiel 2: $n = 10^9$ (1 Milliarde Zahlen = 4 GB) × 1 Millionen

    … dann $C \cdot n^2 = 10^{-9} \cdot 10^{18}$ s $= 10^9$ s $= 31.7$ Jahre

Quadratische Laufzeit = "große" Probleme unlösbar

Laufzeitanalyse MergeSort 1/6

für irgendeine Konstante A

- Analyse iterativer MergeSort
  - Annahme: n ist eine Zweierpotenz und Mischen von zwei Feldern der Größe m geht in höchstens $A \cdot m$ Zeit

Iteration 1: n Teilfelder der Größe jeweils 1

$n / 2$ mal Mischen à $A \cdot \underbrace{1}_{2^0}$ → Zeit $n / 2 \cdot A \cdot 1 = \mathbf{A / 2 \cdot n}$

Iteration 2: $n/2$ Teilfelder der Größe jeweils 2

$n / 4$ mal Mischen à $A \cdot \underbrace{2}_{2^1}$ → Zeit $n / 4 \cdot A \cdot 2 = \mathbf{A / 2 \cdot n}$

…

Iteration k: 2 Teilfelder der Größe jeweils $n / 2$

1 mal Mischen à $A \cdot \underbrace{n / 2}_{2^{k-1}}$ → Zeit $1 \cdot A \cdot n / 2 = \mathbf{A / 2 \cdot n}$

Laufzeitanalyse MergeSort 2/6

- Analyse iterativer MergeSort ... Fortsetzung
  - Sei k die Anzahl der Iterationen
  - Dann ist die Laufzeit insgesamt $T(n) \leq \mathbf{A/2 \cdot n \cdot k}$
  - Wie groß ist k ?
  - Die Teilfelder in Iteration k sind $2^{k-1} = n / 2$ groß
  - Also $k - 1 = \log_2 (n/2) = (\log_2 n) - 1 \leq \log_2 n$ $\Rightarrow k \leq \log_2 n + 1$
  - Es ist also $T(n) \leq A/2 \cdot n \cdot (1 + \log_2 n)$ ÜBRIGENS : $\log_2 1 = 0$
  - Frage: kommt bei der rekursiven Implementierung auch etwas mit $n \cdot \log_2 n$ heraus?

Laufzeitanalyse MergeSort   3/6

- Analyse rekursiver MergeSort
  - Annahme wieder: n ist eine Zweierpotenz und Mischen von zwei Feldern der Größe m geht in ≤ A · m Zeit
  - Nach dem Bild von Vorlesung 1b, Folie 8 :

    rek. Aufruf linke Hälfte   rek. Aufruf rechte Hälfte   Mischen

    $T(n) \leq T(n/2) + T(n/2) + A \cdot n / 2$

    Das gilt aber nur, wenn wir die Rekursion tatsächlich ausführen, also für n ≥ 2; für n = 1 haben wir einfach:

    $T(1) \leq A$

  - Solch eine rekursive Gleichung ist typisch bei der Analyse der Laufzeit von einem rekursiven Algorithmus

    ohne T auf der rechten Seite

    Wie kommen wir damit auf eine obere Schranke für T(n) ?

Laufzeitanalyse MergeSort 4/6

Annahme: $n =$ Zweierpotenz $= 2^k$

- Auflösung der rekursiven Gleichung

$$\underline{T(n) \le 2 \cdot T(n/2) + A \cdot n/2} \qquad (*)\ \forall n$$

$$\underset{\text{für } n/2}{\overset{(*)}{\le}} 2 \cdot [2 \cdot T(n/4) + A \cdot n/4] + A \cdot n/2$$

$$= 4 \cdot T(n/4) + 2 \cdot A \cdot n/2$$

$$\underset{\text{für } n/4}{\overset{(*)}{\le}} 4 \cdot [2 \cdot T(n/8) + A \cdot n/8] + 2 \cdot A \cdot n/2$$

$$= 8 \cdot T(n/8) + 3 \cdot A \cdot n/2$$

$$\vdots$$

$$\le 2^k \cdot T(n/2^k) + k \cdot A \cdot n/2$$

$2^k = n \iff k = \log_2 n$

$$\le n \cdot \underbrace{T(1)}_{\le A} + \log_2 n \cdot A \cdot \underbrace{n/2}_{\le n}$$

$$\le A \cdot n + A \cdot n \cdot \log_2 n$$

$$\le A \cdot n \cdot (1 + \log_2 n)$$

Laufzeitanalyse MergeSort 5/6

- Laufzeit $n \cdot \log n$
  - Es gibt Konstanten C_1 und C_2, so dass gilt

    $C_1 \cdot n \cdot \log_2 n \leq T(n) \leq C_2 \cdot n \cdot \log_2 n$ für $n \geq 2$
  - Dann gilt insbesondere

    Doppelt so große Eingabe → etwas mehr als doppelt so lange

$$T(2n) = C \cdot 2 \cdot n \cdot \underbrace{\log_2(2n)}_{\log_2 2 + \log_2 n} = 2 \cdot C \cdot n \cdot \underbrace{(1 + \log_2 n)}_{\approx \log_2 n} \approx 2 \cdot T(n)$$

$1 + \log_2 n$

Laufzeitanalyse MergeSort 6/6

- Laufzeit $n \cdot \log n$, Rechenbeispiel

  10^{-9}

  - Annahme: $C = 1$ ns (1 einfache Anweisung ≈ 1 Nanosekunde)

  $2^{10} = 1024 \approx 10^3$

  - Beispiel 1: $n = 2^{20}$ (≈ 1 Millionen Zahlen = 4 MB)

    $\approx 10^6$

    … dann $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{20} \cdot 20$ s = 21 Millisekunden

  $\approx 10^9$

  - Beispiel 2: $n = 2^{30}$ (≈ 1 Milliarde Zahlen = 4 GB) $\times 1000$ $\times 1.5$

    $\approx 10^9$

    … dann $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{30} \cdot 30$ s = 32 Sekunden

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

Auffrischung 1/4

- Vollständige Induktion, Prinzip
  - Man möchte beweisen, dass eine Aussage für alle natürlichen Zahlen gilt, also: $A(n)$ gilt für alle $n \in \mathbf{N}$
  - **Induktionsanfang**: Wir zeigen, dass $A(1), \ldots, A(k)$ gelten

    Meistens reicht $k = 1$, aber manchmal braucht man mehr
  - **Induktionsschritt:** Wir nehmen für ein beliebiges $n > k$ an, dass $A(1), \ldots, A(n-1)$ gelten, und zeigen: dann gilt auch $A(n)$

    Meistens reicht $A(n-1)$, aber manchmal auch $A(n-2), \ldots$
  - 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

Auffrischung 2/4

- Vollständige Induktion, Beispiel
  - Wir haben vorhin benutzt: $\Sigma_{i=1..n}\, i = ½ \cdot n \cdot (n + 1)$

Induktionsanfang $n = 1$:

$$\sum_{i=1}^{1} i = 1 = 1 = \frac{1}{2} \cdot 1 \cdot \underbrace{(1+1)}_{=2}$$

Induktionsschritt $1, \ldots, n \to n+1$:

für bel. $n \geq 1$

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + n + 1$$

$$= \frac{1}{2} n (n+1) \text{ nach Induktionsvoraussetzung}$$

$$= \frac{1}{2} n (n+1) + n + 1$$

$$= \Big(\frac{1}{2} n + \underbrace{1}_{\frac{1}{2} \cdot 2}\Big)(n+1)$$

$$= \frac{1}{2}(n+2)(n+1)$$

Auffrischung 3/4

- Der Logarithmus (≠ Algorithmus)
  - Der "Logarithmus zur Basis b" ist gerade die inverse Funktion zu "b hoch"

    Formal: $\log_b n = x \Leftrightarrow b^x = n$

    Beispiel: $\log_2 1024 = 10 \Leftrightarrow 2^{10} = 1024$
  - Die Rechenregeln ergeben sich dann einfach aus den Rechenregeln für das Potenzieren
  - Zum Beispiel: $\underbrace{\log_b (x \cdot y)}_{=: z} = \underbrace{(\log_b x)}_{=: z_1} + \underbrace{(\log_b y)}_{=: z_2}$

$z = \log_b(x \cdot y) \Rightarrow b^z = x \cdot y$

$z_1 = \log_b x \Rightarrow b^{z_1} = x$

$z_2 = \log_b y \Rightarrow b^{z_2} = y$

$b^z = x \cdot y = b^{z_1} \cdot b^{z_2} = b^{z_1 + z_2}$

$\Rightarrow z = z_1 + z_2$

$\Leftrightarrow \log_b(x \cdot y) = \log_b x + \log_b y$

Auffrischung 4/4

- Der Logarithmus, Fortsetzung
  - Der Logarithmus kommt in der Informatik **sehr häufig** vor, insbesondere bei der Analyse von Laufzeiten

    $\log_2 n$ ist gerade: wie oft man n halbieren muss, bis man bei 1 ankommt ... oder umgekehrt: wie oft man 1 verdoppeln muss, bis man bei n ankommt

  - Es kommt auch öfter mal vor, dass der Logarithmus in einer Potenz auftaucht, aber mit einer anderen Basis

    Zum Beispiel: $3^{\log_2 n}$

    $(b^x)^y = b^{x \cdot y} = (b^y)^x$

    In welcher Größenordnung liegt das?

    $3 = 2^{\log_2 3} \Rightarrow 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n}$

    $= 2^{\log_2 3 \cdot \log_2 n}$

    $= 2^{\log_2 n \cdot \log_2 3} = (\underbrace{2^{\log_2 n}}_{= n})^{\log_2 3}$

    $= n^{\log_2 3} \approx n^{1.58496\ldots}$

Literatur / Links

- Laufzeitanalyse
  - Mehlhorn/Sanders: 2.6 Basic Program Analysis
  - Wikipedia: Vollständige Induktion
  - Stupidedia: Unvollständige Induktion