Algorithmen und Datenstrukturen

Sortierung durch Vergleichen, Asymptotische Komplexität von Algorithmen
Entwurfsprinzip Teile und Herrsche

Prof. Dr. Ralf Möller

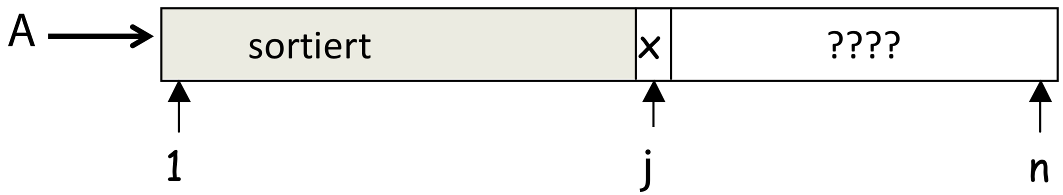
Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Wiederholung: Insertion-Sort (aufsteigend)




```
procedure INSERTION-SORT(A)
    for j ← 2 to length(A) do
        key ← A[j]
        ▷ Insert A[j] into A[1..j − 1]
        i ← j − 1
        while i > 0 and A[i] > key do
            A[i + 1] ← A[i]
            i ← i − 1
        A[i + 1] ← key
```

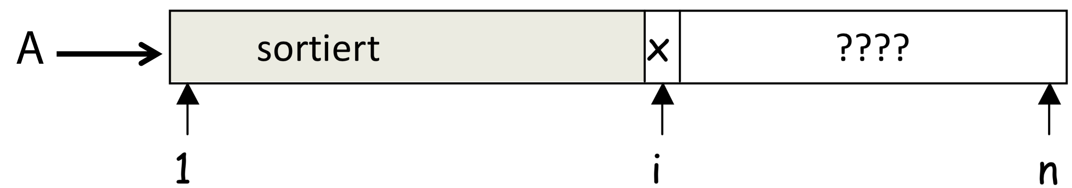
Schlechtester Fall: A ist absteigend sortiert:

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\frac{n(n-1)}{2}$$

$$T(n) = c_0\, n^2 + \ldots$$

Eine andere "Idee" für Sortieren

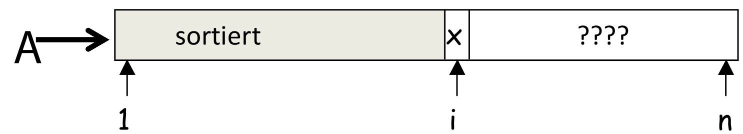
- Gegeben: a = [4, 7, 3, 5, 9, 1]
- Gesucht: In-situ-Sortierverfahren



procedure SELECTION-SORT(A)
    for $i \leftarrow 1$ to $length(A)$ **do**
        $min \leftarrow i$
        for $j \leftarrow i + 1$ to $length(A)$ **do**
            $\triangleright$ find the minimum in the unsorted part
            if $A[j] < A[min]$ **then**
                $min \leftarrow j$
        $x \leftarrow A[i]; A[i] \leftarrow A[min]; A[min] \leftarrow x$
        $\triangleright$ swap the found minimum into the sorted part

Sortieren durch Auswählen (Selection-Sort)

- Gleiches **Entwurfsmuster: Verkleinerungsprinzip**
- Aufwand im **schlechtesten Fall**?
  - $T(n) = c_1 n^2 + \ldots$
- Aufwand im **besten Fall**?
  - $T(n) = c_2 n^2 + \ldots$



- Sortieren durch Auswählen scheint also noch schlechter zu sein als Sortieren durch Einfügen !
- Kann sich jemand eine Situation vorstellen, in der man trotzdem zu Sortieren durch Auswählen greift?
  - Was passiert, wenn die Elemente sehr groß sind?
  - Verschiebungen sind aufwendig

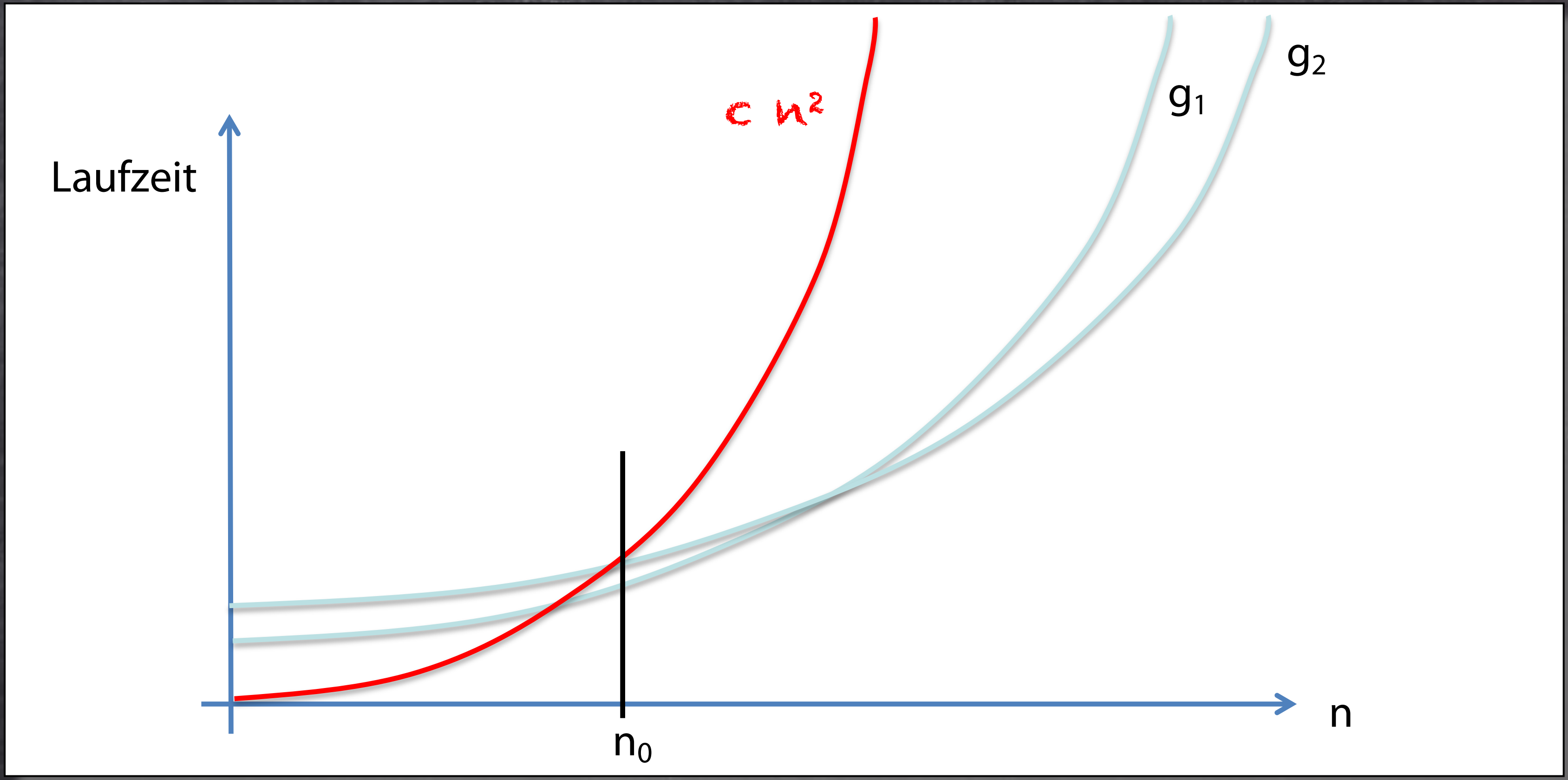
Vergleich der beiden Algorithmen

- Anzahl Vergleiche?
- Anzahl Zuweisungen?
- Berechnungen?

- Was passiert, wenn die Eingabe immer weiter wächst?
  - $n \longmapsto \infty$
- Asymptotische Komplexität eines Algorithmus
- Die Konstanten c_i sind nicht dominierend
- Lohnt es sich, die Konstanten zu bestimmen?
- Sind die beiden Algorithmen substantiell verschieden?

Quadratischer Aufwand

- Algorithmus 1: $g_1(n) = b_1 + c_1 * n^2$
- Algorithmus 2: $g_2(n) = b_2 + c_2 * n^2$



Oberer „Deckel“: O-Notation

- Charakterisierung von Algorithmen durch Menge von Funktionen

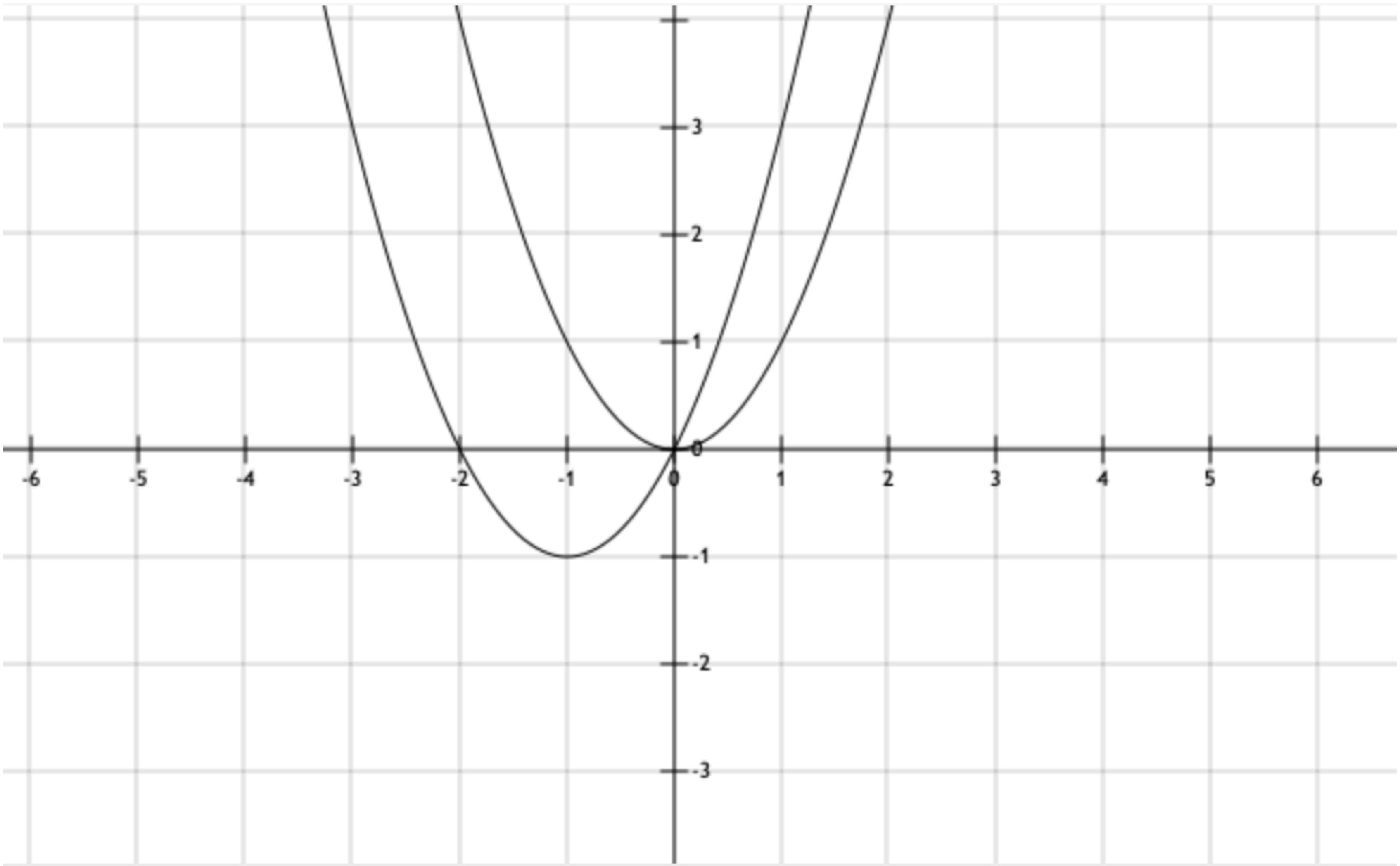
$$O(\,f(n)\,) = \{\, g: \mathbb{N} \to \mathbb{N} \mid \exists\, n_0 > 0 \wedge \exists\, c > 0, \text{ so dass } \forall\, n \geq n_0: \; g(n) \leq c * f(n) \,\}$$

- Sei $f(n) = n^2$
- $g_1 \in O(n^2)$
- $g_2 \in O(n^2)$
- g_1 und g_2 sind „von der gleichen Art“
- O(f(n)) definiert „Klasse“ von Funktionen

Erstmals vom deutschen Zahlentheoretiker Paul Bachmann in der **1894** erschienenen zweiten Auflage seines Buchs Analytische Zahlentheorie verwendet. Verwendet auch vom deutschen Zahlentheoretiker Edmund Landauer, daher auch Landau-Notation genannt (**1909**) [Wikipedia 2015]

In der Informatik populär gemacht durch Donald Knuth, In: The Art of Computer Programming: Fundamental Algorithms, Addison-Wesley, **1968**

O-Notation: $O(n^2) = O(n^2 + 2n)$

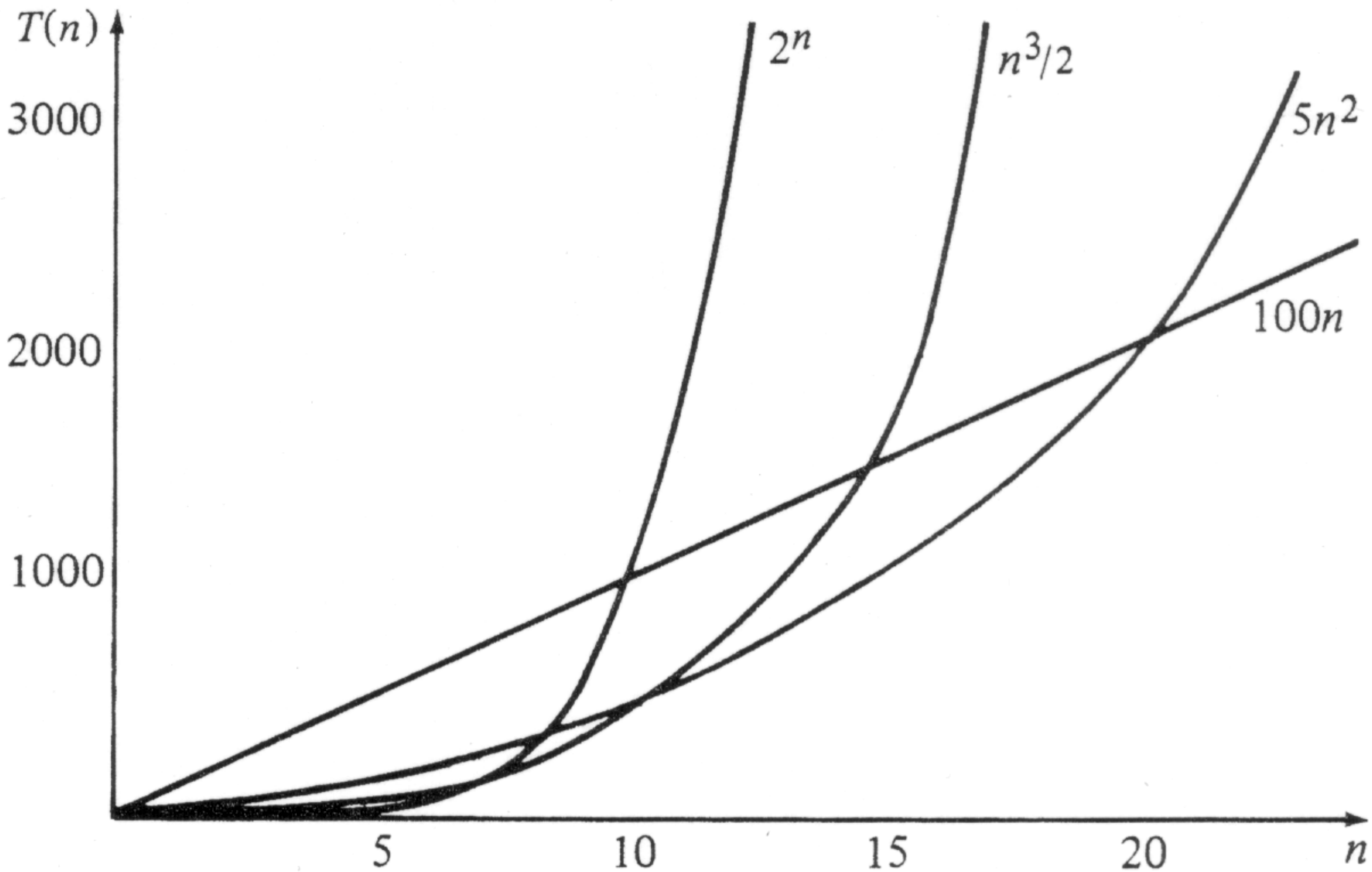


O-Notation

$$O(\, f(n)\,) = \{\, g: \mathbb{N} \to \mathbb{N} \mid \exists\, n_0 > 0 \wedge \exists\, c > 0, \text{ so dass } \forall\, n \geq n_0: \; g(n) \leq c * f(n) \,\}$$

- Quadratischer Aufwand: $O(n^2)$
- Linearer Aufwand: $O(n)$
- Konstanter Aufwand: $O(1)$

Laufzeiten

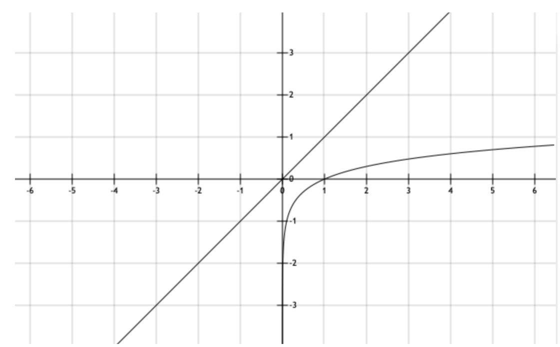


Verfeinerung: Lernziele

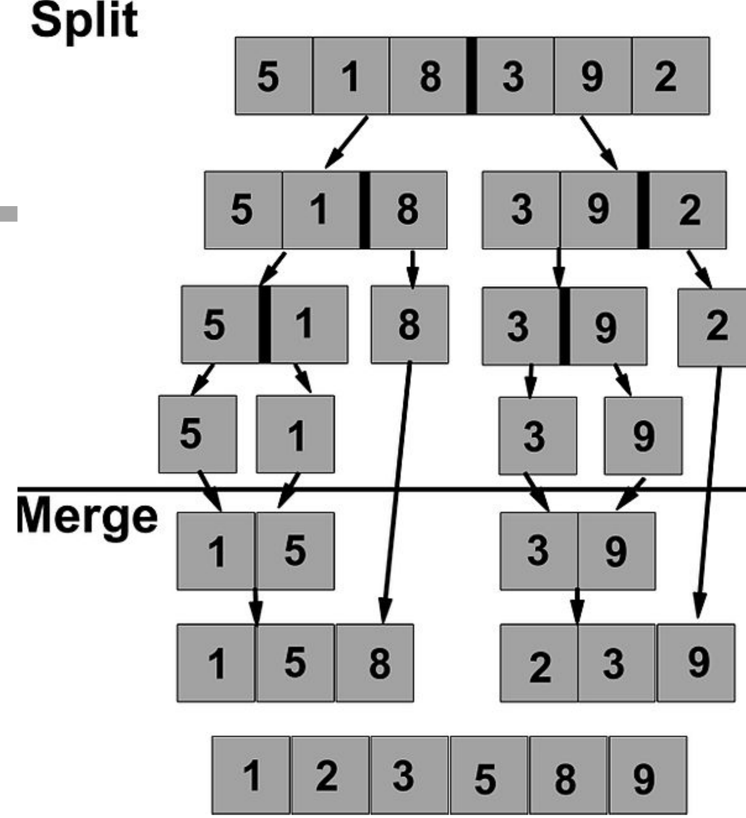
- Entwickeln einer **Idee** zur Lösung eines Problems
- **Notieren** der Idee
  - Zur Kommunikation mit Menschen (Algorithmus)
  - Zur Ausführung auf einem Rechner (Programm)
- Analyse eines Algorithmus in Hinblick auf den **Aufwand**
  - O-Notation
  - Algorithmus zur Lösung eines Problems liefert **Obergrenze** für Komplexität eines Problems
- Ist der Algorithmus **optimal**?
  - **Asymptotische Komplexität** eines optimalen Algorithmus ist **gleich** der **Komplexität des Problems**

Das In-situ-Sortierproblem

- Betrachtete Algorithmen sind quadratisch, d.h. sie haben eine Zeitfunktion in $O(n^2)$
- In-situ-Sortierproblem ist nicht schwieriger als quadratisch
  - n^2 ist ein Polynom, daher sagen wir,
  - das Problem ist polynomiell lösbar (vielleicht aber einfacher)
- In-situ-Sortierproblem im typischen Fall schneller lösbar?
- In-situ-Sortierprobleme brauchen im allgemeinen Fall mindestens n Schritte (jedes Element falsch positioniert)
  - Ein vorgeschlagener Algorithmus, der eine konstante Anzahl von Schritten als asymptotische Komplexität hat, kann nicht korrekt sein
  - Was ist mit logarithmisch vielen Schritten?



Idee: Teile und Herrsche



procedure $\textsc{Merge-Sort}(A, p, r)$
  $\triangleright$ Sortiere $A[p..r]$
  if $p < r$ **then**
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    $\textsc{Merge-Sort}(A, p, q)$
    $\textsc{Merge-Sort}(A, q + 1, r)$
    $\textsc{Merge}(A, p, q, r)$

procedure $\textsc{Merge}(A, p, q, r)$
  $\triangleright$ Sortiere $A[p..r]$, nehme an, dass $A[p..q]$ und $A[q + 1..r]$ sortiert
  $i \leftarrow p$; $j \leftarrow q + 1$
  for $k \leftarrow 1$ **to** $r - p + 1$ **do**
    if $j > r$ or $(i \leq q$ and $A[i] \leq A[j])$ **then**
      $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$ **else** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$
  for $k \leftarrow 1$ **to** $r - p + 1$ **do** $A[k + p - 1] \leftarrow B[k]$

Entwickelt von John von Neumann, **1945**, nach Donald E. Knuth: *The Art of Computer Programming. Volume 3: Sorting and Searching.*. 2 Auflage. Addison-Wesley, 1998, S. 158.

Analyse der Merge-Sort-Idee

- Was haben wir aufgegeben?
- Speicherverbrauch nicht konstant, sondern von der Anzahl der Elemente von A abhängig:
  - Hilfsfeld B (zwar temporär aber gleiche Länge wie A!)
  - Logarithmisch viele Hilfsvariablen
  - Wir können vereinbaren, dass Letzteres für In-situ-Sortieren noch OK ist
  - Es ist aber kaum OK, eine „Kopie" B von A anzulegen
- Merge-Sort löst also nicht (ganz) das gleiche Problem wie Insertion-Sort (oder Selection-Sort)
- Problem mit dem Mischspeicher B lässt sich lösen

Analyse von Merge-Sort

Sei $T(n)$ die Laufzeit von MERGE-SORT.

Das Aufteilen braucht $O(1)$ Schritte.

Die rekursiven Aufrufe brauchen $2T(n/2)$ Schritte.

Das Mischen braucht $O(n)$ Schritte.

Also:

$T(n) = c + 2T(n/2) + c'n$, *wobei die Konstanten für die Ordnung O irrelevant sind*

$T(n) \approx 2T(n/2) + n$

Iterative Expansion

MERGE-SORT(A, 1, n)

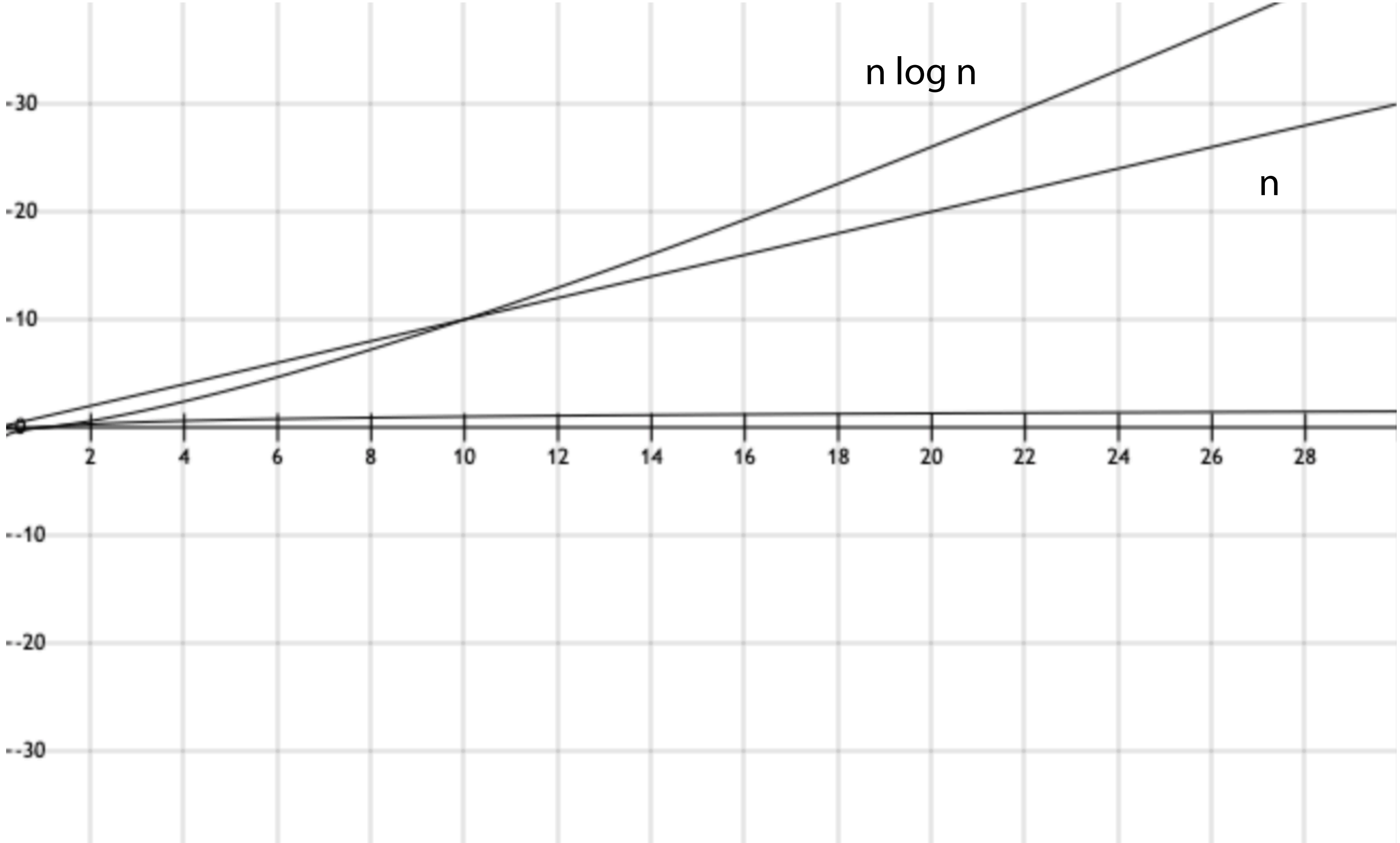
Annahme: $n = 2^k$ (also $k = \log n$).

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(\,2T(n/2^2) + n/2\,) + n \\
&= \mathit{2^2T(n/2^2) + 2n} \\
&= 2^2\,(\,2T(n/2^3) + n/2^2\,) + 2n \\
&= \mathit{2^3T(n/2^3) + 3n} \\
&= \ldots \\
&= \mathit{2^kT(n/2^k) + kn} \\
&= nT(1) + n\log n
\end{aligned}
$$

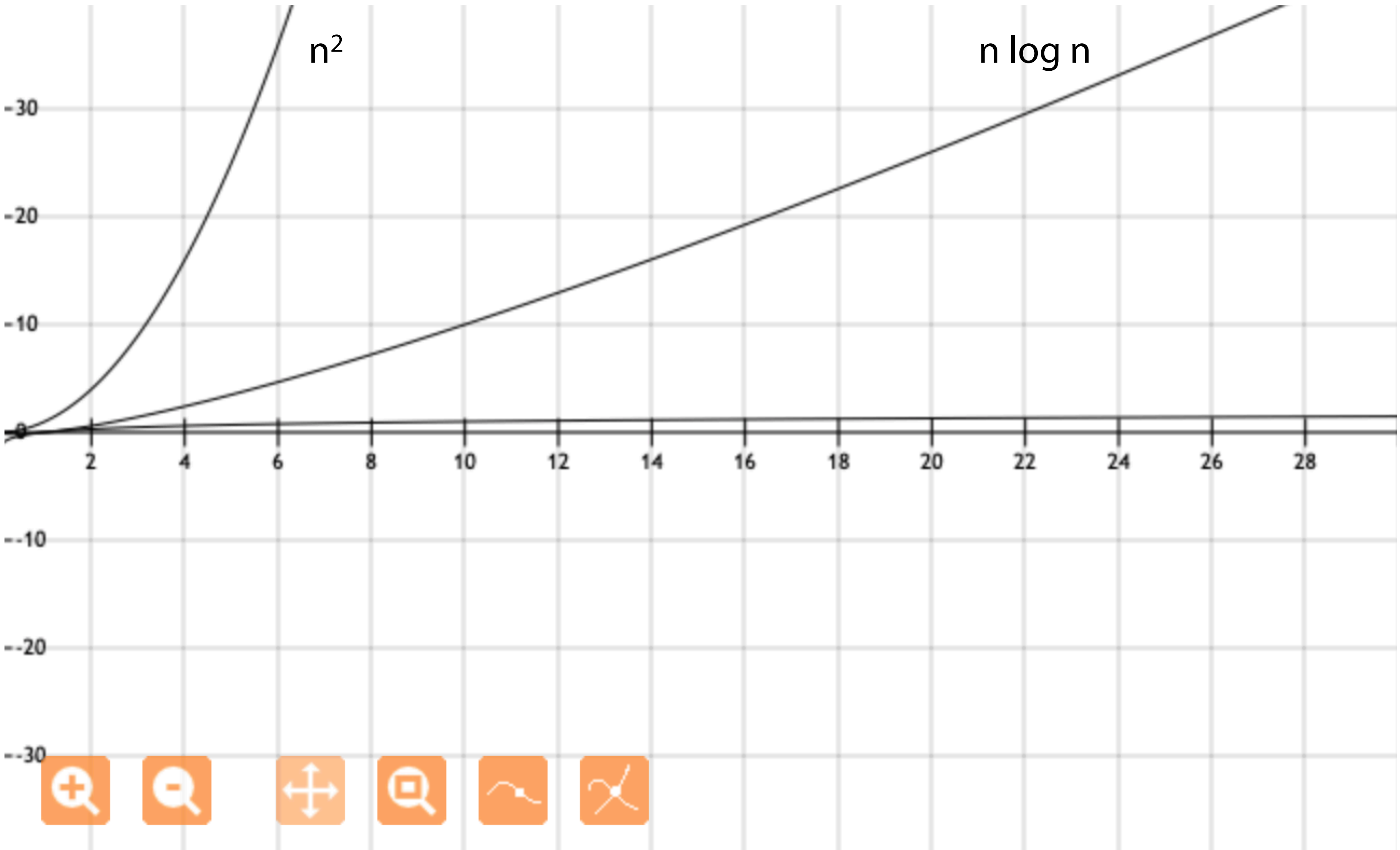
$$T(n/2) = 2T(n/2^2) + n/2$$

$$T(n/2^2) = 2T(n/2^3) + n/2^2$$

Verständnis für n log n erwerben



n^2 vs. n log n



Zusammenfassung

- Problemspezifikation
  - Beispiel: Sortieren mit Vergleichen
- Algorithmenanalyse:
  - Asymptotische Komplexität (O-Notation)
  - Bester, typischer und schlimmster Fall
- Problemkomplexität
- Entwurfsmuster für Algorithmen
  - Verkleinerungsprinzip + Invarianten
  - Teile und Herrsche