Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 2b, Mittwoch, 24. April 2013
(Fortsetzung Laufzeitanalyse + untere Schranke)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Laufzeitanalyse Sortieren ... Fortsetzung
  - Wir haben gesehen

    QuickSort im schlechtesten Fall: $T(n) \geq C_1 \cdot n^2$

    QuickSort im besten Fall: $T(n) \leq C_2 \cdot n \cdot (1 + \log_2 n)$
  - Heute zeigen wir dann noch

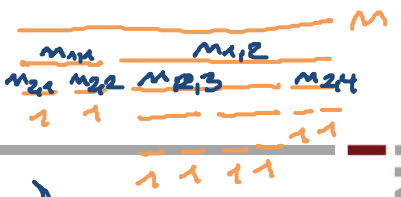
    QuickSort im "normalen" Fall: $T(n) \leq C_3 \cdot n \cdot (1 + \log_2 n)$
  - Außerdem

    Andere Sortierverfahren (kurzer Überblick)

    Geht Sortieren auch ganz linear, also mit $T(n) \leq C_4 \cdot n$?

    Untere Schranke von $\geq C_5 \cdot n \cdot \log_2 n$ wenn vergleichsbasiert

Laufzeitanalyse QuickSort 4/6



- Der allgemeine Fall
  - Rekursionstiefe k: ℓ_k Teilfelder mit Größen $n_{k,1}$, $n_{k,2}$, … , n_{k,ℓ_k}
  - Dann gibt es eine Konstante $A > 0$ so dass gilt:

    $T(n) \leq T(n_{1,1}) + T(n_{1,2}) + A \cdot n$

    $T(n_{1,1}) + T(n_{1,2}) \leq T(n_{2,1}) + \ldots + T(n_{2,\ell_2}) + A \cdot n$

    $T(n_{2,1}) + \ldots + T(n_{2,\ell 2}) \leq T(n_{3,1}) + \ldots + T(n_{3,\ell_3}) + A \cdot n$

    Und so weiter …
  - Mit vollständiger Induktion folgt dann (Ü2, Aufgabe 3):

    $T(n) \leq T(n_{k,1}) + \ldots + T(n_{k,\ell k}) + k \cdot A \cdot n$

Laufzeitanalyse QuickSort 5/6

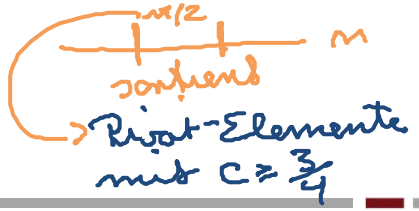
UNI FREIBURG

- Der allgemeine Fall
  - Nehmen wir jetzt an, jedes Pivot-Element teilt Feld der Größe m in zwei Teilfelder $\leq c \cdot m$ für ein $c < 1$ (bester Fall: $c = ½$)
  - Dann haben auf Rekursionstiefe k alle Teilfelder Größe $\leq c^k \cdot n$
  - Dann ist ab Rekursionstiefe $k \geq \log_{1/c} n$ kein Teilfeld mehr > 1
  - Dann gibt es eine Konstante $C > 0$ so dass gilt:

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$

    Also wie im besten Fall, nur mit einer anderen Konstante
  - **Beweis:** Übungsblatt 2, Aufgabe 3

Laufzeitanalyse QuickSort 6/6



UNI FREIBURG

- Der "durchschnittliche" Fall (engl. "average case")
  - **Annahme:** das Pivot-Element wird zufällig gewählt

    Wenn die Eingabe-Elemente zufällig sind, ist auch ein fest gewähltes Pivot-Element (z.B. das erste) zufällig
  - Pro Pivot-Element gilt dann mit Wahrscheinlichkeit ½ die Annahme von der Folie vorher mit $c = \frac{3}{4}$
  - Im Erwartungsfall ist man also nach spätestens der doppelten Rekursionstiefe wie auf der Folie vorher fertig
  - Auch im "average case" gibt es also eine Konstante $C > 0$, so dass gilt:

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$
  - Solche probabilistischen Analysen machen wir in einer späteren Vorlesung noch genauer

Andere Sortierverfahren 1/3

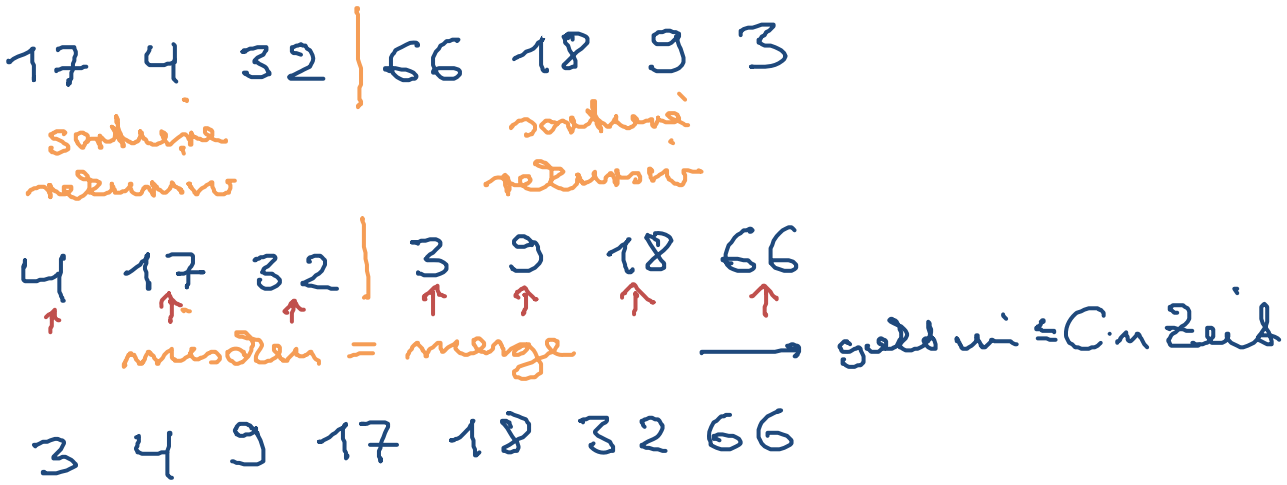
UNI FREIBURG

- MergeSort
  - Auch MergeSort teilt das Feld rekursiv auch, aber:

    Immer in (fast) genau zwei Hälften

    Es gibt keinen "divide" Schritt vor den rek. Aufrufen

    Dafür müssen hinterher die beiden (rekursiv) sortieren Hälften noch zusammen "gemischt werden"



Andere Sortierverfahren 2/3

UNI FREIBURG

- MergeSort, Fortsetzung
  - Die Laufzeit ist dann **in jedem Fall** wie bei QuickSort im besten Fall, nämlich

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$ für eine Konstante $C > 0$
  - **Anmerkung:** Trotzdem wird man in der Praxis QuickSort bevorzugen, weil

    … bei geigneter Wahl des Pivots hat auch QuickSort praktisch immer $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$

    … man bei QuickSort keine Felder umkopieren muss, sondern alles "in place" machen kann, das spart Zeit und führt zu einem kleineren Faktor C

Andere Sortierverfahren 3/3

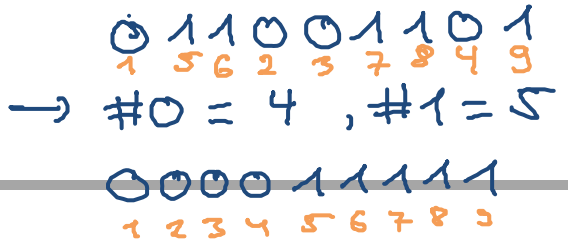
UNI FREIBURG

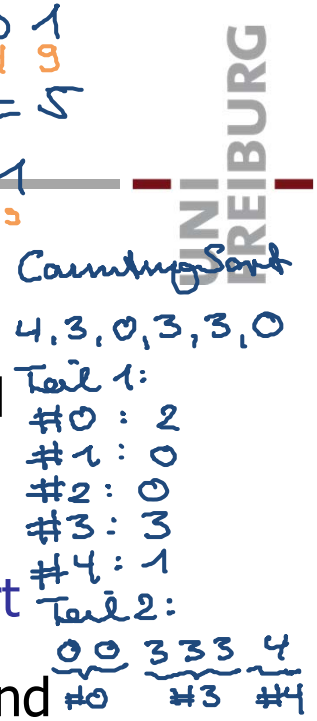
- HeapSort
  - HeapSort benutzt einen binären Heap zum Sortieren
  - Das ist eine Datenstruktur, die aus einer Menge von n Elementen mit $\leq C \cdot \log n$ Operationen das kleinste Element extrahieren und entfernen kann

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

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$ für eine Konstante $C > 0$
  - In der Praxis ist das C etwas kleiner (= besser) als das von MergeSort, aber etwas größer (= schlechter) als das von QuickSort

Sortieren in Linearzeit





- 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
  - Die Idee klappt auch wenn die Elemente aus 0..n-1 sind
  - Dazu schreiben wir gerade ein Programm CountingSort
  - Für beide Verfahren gilt $T(n) \leq C \cdot n$ ... für ein $C > 0$
  - Ist das vielleicht sogar für beliebige Eingaben möglich?

    CountingSort braucht ein Feld der Größe m für Zahlen aus dem Bereich 0..m-1, das lässt sich also nicht so einfach auf beliebige Eingaben verallgemeinern ...

Untere Schranke Sortieren 1/8

UNI FREIBURG

- 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 2/8

UNI FREIBURG

- 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:

    … er läuft genauso schnell wie **A**, bis auf $\leq C_1 \cdot n$ Operat.

    … die Ausgabe von **A** kann leicht, d.h. mit $\leq C_2 \cdot n$ Operat. in die Ausgabe von **A'** überführt werden
  - Wenn wir dann zeigen $T_{A'}(n) \geq n \cdot \log n$

    dann gilt auch $T_A(n) \geq n \cdot \log n$

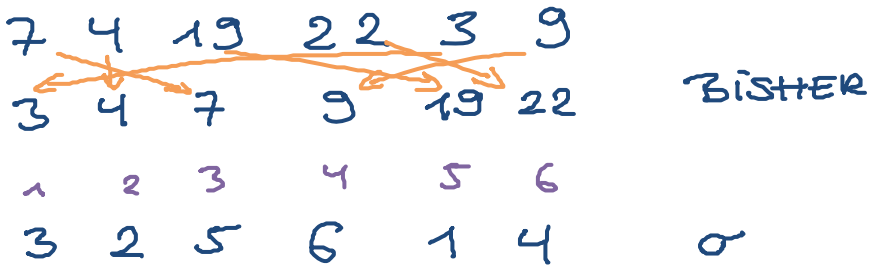
    Wäre **A** schneller, könnten wir auch **A'** schneller machen

Untere Schranke Sortieren 3/8

UNI FREIBURG

- 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 $x_{\sigma(1)} \leq x_{\sigma(2)} \leq \ldots \leq x_{\sigma(n)}$
  - Beobachtung: alle unseren bisherigen Algorithmen können dahingehend abgewandelt werden, ohne dass sich die Anzahl Operationen um mehr als $C \cdot n$ ändert

    Das gilt auch für ZeroOneSort und CountingSort !



Untere Schranke Sortieren 4/8

UNI FREIBURG

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

    while (…) { … }

    for (…) { …}

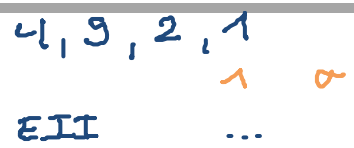
    if (…) { … } else { … }
  - Ohne Beschränken der Allgemeinheit seien all diese Verzweigungen von der Form if (…) { …} else {… }
  - Zum Beispiel kann man jede while Schleife

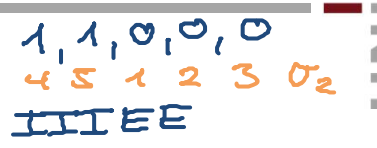
    while (…) { … }

    äquivalent umformen zu

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

Untere Schranke Sortieren 4/8





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

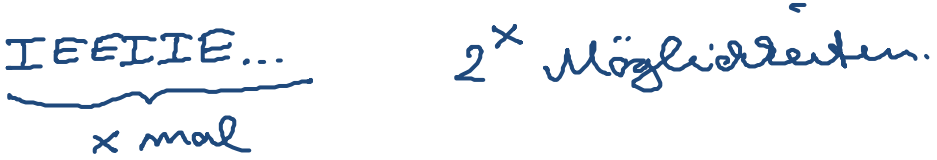
    I = if-Teil wird ausgeführt, **E** = else-Teil wird ausgeführt
  - Ein Algorithmus heißt **vergleichsbasiert**, wenn diese Folge die Permutation, die am Ende ausgegeben wird, **eindeutig** bestimmt
  - MinSort, QuickSort, MergeSort, HeapSort sind alle vergleichsbasiert bzw. können leicht dazu gemacht werden
  - ZeroOneSort und CountingSort sind es nicht, und können auch nicht dahingehend abgeändert werden

    Man muss etwas nachdenken, um das zu sehen !

Untere Schranke Sortieren 5/8

UNI FREIBURG

- 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 $1, \ldots, n$ ausgibt
  - Sei $T(n)$ so, dass für jede Eingabe der Größe n die Anzahl der von **A** benötigten Operationen $\leq T(n)$ ist
  - Insbesondere gibt es dann höchstens $T(n)$ Verzweigungs-Anweisungen
  - Der Algorithmus gibt also für Eingabegröße n höchstens $2^{T(n)}$ verschiedene Permutationen aus



Untere Schranke Sortieren 6/8

UNI FREIBURG

- 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 7/8

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

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

$$\log_2(n!) \quad ; \quad n! = n \cdot (n-1) \cdot \ldots \cdot 1 = \prod_{i=1}^{n} i$$

$$= \log_2(n \cdot (n-1) \cdot \ldots \cdot 1)$$

$$= \log_2 n + \log_2(n-1) + \ldots + \log_2 1$$

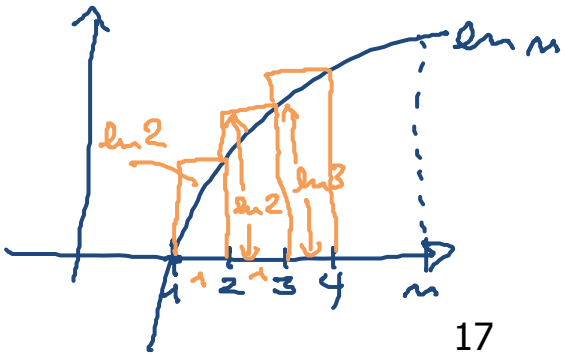
$$= \sum_{i=1}^{n} \log_2 i \overset{i \geq 2}{=} \sum_{i=1}^{n} \frac{\ln i}{\ln 2} = \frac{1}{\ln 2} \sum_{i=1}^{n} \ln i$$

$$\geq \frac{1}{\ln 2} \int_1^n \ln x \, dx = \frac{1}{\ln 2} \underbrace{x \cdot \ln x - x \Big|_1^n}_{n \cdot \ln n - n + 1 = n \cdot (\ln n - 1) \geq 0}$$

$$\geq n(\log_2 n - 1)$$

$\ln = \log_e$

$e = 2.71828\ldots$



Untere Schranke Sortieren 8/8

UNI FREIBURG

- Beweis unteren Schranke, Zusammenfassung
  - Wir haben gezeigt:

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

    Dann ist $T(n) \geq \log_2 (n!) \geq ½ \cdot n \cdot \log_2 n$ für $n \geq 3$
  - Unter den vergleichsbasierten Algorithmen sind also QuickSort, MergeSort, HeapSort alle optimal !
  - **Aber:** in der Praxis zählen auch (unter anderem)

    Konstante Faktoren ... n log n ist 10x schneller als 10 n log n

    Cache-Effizienz ... d.h. Lokalität der Speicherzugriffe
  - Zu diesen Aspekten in einer späteren Vorlesung mehr !

Literatur / Links

UNI FREIBURG

- Weiterführende Literatur (bei Interesse)
  - Analyse QuickSort für zufällig gewähltes Pivot-Element:

    Cormen/Leiserson/Rivest: II.8.4 Analysis of Quicksort

    http://en.wikipedia.org/wiki/Quicksort#Formal_analysis
  - Untere Schranke für vergleichsbasiertes Sortieren

    Mehlhorn/Sanders: 5.3 A Lower Bound [for Sorting]