universität freiburg

Algorithmen und Datenstrukturen SS 2025

Vorlesung 2: Laufzeitanalyse MinSort & MergeSort

Dienstag, 29. April 2025

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen zum **Ü1** — MergeSort, Programm und Laufzeit
  - Einführungskurs Do+Mo — kurzer Erfahrungsbericht
  - Demografie — fünf Gruppen von Studierenden
  - Vorgucker **Ü2** — sehr schöne Theorie, siehe **Folie 6**
  - Die ersten Vorlesungen — sind besonders wichtig
- Laufzeitanalyse
  - Laufzeit allgemein — wie fasst man das mathematisch?
  - Laufzeit MinSort — "quadratische" Laufzeit
  - Laufzeit MergeSort — besser, aber auch nicht "linear"
  - Andere Sortierverfahren — ZeroOneSort und viele andere mehr

Erfahrungen mit dem Ü1

237 Anmeldungen in Daphne

- Zusammenfassung und Auszüge
  - Gute Einstiegsaufgabe, viel Zeit fürs Drumherum gebraucht, einige beim Programmieren etwas aus der Übung

    "Thema ziemlich interessant, Aufgabe war gut machbar"

    "Vorlesung war gut verständlich und gut strukturiert"

    "Die meiste Zeit hat eigentlich die Einrichtung gekostet"

    "Meine Programmierkenntnisse waren leicht eingerostet"

    "Laufzeit für MergeSort linear, für MinSort exponentiell" [oft]

    "Erstaunlicherweise keine Probleme, obwohl ich im SS 2024 AlgoDat abgebrochen hatte, weil MergeSort zu schwer war"

    "Ich hatte einen Platz bei Let's meet, aber Raum war schon voll"

    "Ich habe 3,5 Stunden gebraucht und es war schrecklich"

Einführungskurs für Linux, SVN, etc.

- Es gab zwei Termine (Donnerstag und Montag)
  - Es waren insgesamt ca. 80 Leute da (von ca. 250)
  - Es gab sehr sehr wenige Fragen
  - Ihre Kameras waren (fast) alle aus
  - Breakout-Räume wurden nicht in Anspruch genommen

  Wir hoffen mal, dass das bedeutet, dass Sie besonders selbständig sind und sich alles selber beibringen möchten

  Falls nicht, lernen Sie bitte schnellstmöglich, sich helfen zu lassen, wenn Sie eigentlich Hilfe brauchen

Demografie unseres Publikums hier

- Wir sehen **fünf** Gruppen von Studierenden

  Sie könnten sich das auch ohne Vorlesung selber beibringen

  Sie sollen sich bei uns nicht langweilen + noch was dazu lernen

  Sie können dem Stoff gut folgen, brauchen aber etwas Motivation

  Die wollen wir Ihnen geben + gute Erklärungen dazu

  Sie haben Schwierigkeiten, können es aber mit Mühe schaffen

  Wir wollen Ihnen helfen, so gut es geht

  Für Sie ist das hier noch eine Nummer zu groß (Voraussetzungen fehlen)

  Wir wollen Ihnen helfen, das möglichst früh zu erkennen

  Sie sind hier ohne besonderes Interesse oder Engagement dabei

  Stört uns nicht, solange Sie uns keine Arbeit oder Ärger machen

Vorgucker auf das Ü2

- Ein sehr schönes theoretisches Übungsblatt

  A1 Beweis von $T(n) \geq C' \cdot n^2$ für MinSort

  Analog zum dem Beweis von $T(n) \leq C \cdot n^2$ in der VL heute; aber nicht abschreiben, sondern verstehen + dann **selber machen**

  A2 Anzahl Bits der Binärdarstellung einer Zahl n beweisen

  Nicht schwer, aber eine super Aufgabe um zu üben, was ein Beweis ist und wie man mathematisch sauber beweist

  A3 Ein gegebenes Programm verstehen und analysieren

  Weiterführende Übung in Laufzeitanalyse und Beweisen

  A4 Eine optionale Knobelaufgabe (mit Hinweis gut machbar)

  Nutzen Sie **jetzt** die Gelegenheit, solche Aufgaben zu üben, so etwas in der Art kommt auch immer in der Klausur dran

Wichtigkeit der ersten Vorlesungen

- **Gerade am Anfang baut alles aufeinander auf**
  - Wenn Sie jetzt nicht richtig aufpassen, sind Sie spätestens in ein paar Wochen komplett abgehängt

    Das dann noch nachzuholen, schaffen vielleicht ein paar ganz wenige super Schlaue, aber die meisten nicht
  - Die allermeisten Probleme mit der Mathematik sind übrigens von genau der Art

    Wenn man schon mit Bruchrechnen Probleme hatte und den Logarithmus nicht richtig verstanden hat, hat man überhaupt keine Kapazität frei, die weiterführenden Sachen zu verstehen

    (dabei sind die weiterführenden Sachen oft erstaunlich einfach zu verstehen, wenn man die Grundlagen mal verstanden hat)

Laufzeitanalyse allgemein 1/5

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

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

    Doppelt so große Eingabe → mehr als doppelt so langsam

    MergeSort: die 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?
  - **Ideal:** Eine Formel, die uns für eine bestimmte Eingabe sagt, wie lange das Programm darauf läuft, z.B.

    Für Eingabe xyz läuft das Programm 2,37 Sekunden
  - **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
    - was für eine Programmiersprache benutzt wurde
    - welcher Compiler benutzt wurde
    - Jahreszeit, Mondphase, Raumzeitverkrümmung, ...

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
  - Die genau Anzahl von Operationen zu berechnen ist schwierig, aufwendig (oder gar unmöglich), und sowieso viel zu detailliert

    Wir haben ja eh schon abstrahiert von der exakten Laufzeit zu der Anzahl Operationen
  - Eine **Abschätzung** der Anzahl der Operationen macht das Informatik-Leben leichter und ist oft gut genug
  - 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

Laufzeitanalyse allgemein 5/5

- Abstraktion 3: Schranken pro Eingabegröße **n**
  - Oft hängt die Laufzeit vor allem von der **Größe** der Eingabe ab, und weniger 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

    $T(n) = \mathbf{max}\{ \#\text{Operationen für Eingabe } I : \text{Größe}(I) = n \}$

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

    $T(n) = \mathbf{min}\{ \#\text{Operationen für Eingabe } I : \text{Größe}(I) = n \}$

Laufzeitanalyse MinSort 1/5

- Erstmal: Abschätzungen für die Einzelteile
  - MinSort hat eine äußere und eine innere Schleife

```
def min_sort(array):
    ...                                          ≤ C_R
    for i in range(n – 1):
        ...                                      ≤ C_A
        for j in range(i + 1, n):
            ...                                  ≤ C_I
```

  - Seien **C_I, C_A, C_R Konstanten** (unabhängig von n), so dass:

    Kosten für einen Durchlauf der inneren Schleife ≤ C_I
    Kosten für einen Durchlauf äußere Schleife ohne innere ≤ C_A
    Kosten für den Rest ≤ C_R

    Beachte: Kosten können variieren, deswegen ≤ und nicht =

Laufzeitanalyse MinSort 2/5

- Wir beweisen: $T(n) \leq C \cdot n^2$ … für irgendeine Konstante C
  - Erstmal schätzen wir für jede Iterationen i der äußeren Schleife, die Anzahl Operationen der inneren Schleife ab

    Iteration 1: $T_1(n) \leq C_I \cdot (n-1) + C_A$

    Iteration 2: $T_2(n) \leq C_I \cdot (n-2) + C_A$

    …

    Iteration $n-1$: $T_{n-1}(n) \leq C_I \cdot 1 + C_A$

    $1 + 2 + \cdots + n$

    $n + n-1 + \cdots + 1$

    $(n+1) + (n+1) + \cdots + (n+1) = n \cdot (n+1)$

  - Damit können wir jetzt die Gesamtlaufzeit abschätzen

$$
\begin{aligned}
T(n) &\leq T_1(n) + T_2(n) + \cdots + T_{n-1}(n) + C_R \\
&\leq C_I \cdot \underbrace{(n-1 + n-2 + \cdots + 1)}_{= n \cdot (n-1)/2 \leq n^2} + C_A \cdot \underbrace{(n-1)}_{\leq n^2} + C_R \\
&\leq C_I \cdot n^2 + C_A \cdot n^2 + C_R \cdot \underbrace{n}_{\leq n^2} \\
&\leq \underbrace{(C_I + C_A + C_R)}_{=: C} \cdot n^2 \leq C \cdot n^2
\end{aligned}
$$

Laufzeitanalyse MinSort 3/5

- Es gilt auch: $T(n) \geq C' \cdot n^2$ ... für eine Konstante $C' > 0$
  - Der Beweis geht analog zu dem von der vorherigen Folie

    Das ist Aufgabe 1 vom Ü2 und eine gute Aufgabe, um diese Art von Beweis zu verstehen

    Schreiben Sie den Beweis von der vorherigen Folie nicht einfach ab, sondern versuchen Sie erst, ihn zu verstehen und beweisen Sie dann **selber** $T(n) \geq C' \cdot n^2$

Laufzeitanalyse MinSort 4/5

- Quadratische Laufzeit
  - Es gibt Konstanten C' und C , so dass gilt

    $C' \cdot n^2 \leq T(n) \leq C \cdot n^2$

  - Um konkreter zu verstehen, was so ein Laufzeitverhalten bedeutet, nehmen wir der Einfachheit halber an

    $T(n) = C \cdot n^2$ für eine Konstante C

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

    Doppelt so große Eingabe → viermal so große Laufzeit

Laufzeitanalyse MinSort 5/5

- Quadratische Laufzeit, Rechenbeispiel
  - Unabhängig von den Konstanten wird das schnell **sehr** teuer
  - Annahme: $C = 10^{-9}$ (1 einfache Anweisung ≈ 1 Nanosekunde)
  - Beispiel 1: $n = 10^6$ (1 Millionen Zahlen = 8 MB, 8 Bytes / Zahl)

    … dann $C \cdot n^2 = 10^{-9} \cdot 10^{12}$ s = 10^3 s = 16,7 min

    × 1 M

    × 1000
  - Beispiel 2: $n = 10^9$ (1 Milliarde Zahlen = 8 GB)

    … dann $C \cdot n^2 = 10^{-9} \cdot 10^{18}$ s = 10^3 s = 31,7 Jahre

Quadratische Laufzeit = "große" Probleme unlösbar

Wenn Sie nur eine Sache aus dieser ganzen Vorlesung lernen,
dann bitte, wie man **quadratische Laufzeit** erkennt,
und dass sie **in der Praxis absolut inakzeptabel** ist

Laufzeitanalyse MergeSort 1/5

- Laufzeitanalyse Mischen
  - Die Anzahl der Operation für das **Mischen** von zwei Feldern der Gesamtgröße m ist $A \cdot m$, für eine Konstante A

    Beweis:

    0. Seien m_1 und m_2 die Größen der beiden Felder und $m = m_1 + m_2$ die Gesamtgröße
    1. Zu Beginn sind i und j beide 0, also $i + j = 0$
    2. Am Ende ist $i = m_1$ und $j = m_2$, also $i + j = m$
    3. In jedem Schleifendurchlauf wird entweder i oder j um 1 erhöht, also immer $i + j$ um genau 1 erhöht
    4. Es gibt also genau m Schleifendurchläufe und die Anzahl der Operationen ist $\leq A_1 \cdot m + A_2 \leq A \cdot m$

$= (A_1 + A_2) \cdot m$ $\quad A := A_1 + A_2$

Laufzeitanalyse MergeSort 2/5

- Laufzeitanalyse der Iterationen von MergeSort
  - Der Einfachheit halber nehmen wir an, dass n eine Zweierpotenz ist

    Iteration 1: n Teilfelder der Größe jeweils $\mathbf{1} = 2^0$

    $n / 2$ mal Mischen à $A \cdot 2$ ⇨ Zeit $\leq n / 2 \cdot A \cdot 2 = \mathbf{A \cdot n}$

    Iteration 2: $n/2$ Teilfelder der Größe jeweils $\mathbf{2} = 2^1$

    $n / 4$ mal Mischen à $A \cdot 4$ ⇨ Zeit $\leq n / 4 \cdot A \cdot 4 = \mathbf{A \cdot n}$

    …

    Iteration k: 2 Teilfelder der Größe jeweils $\mathbf{n / 2} = 2^{k-1}$

    1 mal Mischen à $A \cdot n$ ⇨ Zeit $\leq 1 \cdot A \cdot n = \mathbf{A \cdot n}$

Laufzeitanalyse MergeSort 3/5

- Analyse iterativer MergeSort … Fortsetzung
  - Sei **k** die Anzahl der Iterationen
  - Dann ist die Laufzeit insgesamt **$T(n) \leq A \cdot n \cdot k$**
  - Wie groß ist **k** ?
  - Die beiden Teilfelder der letzten Iteration k haben die Größe jeweils $2^{k-1} = n / 2$ (siehe vorherige Folie), also gilt:

    $k - 1 = \log_2 (n/2) = (\log_2 n) - 1 \leq \log_2 n$

    $\Rightarrow \quad k \leq 1 + \log_2 n$

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

    Anmerkung: bei der rekursiven Implementierung kommt auch etwas mit $n \cdot \log_2 n$ heraus

$\log(x \cdot y) = \log x + \log y$
$\log(x / y) = \log x - \log y$
$\log_2(n/2) = \log_2 n - \log_2 2$ ($= 1$)

"1+" ist egal, weil $\log_2 1 = 0$

Laufzeitanalyse MergeSort 4/5

- Laufzeit $n \cdot \log n$
  - Es gibt Konstanten C' und C, so dass gilt

    $C' \cdot n \cdot \log_2 n \leq T(n) \leq C \cdot n \cdot \log_2 n$ für $n \geq 2$

  - Um konkreter zu verstehen, was so ein Laufzeitverhalten bedeutet, nehmen wir der Einfachheit halber an

    $T(n) = C \cdot n \cdot \log_2 n$ für eine Konstante C

    Dann gilt: $T(2n) = C \cdot 2n \cdot \log_2(2n) \approx 2 \cdot C \cdot n \cdot \log_2 n = 2 \cdot T(n)$

    $\log_2(2n) = \log_2 n + 1 \approx \log_2 n$ für große n

    Doppelt so große Eingabe → etwas mehr als doppelt so lange

Laufzeitanalyse MergeSort 5/5

- Laufzeit $n \cdot \log n$, Rechenbeispiel
  - Annahme: $C = 10^{-9}$ (1 einfache Anweisung ≈ 1 Nanosekunde)
  - Beispiel 1: $n = 2^{20}$ ($\approx 10^6$ = 1 Millionen Zahlen = 8 MB)

    … dann $C \cdot n \cdot \log_2 n = 10^{-9} \cdot 2^{20} \cdot 20\ s =$ 21 ms
  - Beispiel 2: $n = 2^{30}$ ($\approx 10^9$ = 1 Milliarde Zahlen = 8 GB)

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

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

$2^{16} = 65536$

$\approx 10^6$

$\approx 10^9$

× 1000

× 1500

Laufzeit n · log n ist also FAST SO GUT wie linear!

Noch ein Hinweis zum Ü2

- Zum Verständnis des Logarithmus
  - Nehmen wir an, n ist eine Zweierpotenz und $k = \log_2 n$
  - Dann ist k genau, wie oft man 2 mit sich selber mal nehmen muss, bis man n erreicht

    $n = 32$ $\quad 1 \xrightarrow{1.} 2 \xrightarrow{2.} 4 \xrightarrow{3.} 8 \xrightarrow{4.} 16 \xrightarrow{5.} 32$ $\quad 32 = 2^5$ $\quad \log_2 32 = 5$
  - Umgekehrt, ist k genau, wie oft man n durch 2 teilen muss, bis man bei 1 ankommt

    $n = 32$ $\quad 32 \xrightarrow{1.} 16 \xrightarrow{2.} 8 \xrightarrow{3.} 4 \xrightarrow{4.} 2 \xrightarrow{5.} 1$

    Diese Intuition ist in der Informatik sehr wichtig, um zu verstehen, warum dort so oft der Logarithmus auftaucht

    Das ist auch für die Aufgabe 2 und 3 vom Ü2 hilfreich

Andere Sortierverfahren 1/9

- QuickSort, Grundprinzip
  - Ähnlich wie beim 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 2/9

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

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

    Aber in der Praxis schneller als MergeSort, weil das Mischen wegfällt und keine Felder kopiert werden müssen

    Man sagt, QuickSort sortiert die Eingabe **in-place**
  - Im schlechtesten Fall wird das Feld pro Rekursionsstufe immer nur eins kleiner und die Laufzeit ist $\geq C' \cdot n^2$
  - Wenn das sogenannte "Pivot-Element" (nach dem die Felder aufgeteilt werden) immer zufällig gewählt wird, ist der **Erwartungswert** der Laufzeit $\leq C'' \cdot n \cdot \log_2 n$

Andere Sortierverfahren 3/9

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

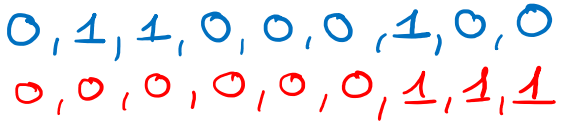
    Das ist eine Datenstruktur, die aus einer Menge von n Elementen mit $\leq C \cdot \log n$ Operationen das kleinste Element extrahieren und entfernen kann

    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 ... weil kleineres C
    HeapSort etwas schlechter als QuickSort ... weil größeres C

Andere Sortierverfahren 4/9



- ZeroOneSort und CountingSort
  - Geht Sortieren auch besser als $C \cdot n \cdot \log n$?
  - Ja, zum Beispiel wenn alle Elemente nur 0 oder 1 sind

    1. Zähle die Anzahl der 0en und 1en → n_0 und n_1
    2. Gebe n_0 mal die 0 gefolgt von n_1 mal die 1 aus

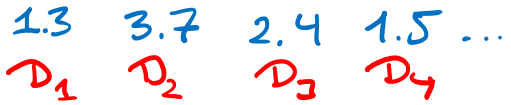
    Die Laufzeit ist $\leq C \cdot n$, für eine Konstante C
  - Das lässt sich verallgemeinern für Elemente aus $0 .. m-1$ indem man zum Zählen ein Feld der Größe m verwendet

    Das zugehörige Sortierverfahren heißt **CountingSort**

    Es läuft in $\leq C \cdot (n + m)$ Operationen, was für $m \leq n$ auch besser als $C \cdot n \cdot \log n$ ist

Andere Sortierverfahren 5/9

- Sortieren von (komplexen) Objekten



  - 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 Verweis** auf die anderen Daten

    Verweis = die Adresse unter der die anderen Daten stehen

    Dann müssen bei jedem Vertauschen nur die beiden Werte und die beiden Verweise ("Zeiger") kopiert werden

Andere Sortierverfahren 6/9

- SleepSort
  - Für Eingabezahl x, warte ~ x Sekunden, dann x ausgeben

```
def sleep_sort(numbers: list[int]):
    for x in numbers:
        threading.Thread(target=lambda x=x:
                (time.sleep(0.001 * x), print(x))).start()
```

Genial, oder?

Andere Sortierverfahren 7/9

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

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

    Das geht ebenfalls in Zeit $\leq C' \cdot n$, für eine Konstante C'
  - Die erwartete Laufzeit ist $\geq \mathbf{C'' \cdot n \cdot n!}$

    Die große Frage ist: geht es noch **langsamer**?

Andere Sortierverfahren 8/9

- 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

  - **BogoBogoSort** ist korrekt, aber beeindruckend langsam: schon auf Eingaben der Größe 7 wird es nicht fertig

Andere Sortierverfahren 9/9

- VerschwörungsSort
  - Angenommen, die Eingabezahlen sind alle verschieden, dann ist die Wahrscheinlichkeit, dass die Zahlen genau so permutiert sind wie sie in der Eingabe vorliegen $1/n!$

    Das ist **extrem** klein: z.B. für $n = 100$ ist $1/n! \approx 10^{-158}$
  - Es ist absurd zu denken, dass die Eingabe zufällig in genau dieser Permutation vorliegt
  - Man kann deswegen annehmen, dass die Eingabe schon optimal sortiert ist … in einer Reihenfolge, die unser weltliches Verständnis von "sortiert" transzendiert

    Laufzeit 0, Denkarbeit 0, praktischer Nutzen 0

Fazit Sortieren

- Fazit
  - Unter den vergleichsbasierten Algorithmen sind **MergeSort**, **QuickSort** und **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

- Mehlhorn / Sanders
  - Laufzeit allgemein 2.6 Basic Program Analysis
- Wikipedia
  - Artikel zu allen gängigen Sortierverfahren

    Die englischen Artikel sind fast immer (viel) besser
- YouTube
  - Visualisierung und Tonalisierung von **15** Sortieralgorithmen, von Timo Bingmann (MinSort heißt dort SelectionSort)

    https://www.youtube.com/watch?v=kPRA0W1kECg