Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 2b, Mittwoch, 3. Mai 2017
(Andere Sortierverfahren, Sortieren von Objekten, Sortieren in Linearzeit, Untere Schranke $n \cdot \log n$)

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

Blick über die Vorlesung heute

- Inhalt
  - Andere Sortierverfahren — QuickSort, HeapSort, …
  - Sortieren von Objekten — Und nicht nur Zahlen
  - Sortieren in Linearzeit — 0-1-Sort und CountingSort
  - Untere Schranke — vergleichsbasiert geht es nicht besser als $n \cdot \log n$

Andere Sortierverfahren 1/8

- QuickSort, Grundprinzip
  - Ähnlich wie dem rekursiven MergeSort wird das Feld in zwei Teile aufgeteilt, die dann rekursiv sortiert werden
  - **Unterschied 1:** die Aufteilung ist so, dass alle Elemente im linken Teil ≤ alle Elemente im rechten Teil sind

    Teilergebnisse müssen dann nicht mehr gemischt werden
  - **Unterschied 2:** die Teile können sehr unterschiedlich groß sein, im schlechtesten Fall hat ein Teil nur Größe 1

    Die Rekursionstiefe kann so viel größer als $\log_2 n$ werden

Andere Sortierverfahren 2/8

- QuickSort, Aufteilen + Beispiel
  - Zum Aufteilen wird ein Element P des Feldes gewählt (z.B. das erste Element oder ein zufälliges Element)
  - Das Feld wird dann so aufgeteilt, dass links alle Elemente ≤ P stehen und rechts alle Elemente ≥ P

der sogenannte PIVOT

Das geht für ein Feld der Größe n in Zeit ≤ A · n

PIVOT

2 7 1 9 5 3 4 6

2 4 1 3 5 9 7 6

≤ 5   > 5

REK   REK

1 2 3 4 5 6 7 9

schon fertig, Mischen nicht mehr nötig

Andere Sortierverfahren 3/8

- QuickSort, Laufzeit
  - Im besten Fall werden die Felder immer in zwei (fast) gleich große Hälften aufgeteilt, wie bei MergeSort

    Die Laufzeit ist dann $\leq C_1 \cdot n \cdot \log_2 n$ wie bei MergeSort

    Aber in der Praxis schneller als MergeSort, weil das Mischen wegfällt und keine Felder kopiert werden müssen
  - Im schlechtesten Fall wird das Feld pro Rekursionsstufe immer nur eins kleiner und die Laufzeit ist $\geq C_2 \cdot n^2$
  - Wenn das Pivot-Element immer zufällig gewählt wird, ist der Erwartungswert der Laufzeit auch $\leq C_3 \cdot n \cdot \log_2 n$

Andere Sortierverfahren 4/8

- HeapSort
  - HeapSort benutzt einen **binären Heap** zum Sortieren

    Das ist eine Datenstruktur, die aus einer Menge von n Elementen mit $\leq C \cdot \log n$ Operationen das kleinste Element extrahieren und entfernen kann

    Dazu in einer späteren Vorlesung mehr !
  - HeapSort schafft damit auch **in jedem Fall**

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$ für eine Konstante $C > 0$
  - In der Praxis:

    HeapSort etwas besser als MergeSort … kleineres C
    HeapSort etwas schlechter als QuickSort … größeres C

Andere Sortierverfahren 5/8

- Intelligent Design Sort
  - Annahme: die Eingabezahlen sind alle verschieden
  - Dann gibt es n! mögliche Permutationen dieser Zahlen
  - Mit Wahrscheinlichkeit 1/n! ist die Eingabe also sortiert
  - Weil diese Wahrscheinlichkeit so klein ist, ist es absurd zu denken, dass dies zufällig passiert ist
  - Es muss durch einen Intelligenten Sortierer erfolgt sein
  - Man kann deswegen annehmen, dass die Eingabe schon optimal sortiert war … in einer Reihenfolge, die unser weltliches Vertständnis von "sortiert" transzendiert

Andere Sortierverfahren 6/8

- BogoSort
  - **Schritt 1:** Prüfe, ob die Eingabe bereits sortiert ist

    Das geht in Zeit $\leq C_1 \cdot n$, für eine Konstante C_1
  - **Schritt 2:** Falls nicht, permutiere die Zahlen zufällig und gehe zu Schritt 1

    Das geht ebenfalls in Zeit $\leq C_2 \cdot n$, für eine Konstante C_2
  - Die erwartete Laufzeit ist $\boldsymbol{\geq C_3 \cdot n \cdot n!}$

    Die Frage ist: geht es noch langsamer?

Andere Sortierverfahren 7/8

- BogoBogoSort
  - Ersetze Schritt 1 (Prüfung, ob Eingabe sortiert) durch einen rekursiven Algorithmus wie folgt:

    1. Mache eine Kopie des Feldes
    2. Sortiere die ersten n – 1 Elemente rekursiv
    3. Prüfe, ob das n-te Element größer ist, als das das letzte Element der rekursiv sortierten n – 1 Elemente
    4. Falls nicht, permutiere die Element zufällig und gehe zu Schritt 2
  - ÜB2 Zusatzaufgabe: schätzen Sie die Laufzeit ab

    Schon auf Eingaben der Größe 7 wird es nicht fertig

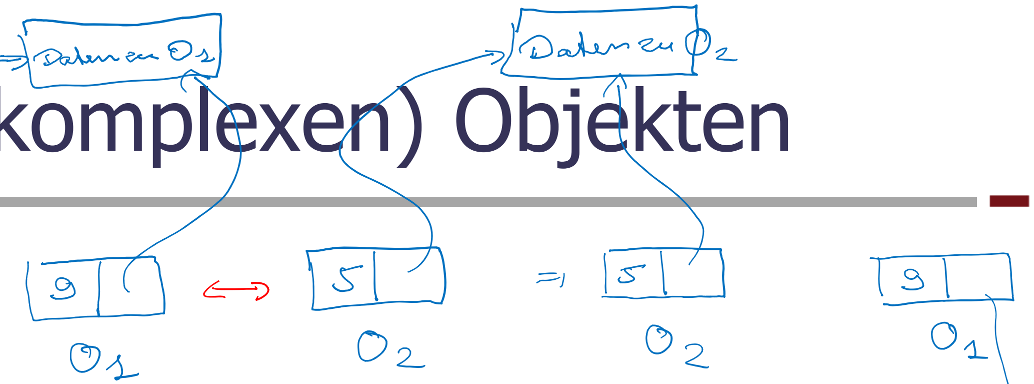
Andere Sortierverfahren 8/8

- DropSort
  - Gehe von links nach rechts durch das Feld
  - Wenn eine Element kleiner ist als das vorhergehende, wird es einfach nicht mit ausgegeben
  - Die Eingabe ist dann garantiert immer sortiert, es fehlen nur vielleicht ein paar Elemente
  - Einen solchen Algorithmus nennt man **"lossy"**

    Das Prinzip wird zum Beispiel auch bei der Kompression von Bildern verwendet (JPEG Format)

    Warum also nicht auch beim Sortieren?

Sortieren von (komplexen) Objekten



- **Motivation**
  - Meistens will man nicht einfach nur Zahlen sortieren, sondern komplexere Objekte nach bestimmten Werten

    Zum Beispiel: Studierende nach Punktzahl
  - Dann muss man aufpassen, dass man nicht bei jedem Vergleich zwei (evtl. große) Objekte hin- und her kopiert
  - Lösung: in dem Objekt steht außer dem Wert, nach dem sortiert wird, nur **ein Zeiger** auf die anderen Daten

    Dann müssen bei jedem Vertauschen nur die beiden Werte und die beiden Zeiger kopiert werden

Sortieren in Linearzeit 1/3

- ZeroOneSort
  - Geht Sortieren auch schneller als $n \cdot \log n$?
  - Ja, zum Beispiel wenn alle Elemente nur 0 oder 1 sind
  - Dann kann man einfach die 0en und 1en zählen

    Dazu schreiben wir gerade ein Programm ZeroOneSort

$0, 1, 1, 0, 0, 1, 0$

$\#0 = 4 \,, \quad \#1 = 3$

$$\underbrace{0, 0, 0, 0}_{4 \text{ mal}}, \underbrace{1, 1, 1}_{3 \text{ mal}}$$

Sortieren in Linearzeit 2/3

- CountingSort
  - Die Idee klappt auch noch, wenn die Elemente aus dem Bereich 0 .. n – 1 sind

Dazu schreiben wir gerade ein Programm CountingSort

5, 3, 1, 5, 2, 3, 0

n = 7

#0	#1	#2	#3	#4	#5	#6
I	I	I	II		II	

↓ Ausgabe

$\underbrace{0}_{I}\ \underbrace{1}_{I}\ \underbrace{2}_{I}\ \underbrace{3\ \ 3}_{II}\ \underbrace{5\ \ 5}_{II}$

Sortieren in Linearzeit 3/3

- Laufzeit
  - Für beide Verfahren gilt $\mathbf{T(n) \leq C \cdot n}$ ... für ein $C > 0$

    Man geht einmal über das Eingabefeld zum "Zählen", und dann gibt man die (gleich große) Ausgabe aus
  - Ist das vielleicht sogar für beliebige Eingaben möglich?

    CountingSort braucht ein Feld der Größe m für Zahlen aus dem Bereich 0 .. m-1

    Für $m \gg n$ ist die Laufzeit (und der Platzverbrauch) dann proportional zu m, und nicht zu n

Untere Schranke Sortieren 1/12

- Vergleichsbasiertes Sortieren
  - ZeroOneSort und CountingSort sortieren die Elemente nicht durch "Umsortieren", sondern durch "Zählen"
  - Wir wollen jetzt zeigen: wenn man nur "Umsortieren" zulässt, geht es tatsächlich nicht schneller als $n \cdot \log n$
  - Dazu müssen wir erst mal genauer fassen, was es heißt, dass ein Algorithmus "nur umsortiert"

Untere Schranke Sortieren 2/12

- Vorbetrachtung 1
  - Wir werden uns bei unserem Beweis auf Algorithmen **von einer bestimmten Art** beschränken

    Wir werden sehen: weil das die Argumentation erleichtert
  - Nehmen wir an, ein Algorithmus **A** ist nicht von dieser Art, aber es gibt einen Algorithmus **A'** von der Art für den gilt:

    A ist braucht höchstens $\leq C_1 \cdot n$ Operationen mehr als **A'**

    Die Ausgabe von **A** kann mit $\leq C_2 \cdot n$ Operationen in die Ausgabe von **A'** überführt werden
  - Dann gilt $T_{A'}(n) \geq n \cdot \log n \quad \Rightarrow \quad T_A(n) \geq n \cdot \log n$

    Wäre **A** schneller, könnten wir auch **A'** schneller machen

Untere Schranke Sortieren 3/12

σ 4 3 1 2 5

7, 5, 1, 3, 17

1, 3, 5, 7, 17

1 2 3 4 5

- Vorbetrachtung 2
  - Bisher haben unsere Algorithmen für Eingabe $x_1, \ldots, x_n$ diese Zahlen in sortierter Reihenfolge ausgegeben
  - Wir wollen im Folgenden Algorithmen betrachten, die stattdessen eine Permutation σ der Zahlen $1, \ldots, n$ ausgeben, so dass $x_{\sigma(1)} \leq x_{\sigma(2)} \leq \ldots \leq x_{\sigma(n)}$

i	1	2	3	4	5
$\sigma(i)$	4	3	1	2	5

  - **Beobachtung:** alle unseren bisherigen Algorithmen können dahingehend abgewandelt werden, ohne dass sich die Anzahl Operationen um mehr als $C \cdot n$ ändert

    Das gilt insbesondere für ZeroOneSort und CountingSort

Untere Schranke Sortieren 4/12

- Vorbetrachtung 3
  - In einem Programm (Python, Java, C++) können an diversen Stellen Verzweigungen auftreten

    while (…) { … }

    for (…) { … }

    if (…) { … } else { … }
  - Ohne Beschränken der Allgemeinheit seien all diese Verzweigungen von der Form if (…) { …} else {… }
  - Zum Beispiel ist while (EXPR) { … } äquivalent zu:

    while (true) { if (EXPR) { … } else { break; } }

Untere Schranke Sortieren 5/12

- Vorbetrachtung 4
  - Betrachten wir die Folge von Entscheidungen in den if (…) { … } else { … } Teilen im Ablauf eines Programms
  - Dann entspricht jeder Ablauf einer Folge **IEEIIIEIIIE**…

    I = if-Teil wird ausgeführt, **E** = else-Teil wird ausgeführt
  - Ein Algorithmus heißt **vergleichsbasiert**, wenn die I/E Folge die Ergebnispermutation **eindeutig** bestimmt
  - MinSort, MergeSort, QuickSort, HeapSort sind allesamt vergleichsbasiert bzw. können dazu gemacht werden
  - ZeroOneSort und CountingSort sind es nicht, und können auch nicht dahingehend abgeändert werden

Untere Schranke Sortieren 6/12

- ZeroOneSort ist nicht "vergleichsbasiert"
  - Hier sind zwei Eingaben mit gleicher I/E Folge aber verschiedenen Ergebnispermutationen

  Wir lassen dabei die I und E von den "for" Schleifen weg, die hängen ja sowieso nicht von der Eingabe ab

$$\sigma_1 \quad \overset{1}{0}, \overset{4}{1}, \overset{5}{1}, \overset{2}{0}, \overset{3}{0} \qquad \text{IIIEE}$$

$$\neq$$

$$\sigma_2 \quad \overset{1}{0}, \overset{4}{1}, \overset{2}{0}, \overset{3}{0}, \overset{5}{1} \qquad \text{IIIEE}$$

Untere Schranke Sortieren 7/12

- MinSort ist "vergleichsbasiert"
  - Beispiel: zwei Eingaben mit gleicher I/E Folge und gleicher Permutation, und dritte Eingabe mit anderer I/E Folge und anderer Permutation

    Wieder ohne die Is und Es von den "for" Schleifen

Untere Schranke Sortieren 8/12

- Beweis untere Schranke, Teil 1
  - Wir betrachten jetzt einen beliebigen vergleichsbasierten Algorithmus **A**, der für eine Eingabe der Größe n eine sortierende Permutation σ der Zahlen 1, …, n ausgibt
  - Sei T(n) eine obere Schranke für die Anzahl der von **A** benötigten Operationen auf einer Eingabe der Größe n

    Dann gibt es höchstens T(n) Verzweigungs-Anweisungen
  - Der Algorithmus gibt also für Eingabegröße n höchstens $2^{T(n)}$ verschiedene Permutationen aus

$\underbrace{\text{IEIIIEE} \dots \text{I}}_{x \le T(n) \text{ viele}}$

$2^x \le 2^{T(n)}$ Möglichkeiten

Untere Schranke Sortieren 9/12

- Beweis untere Schranke, Teil 2
  - Ein korrekter Algorithmus muss für Eingabegröße n alle möglichen Permutationen erzeugen können, das sind $n!$

    Wenn er eine Permutation nicht erzeugen könnte, würde er für die Eingabe, die genau diese Permutation zum Sortieren benötigt, nicht das richtige Ergebnis liefern
  - Auf der vorherigen Folie hatten wir gesehen, dass bei Laufzeit $\leq T(n)$ höchstens $2^{T(n)}$ Permutationen erzeugt werden können
  - Wäre $2^{T(n)} < n!$, würden nicht alle Permutationen erzeugt werden können und der Algorithmus wäre nicht korrekt
  - Es muss also $2^{T(n)} \geq n!$ sein, oder äquivalent $T(n) \geq \log_2 (n!)$

Untere Schranke Sortieren 10/12

- Abschätzung von $\log_2 (n!)$
  - Dafür wird oft die Stirling-Formel benutzt:

    $n! \geq \text{sqrt}(2 \cdot \pi \cdot n) \cdot (n/e)^n$
  - Wir können das aber auch (weniger genau, aber ausreichend) elementar-mathematisch abschätzen:

    $n! \geq (n/2)^{n/2}$
  - Daraus folgt:

    $\log2 (n!) \geq \log_2 (n/2)^{n/2} \geq n/2 \cdot \log_2 (n/2)$

$= \log_2 n - \underbrace{\log_2 2}_{=1}$

$\geq \frac{1}{2} \cdot \log_2 n \quad \forall n \geq 4$

$\geq n/4 \cdot \log_2 n$

$6! = 1 \cdot 2 \cdot 3 \cdot \underset{\geq 3}{4} \cdot \underset{\geq 3}{5} \cdot \underset{\geq 3}{6} \geq 3^3$

$5! = 1 \cdot 2 \cdot \underset{\geq 2.5}{3} \cdot \underset{\geq 2.5}{4} \cdot \underset{\geq 2.5}{5} \geq 2.5^3 \geq 2.5^{2.5}$

Untere Schranke Sortieren 11/12

- Beweis untere Schranke, Zusammenfassung
  - Wir haben gezeigt:

    Sei $T(n)$ eine obere Schranke für einen Algorithmus, der n Elemente vergleichsbasiert sortiert

    Dann ist $T(n) \geq \log_2 (n!) \geq \frac{1}{4} \cdot n \cdot \log_2 n$ für $n \geq 4$

Untere Schranke Sortieren 12/12

- Fazit
  - Unter den vergleichsbasierten Algorithmen sind also QuickSort, MergeSort, HeapSort alle optimal !
  - Aber in der Praxis zählen auch (unter anderem):

    Konstante Faktoren: zum Beispiel ist $10 \cdot n \cdot \log n$ offensichtlich 10 mal langsamer als $n \cdot \log n$

    Komplexe oder komplizierte Implementierungen haben typischerweise viel höhere Konstanten

    Cache-Effizienz: der Zugriff auf aufeinanderfolgende Elemente im Speicher ist billiger als wenn "verstreut"

    Spielt vorherrschende Rolle bei großen Datenmengen
  - Zu diesen Aspekten in einer späteren Vorlesung mehr !

Literatur / Links

- Untere Schranke
  - Mehlhorn/Sanders: 5.3 A Lower Bound [for Sorting]
- Laufzeitanalysen von QuickSort
  - Wikipedia: QuickSort#Formal_analysis