Algorithmen und Datenstrukturen (für ESE) WS 2011 / 2012

Vorlesung 1, Dienstag, 25. Oktober 2011
(Einführung, Organisatorisches, Sortieren)

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

UNI FREIBURG

Die guten Nachrichten zuerst

UNI FREIBURG

- Nächste Woche **keine** Vorlesung
  - Wegen Allerheiligen
  - Dafür heute eventuell etwas länger
  - Und ein Übungsblatt für zwei Wochen, dazu später mehr

Allgemeines zu dieser Vorlesung 1/2

UNI FREIBURG

- Thema
  - Letztes Semester haben Sie die Grundzüge des Programmierens gelernt (die meisten in Java)
  - Fragen der Performanz spielten noch kaum eine Rolle
  - Genau darum geht es jetzt in dieser Vorlesung
    - Wie schnell ist mein Programm
    - Wie kann ich es schneller machen
    - Wie kann ich beweisen, dass es immer so schnell läuft
    - Manchmal geht es auch um Sparsamkeit im Platzverbrauch oder andere Ressourcen, aber meistens um **Zeit**

Allgemeines zu dieser Vorlesung 2/2

UNI FREIBURG

- Art und Weise
  - Großer Praxisbezug (wie auch in meiner C++ Vorlesung)
  - Aber auch Theorie (sofern sie der Praxis nutzt)
  - Übungsblätter entsprechend teils theoretisch, teils praktisch
  - Zum Ablauf des Übungsbetriebs gleich mehr ...
- Aufwand
  - Es gibt für die Veranstaltung 4 ECTS Punkte
  - Das entspricht 120 Arbeitsstunden für das ganze Semester
  - 14 Vorlesungen à 6 Stunden Arbeit + Klausur am Ende
  - Also ca. 4 Stunden / Übungsblatt (es gibt jede Woche eins)
  - Für das 1. Übungsblatt haben Sie zwei Wochen Zeit

Übungen

UNI FREIBURG

- Übungsblätter
  - Um an der Klausur teilnehmen zu können, brauchen Sie mindestens 50% der Punkte in den Übungsblättern
  - Wer die Übungsblätter alle macht, braucht am Ende kaum noch was zu lernen für die Klausur
  - Abgabe / Korrektur über unser SVN, gleich mehr dazu
- Übungsgruppen
  - Es gibt **eine** zentrale Übung zur Veranstaltung, nach Bedarf
  - Termin schauen wir, wenn sich alle angemeldet haben
  - Die Tutoren sind: Manuel Bührer, Katja Faist, Sebastian Sester
  - Mitverantwortlich für die Übungsblätter ist: Björn Buchhold

Note / Klausur

- Die **Endnote** ergibt sich folgendermaßen
  - Die Punkte aus den Übungsblättern werden auf max. 20 Punkte skaliert → Ü
  - In der Abschlussklausur wird es 4 Aufgaben à jeweils 20 Punkte geben → K1, K2, K3, K4
    - Termin: Freitag, 30. März 2012, 14:15 – 15:45 Uhr
  - Von Ü, K1, K2, K3, K4 werden die besten 4 genommen und aufsummiert (also maximal 80 Punkte)
  - Aus dieser Summe wird die Endnote ermittelt
  - Schummeln / Plagiat bei den Übungsblättern bzw. bei der Klausur hat wie immer Nicht-Bestehen zur Folge

    (einmal reicht, das tut man ja nicht aus Versehen)

Daphne

UNI FREIBURG

- Daphne ist unser **Kursverwaltungssystem**
  - Link auf dem Wiki zum Kurs, **bitte anmelden!**
  - In Daphne haben Sie eine Übersicht über folgende Infos
    - Wer ihr/e Tutor/in ist
    - Ihre Punkte in den Übungsblättern
    - Info zum aktuellen Übungsblatt
    - Link zum SVN ... gleich mehr dazu
    - Link zu unseren Coding Standards ... gleich mehr dazu
    - Link zu unserem Build System ... gleich mehr dazu

SVN

UNI FREIBURG

- SVN = **Subversion** http://subversion.apache.org/
  - Dateien liegen auf einem zentralen Server, in einem sogenannten repository, die typische Operationen sind
    - Update: neuste Version vom Server ziehen
    - Commit: letzte Änderungen auf den Server hochladen
  - Vollständige Historie von allen Änderungen an den Dateien
  - Insbesondere nützlich für das Schreiben von Code
  - Wir benutzen das hier für
    - die Abgaben Ihrer Übungsblätter (Code + alles andere)
    - das Feedback von Ihrem Tutor / Ihrer Tutorin
    - Vorlesungsdateien / Musterlösungen
  - Ich werde es Ihnen heute einfach einmal vormachen ...

Coding Standards

UNI FREIBURG

- Neben dem eigentlichen Thema (AlgoDat)
  - ... sollen Sie auch weiterhin gutes Programmieren lernen
  - Dazu ein paar Coding Standards
    - Gute Unit Tests für alle nicht-trivialen Methoden (JUnit)
      - **Grund :** Siehe nächste Folie
    - Befolgen eines Stylesheets (checkstyle)
      - **Grund:** Damit ihr Code lesbar und verständlich ist
    - Standardisierte Build-Framework (ant)
      - **Grund:** Damit wir nicht bei jeder Abgabe getrennt schauen müssen, wie man den Code testet / ausführt
  - Ich werde Ihnen das heute alles einmal vormachen ...

Unit Tests

UNI FREIBURG

- Warum Unit Tests
  - **Grund 1:** Eine nicht-trivial Methode ohne Unit Test ist mit hoher Wahrscheinlichkeit nicht korrekt
  - **Grund 2:** Macht das Debuggen von größeren Programmen **viel** leichter und angenehmer
  - **Grund 3:** Wir und Sie selber können automatisch testen ob Ihr Code das tut was er soll
- Was ist ein "guter" Unit Test
  - Ein Unit Test soll überprüfen ob eine Methode für eine gegebene Eingabe, die gewünschte Ausgabe berechnet
  - Für mindestens **eine typische** Eingabe
  - Für mindestes **zwei schwierige** "Grenzfälle", wenn es denn solche gibt (z.B. leeres Feld beim Sortieren)

Jenkins

UNI FREIBURG

- Jenkins ist unser **Build System**
  - Damit können Sie schauen, ob Ihr Code, so wie Sie ihn bei uns hochgeladen haben, kompiliert und läuft
  - Werde ich Ihnen auch vormachen **...**

Forum

- Es gibt ein **Forum** zur Veranstaltung
  - Link dazu auf dem Wiki
  - Bitte fragen Sie, wann immer etwas nicht klar ist
  - Nur keine Hemmungen, auch wenn Sie denken, die Frage ist blöd **...** ist sie meistens nicht
  - Und fragen Sie auf dem Forum, in den allermeisten Fällen interessiert das auch andere
  - Entweder ich oder Björn Buchhold oder einer der Tutoren wird dann möglichst schnell antworten

Themen

- Grundlegende Algorithmen und Datenstrukturen
  - Sortieren, dynamische Felder, assoziative Felder, Hashing, Prioritätswarteschlangen, Listen, Kürzeste Wege / Dijkstra's Algorithmus, Suchbäume, etc.
  - Siehe Wiki von der Vorlesung vom WS 2010/2011

    http://ad-wiki.informatik.uni-freiburg.de/teaching/AlgoDatEseWS1011
- Methodologisches
  - Laufzeitanalyse
  - O-Notation
  - Den einen oder anderen Korrektheits**beweis**
  - Die Mathematik, die wir in diesem Kurs verwenden, ist sehr "basic", aber es ist schon Mathematik, nicht nur "Rechnen"

Sortieren

- Problemdefinition
  - **Eingabe:** eine Folge von n Elementen $x_1, ..., x_n$
  - Sowie ein (transitiver) Vergleichsoperator $<$ der für zwei beliebige Elemente sagt, welches davon kleiner ist
  - **Ausgabe:** die n Elemente in gemäß diesem Operator sortierter Reihenfolge

$x < y \wedge y < z \Rightarrow x < z$

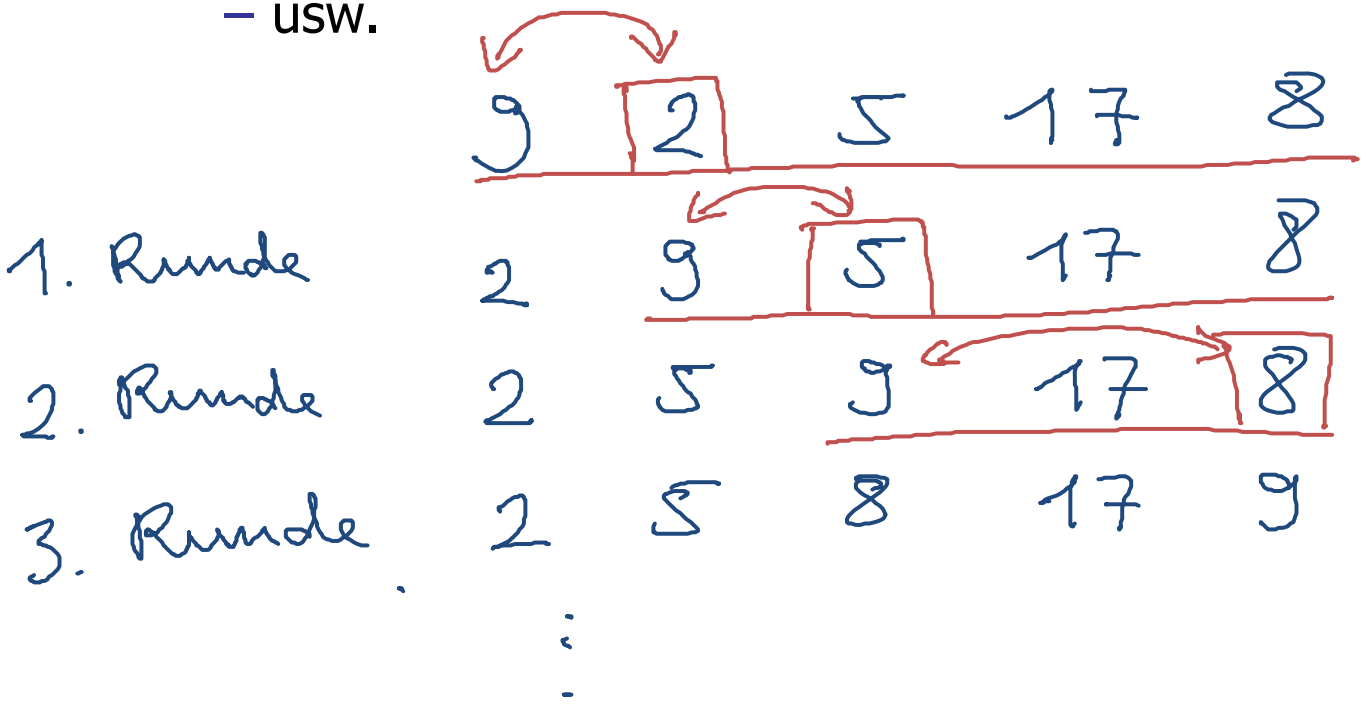
Eingabe: 5 2 8 17 9

Ausgabe: 2 5 8 9 17

- Wo braucht man Sortieren?
  - In praktisch **jedem** größeren Programm
  - Beispiel: Bauen eines Indexes für eine Suchmaschine

MinSort — Algorithmus

- Informaler Algorithmus
  - Finde das Minimum und setze es an die erste Stelle
  - Finde das Minimum im Rest und setze es an die zweite Stelle
  - Finde das Minimum im Rest und setze es an die dritte Stelle
  - usw.



MinSort — Programm

UNI FREIBURG

- ... schreiben wir jetzt zusammen
  - Bei der Gelegenheit zeige ich Ihnen dann auch gleich wie das geht mit
    - den Unit Tests (Junit)
    - unseren Coding Standards (checkstyle)
    - unserem Build Framework (ant)
    - unserem Build System (Jenkins)
    - unserem SVN

MinSort — Laufzeit

UNI FREIBURG

- Wie lange läuft unser Programm?
  - Wir testen das mal für verschiedene Eingabegrößen
  - Beobachtung: es wird "unproportional" langsamer, je mehr Zahlen sortiert werden
  - Können wir das genauer machen?
  - Wir zeichnen erstmal ein Schaubild
    - Das ist auch Teil des 1. Übungsblattes !
  - Wie können wir präziser (= mathematisch) fassen, was da passiert?

MinSort — Laufzeitanalyse

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
    - Mondphase
  - **Abstraktion 1:** Deshalb analysieren wir nicht die Laufzeit, sondern die Anzahl der (Grund)Operationen

Grundoperationen

- Hier eine unvollständige Liste
  - 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
  - Wie 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 — Anzahl Operationen

UNI FREIBURG

- Wieviele Operationen braucht MinSort?
  - **Abstraktion 2:** Wir zählen die Operationen nicht genau, sondern berechnen obere und (manchmal) untere Schranken
    - Grund: das erleichtert die Sache und wir haben ja eh schon 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 $A_1 / 4 \cdot n^2 \leq T(n) \leq A_2 / 2 \cdot n^2$

    wobei A_1 und A_2 irgendwelche Konstanten sind
  - Das nennt man "quadratische Laufzeit" (wegen dem n^2)

MinSort — Anzahl Operationen

- Beweis der Behauptung $A_1 / 4 \cdot n^2 \leq T(n) \leq A_2 / 2 \cdot n^2$

1. Runde : $\leq A \cdot n$ Operationen
2. Runde : $\leq A \cdot (n-1)$ "
3. Runde : $\leq A \cdot (n-2)$ "

$\vdots$

n. Runde : $\leq A \cdot 1$ Operationen

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

Quadratische Laufzeit

- Definition
  - Die Laufzeit T hängt von der Eingabegröße n ab
  - Es gibt Konstanten A_1 und A_2 mit $A_1 \cdot n^2 \leq T(n) \leq A_2 \cdot n^2$
- Betrachtungen dazu
  - Doppelt so große Eingabe → viermal so große Laufzeit
  - Unabhängig von den Konstanten wird das schnell sehr teuer
    - $A_1 = 1$ ns (1 einfache Anweisung ≈ 1 Nanosekunde)
    - $n = 10^6$ (1 Millionen Zahlen = 4 MB) [bei 4 Bytes/Zahl]
      - $A_1 \cdot n^2 = 10^{-9} \cdot 10^{12} = 10^3$ s = 16.7 Minuten
    - n = 10^9 (1 Milliarde Zahlen = 4 GB)
      - $A_1 \cdot n^2 = 10^{-9} \cdot 10^{18} = 10^9$ s = 31.7 Jahre

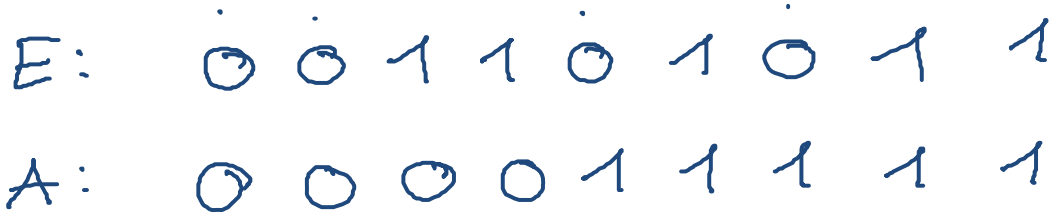
$(2n)^2 = 4 \cdot n^2$

Geht Sortieren auch schneller als quadratisch?

ZeroOneSort

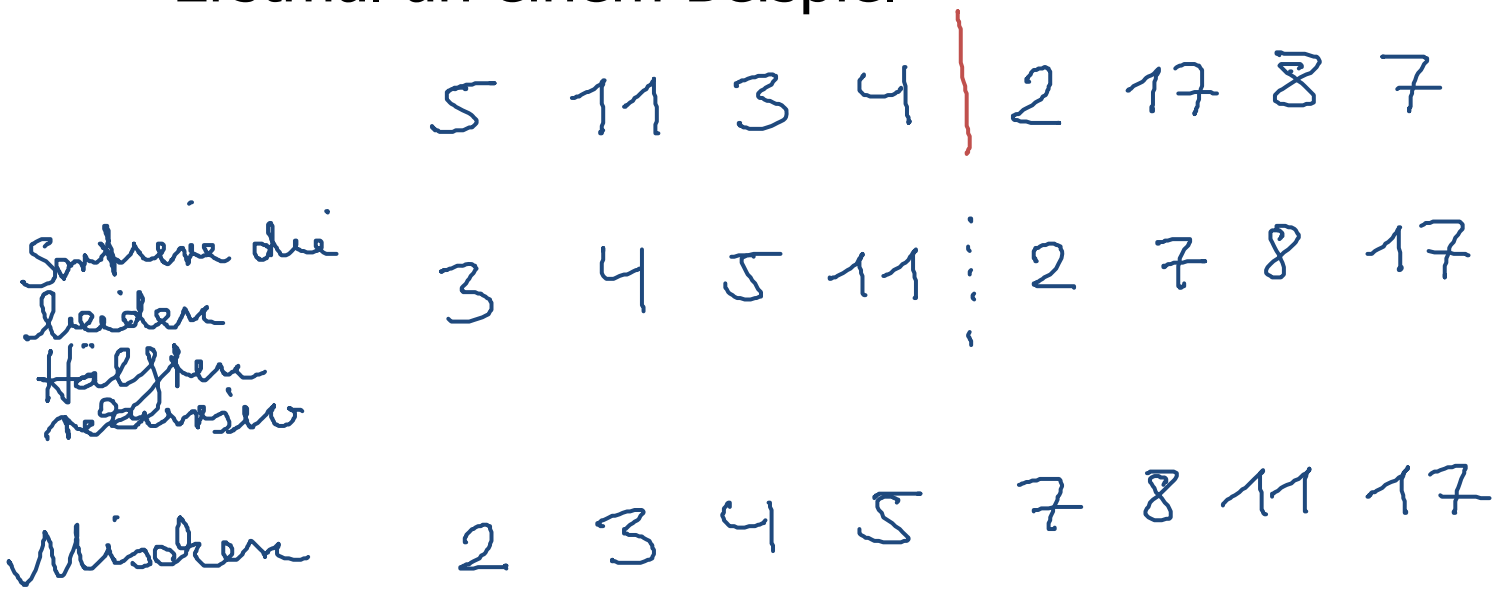
- Annahme: Eingabe besteht nur aus 0en und 1en
  - Gibt es dann einen schnelleren Algorithmus?
  - Ja, sogar mit linearer Laufzeit, das heißt

    $T(n) \leq A \cdot n$
  - Den implementieren wir gerade zusammen



MergeSort — Algorithmus

- Erstmal an einem Beispiel



MergeSort — Algorithmus

UNI FREIBURG

- Informale Beschreibung
  - Teile das Eingabefeld möglichst genau in der Mitte
    - bei ungerader Größe n: $\lfloor \frac{n}{2} \rfloor$ und $\lceil \frac{n}{2} \rceil$ (z. B. $n = 7$: 3 und 4)
  - Sortiere die beiden Hälften **rekursiv**
    - Ich hoffe, das haben Sie schon mal gehört, auf der nächsten Folie ein kleiner Crashkurs dazu
  - Jetzt müssen wir aus den beiden sortierten Hälften ein einziges sortiertes Feld machen
  - Das nennt man "Mischen" (Englisch: "merge")
  - Mischen geht in linearer Zeit ... siehe separate Folie gleich

Einschub: Rekursion 1/2

UNI FREIBURG

- Eine Funktion kann sich auch **selber** aufrufen
  - Man muss nur aufpassen, dass man dabei nicht in eine Endlosschleife gerät, z.B. so

```
int computeSum(int[] array) {
  return computeSum(array);
}
...
int[] array = { 3, 6, 7, 1, 3 };
int sum = computeSum(array);  // This call will never return.
```

Einschub: Rekursion 2/2

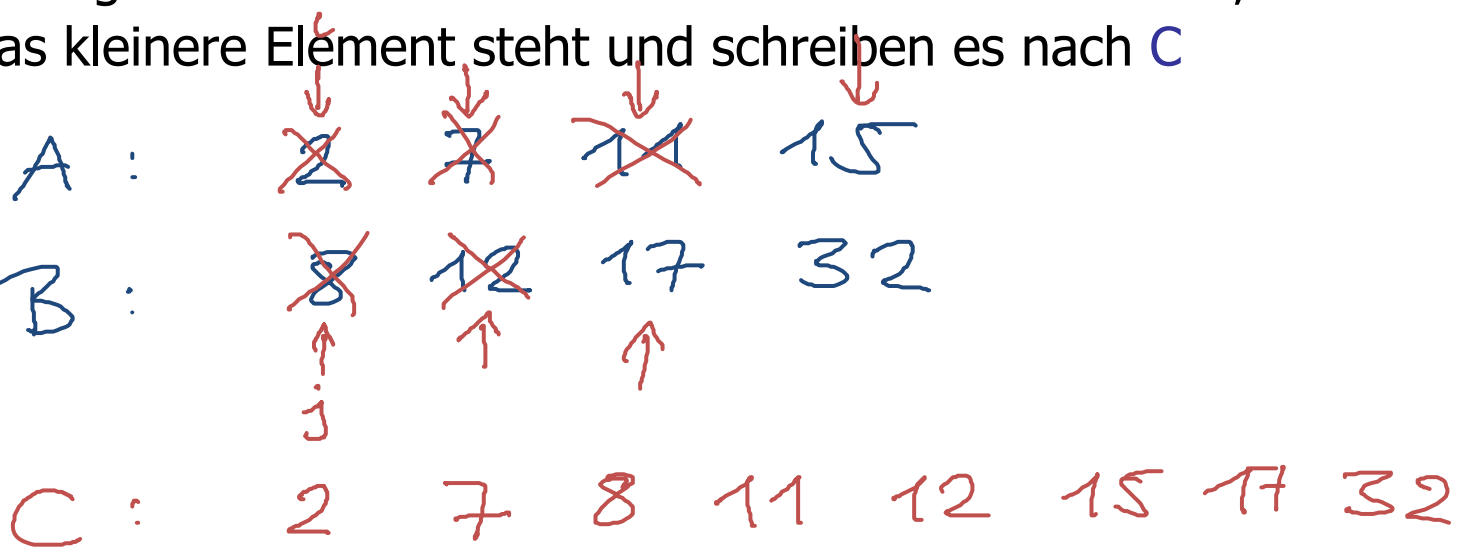
- Fortsetzung ...
  - Wenn die Rekursion irgendwann abbricht, aber kein Problem

```
int computeSum(int[] array) {
  if (array.length == 0) return 0;
  int last = array.pop();  // Removes last element.
  int sumRest = computeSum(array);  // Call for smaller array!
  return last + sumRest;
}
...
int[] array = { 3, 6, 7, 1, 3 };
int sum = computeSum(array);  // Will output 20, as expected.
```

  - Anmerkung: die Summe berechnet man natürlich einfacher iterativ, das hier war nur zur Veranschaulichung von Rekursion

Mischen — Algorithmus

- Informaler Algorithmus
  - Gegeben zwei Felder A und B, die schon sortiert sind
  - Die wollen wir zu einem sortierten Feld C mischen
  - Wir merken uns in jedem Feld eine Position, i für A und j für B, am Anfang sind beide am Anfang, also 0
  - Jetzt gehen wir immer in dem Feld weiter nach rechts, wo das kleinere Element steht und schreiben es nach C



MergeSort — Programm

UNI FREIBURG

- Das ist Ihre Übungsaufgabe
  - Erstmal eine Methode für Mischen
  - Und dann MergeSort
  - Nächste Woche Montag ist Allerheiligen, da gibt es keine Vorlesung und auch kein neues Übungsblatt
  - Deshalb haben Sie für dieses Übungsblatt zwei Wochen Zeit, bis zum 8. November um 16 Uhr
  - Tipp: fangen Sie nicht erst am 7. November damit an

MergeSort — Laufzeitanalyse

UNI FREIBURG

- Wir analysieren wieder die Anzahl der Operationen
  - Sei wieder T(n) diese Anzahl bei Eingabegröße n
  - Anders als bei MinSort können wir jetzt nicht so leicht eine Abschätzung für T(n) hinschreiben
  - Aber wir können relativ leicht eine **rekursive** Gleichung für T(n) formulieren, und zwar:

    $T(n) \leq T(\text{floor}(n/2)) + T(\text{ceil}(n/2)) + A \cdot n$
  - Wie können wir diese Rekursion auflösen?

MergeSort — Laufzeitanalyse

- Auflösen der Rekursion
  - $T(n) \leq T(\text{floor}(n/2)) + T(\text{ceil}(n/2)) + A \cdot n$
  - Wir nehmen erstmal der Einfachheit halber an, dass n eine Zweierpotenz ist, also $n = 2^k$ für ein $k \in \mathbf{N}$, dann
  - $T(n) \leq 2 \cdot T(n/2) + A \cdot n$

$$\leq 2 \cdot [2 \cdot T(n/4) + A \cdot n/2] + A \cdot n$$

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

$$\leq 4 \cdot [2 \cdot T(n/8) + A \cdot n/4] + 2 \cdot A \cdot n$$

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

$$\leq \ldots \qquad k = \log_2 n$$

$$\leq n \cdot T(1) + k \cdot A \cdot n$$

Einschub: Induktionsbeweis

UNI FREIBURG

- Prinzip
  - Man möchte beweisen, dass eine Aussage für alle natürlichen Zahlen gilt, z.B. unsere Schranke für T(n)
  - Dann hat ein Induktionsbeweis zwei Schritte
  - Wir zeigen, dass sie für T(1) gilt (Induktionsanfang)
  - Wir zeigen, dass sie für T(n) gilt, und **dürfen dabei benutzen**, dass die Aussage für T(1), ..., T(n-1) gilt (Induktionsschritt)
  - Wenn wir die beiden Sachen gezeigt haben, haben wir nach dem Prinzip der **vollständigen Induktion** gezeigt, dass die Aussage für **alle** natürlichen Zahlen n gilt

MergeSort — Laufzeitanalyse

- Induktionsbeweis
  - Behauptung: $T(n) \leq n \cdot T(1) + A \cdot n \cdot \log_2 n$
  - Induktionsanfang: für $n = 1$ ist die rechte Seite $= T(1)$
  - Induktionsschritt: seit jetzt n eine natürliche Zahl ≥ 2 und nehmen wir an, die Aussage gilt für $T(1), \ldots, T(n-1)$

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

$$\leq 2 \cdot \left[n/2 \cdot T(1) + A \cdot n/2 \cdot \log_2 (n/2) \right]$$

$$\leq n \cdot T(1) + A \cdot n \cdot \log_2 (n/2) + A \cdot n$$

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

$$\leq n \cdot T(1) + A \cdot n \cdot (\log_2 n - 1)$$

Laufzeit proportional zu n · log n

UNI FREIBURG

- Schauen wir uns wieder Zahlenbeispiele an
  - Nehmen wir also an, es gibt Konstanten A_1 und A_2 mit
    $A_1 \cdot n \cdot \log n \leq T(n) \leq A_2 \cdot n \cdot \log n$ für $n \geq 2$
  - Dann dauert es bei doppelt so großer Eingabe nur geringfügig mehr als doppelt so lange
  - Im Folgenden sei log der Logarithmus zur Basis 2
  - $A_1 = 1$ ns (1 einfache Anweisung ≈ 1 Nanosekunde)
  - $n = 2^{20}$ (≈ 1 Millionen Zahlen = 4 MB) [bei 4 Bytes/Zahl]
    - $A_1 \cdot n \cdot \log n = 10^{-9} \cdot 2^{20} \cdot 20$ s = 21 Millisekunden
  - $n = 2^{30}$ (≈ 1 Milliarde Zahlen = 4 GB)
    - $A_1 \cdot n \cdot \log n = 10^{-9} \cdot 2^{30} \cdot 30$ s = 32 Sekunden

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

Literatur / Links

- Allgemein zur Vorlesung
  - Cormen / Leiserson / Rivest: Introduction to Algorithms

    [Klassisches Lehrbuch zu Algorithmen und Datenstrukturen. Nur das Inhaltsverzeichnis ist online, muss man kaufen oder ausleihen.]

    http://mitpress.mit.edu/algorithms/
  - Mehlhorn / Sanders: Algorithms and Data Structures, The Basic Toolbox

    [Neueres Lehrbuch zu Algorithmen und Datenstrukturen, mit etwas praktischere Ausrichtung als der Cormen/Leiserson/Rivest. Das ganze Buch steht online!]

    http://www.mpi-inf.mpg.de/~mehlhorn/Toolbox.html
- Sortieren
  - In Mehlhorn/Sanders: 5. Sorting and Selection
  - In Cormen/Leiserson/Rivest: 1.3 MergeSort
  - Wikipedia hat Artikel zu MinSort und MergeSort