Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 3 (27.4.2014)

O-Notation, Asymptotische Analyse, Sortieren III

Fabian Kuhn

Algorithmen und Komplexität

UNI FREIBURG

Selection Sort Algorithmus

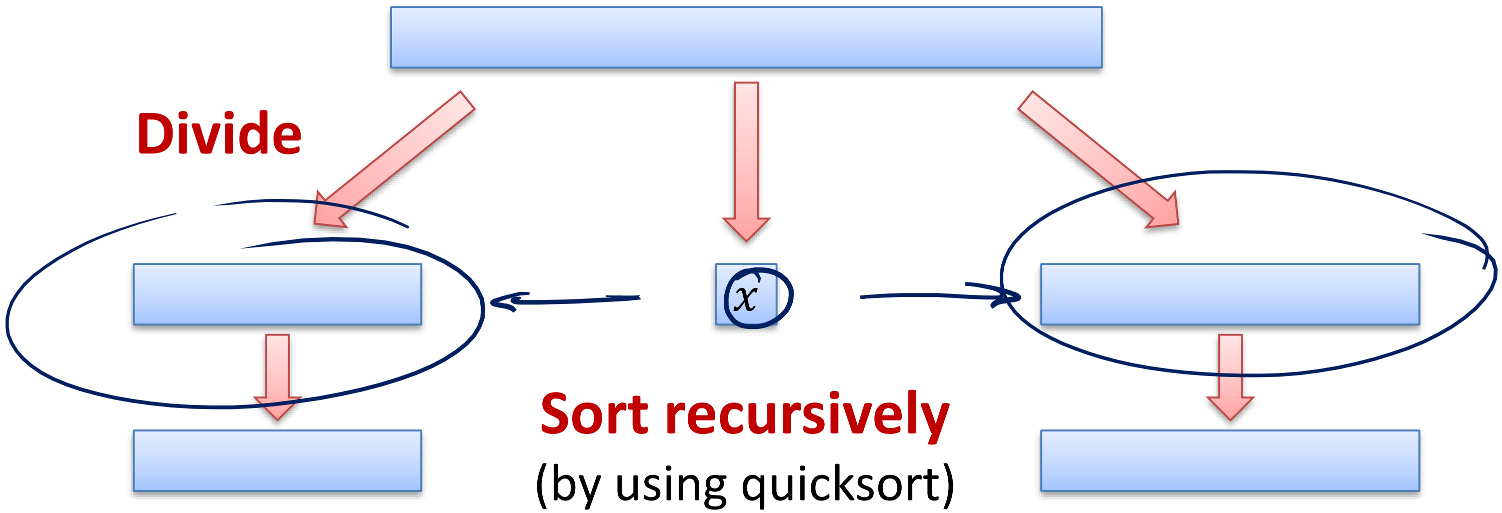
SelectionSort (informell):

1. Finde kleinstes Element im Array, vertausche es an 1. Stelle
2. Finde kleinstes Element im Rest, vertausche es an 2. Stelle
3. Finde kleinstes Element im Rest, vertausche es an 3. Stelle
4. …

```
SelectionSort(A):
for i=0 to n-2 do
  // find min in A[i..n-1]
  minIdx = i
  for j=i to n-1 do
    if A[j] < A[minIdx] then
      minIdx = j
  swap(A[i], A[minIdx])
```

Divide and Conquer

Übersicht QuickSort:

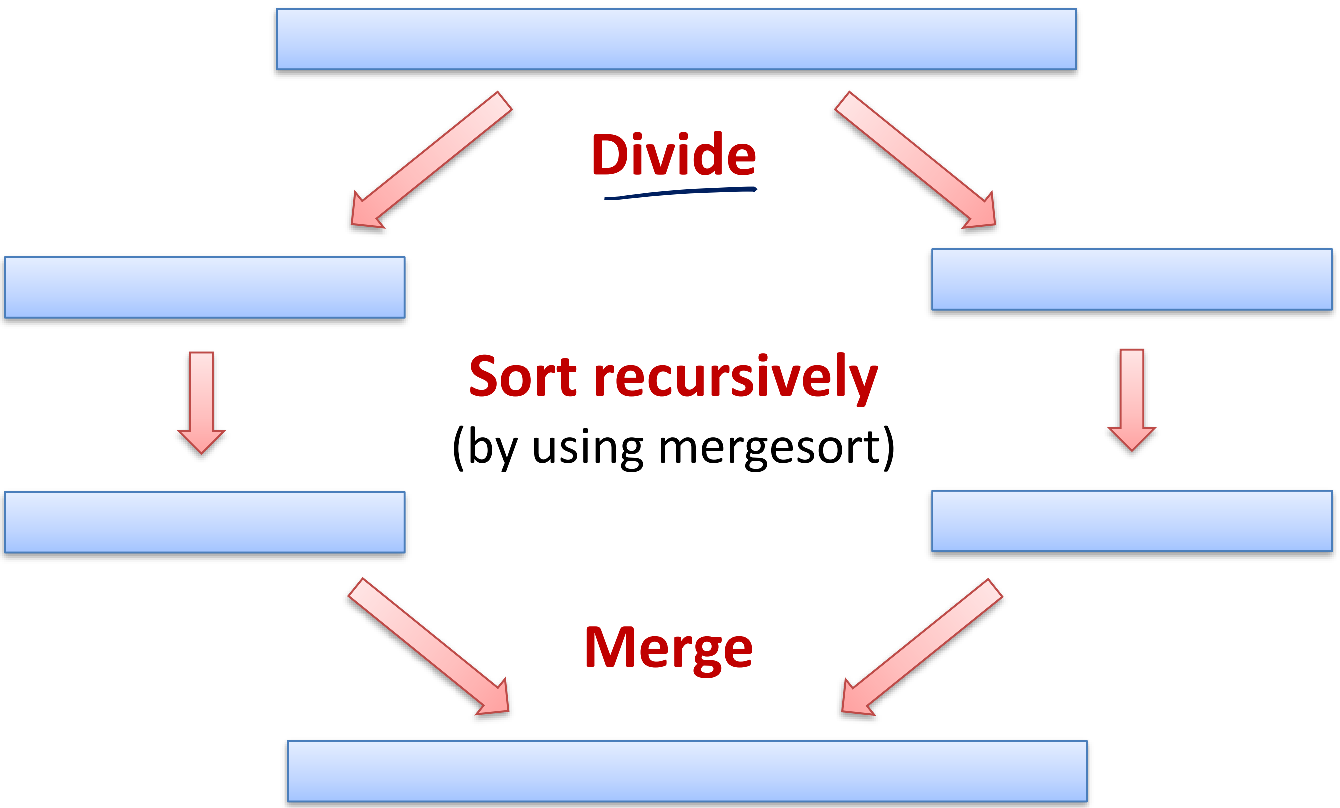


Divide and Conquer:

- Verbreitetes Prinzip für den Algorithmenentwurf

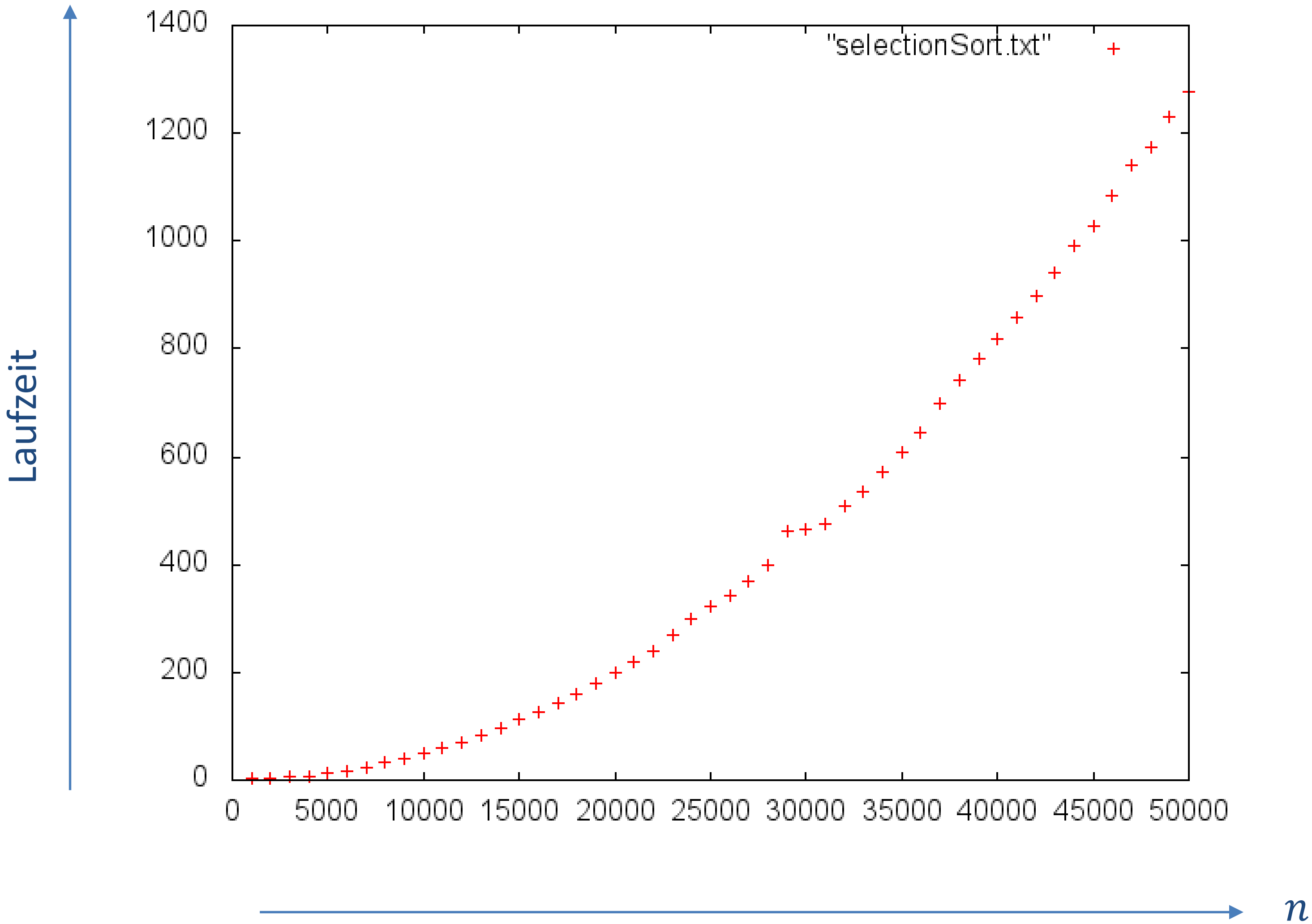
1. Teile Eingabe in 2 oder mehrere kleinere Teilprobleme
2. Löse die Teilprobleme rekursiv
3. Kombiniere die Teillösungen zur Gesamtlösung

Übersicht MergeSort

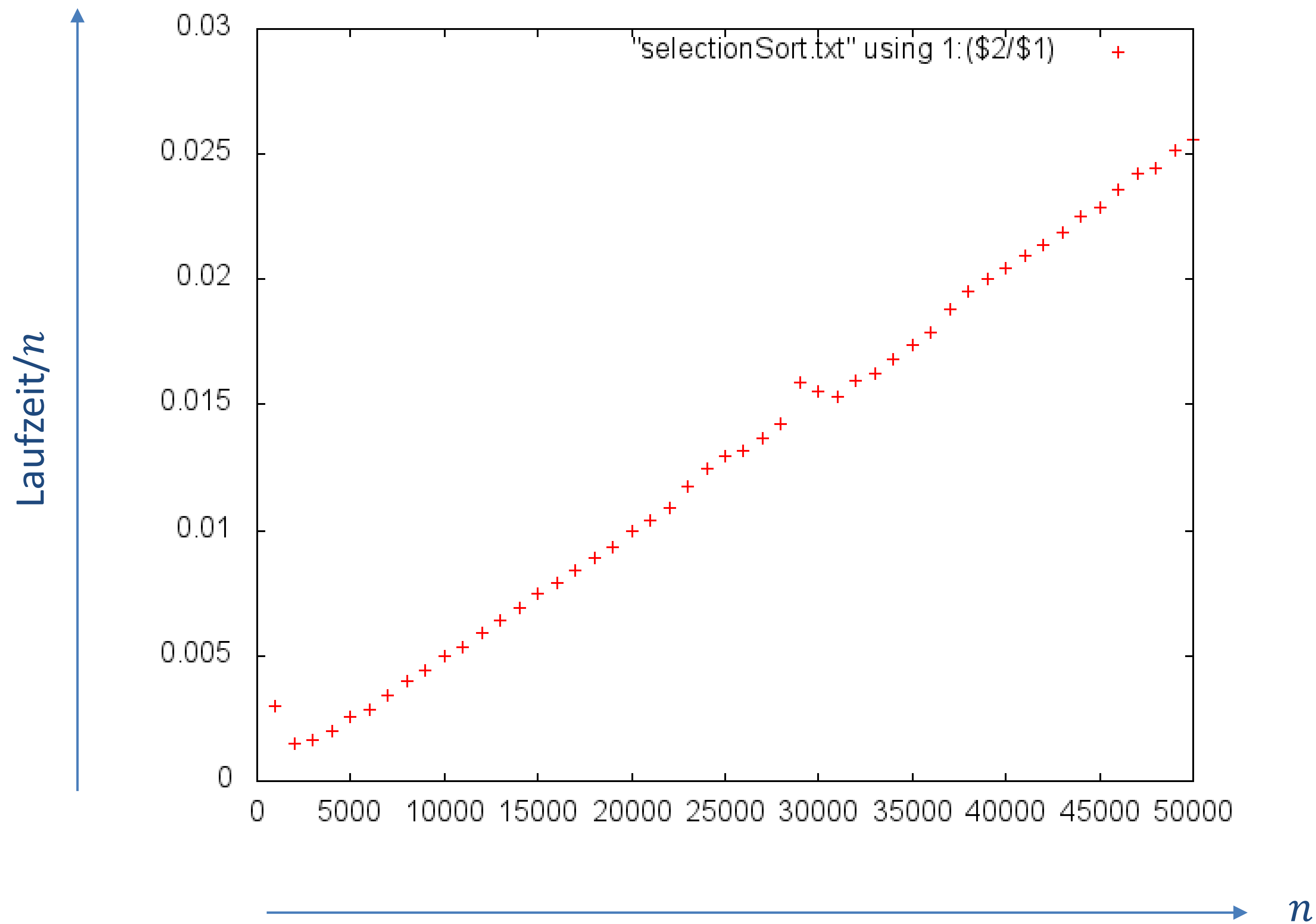


- Divide ist bei MergeSort trivial
- Merge (kombinieren der Lösungen) benötigt dafür Arbeit...

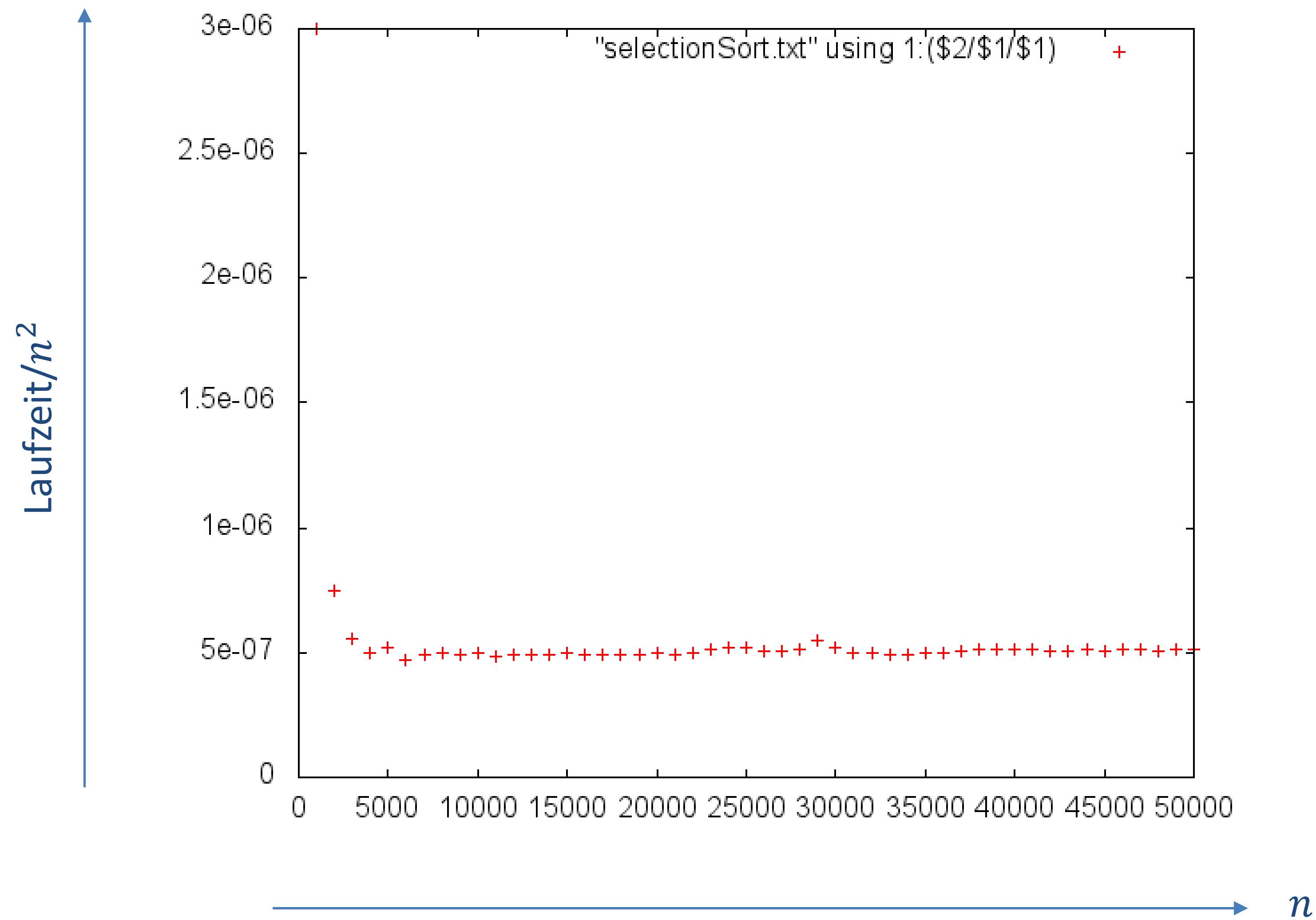
Zeitmessung SelectionSort



Zeitmessung SelectionSort



Zeitmessung SelectionSort



Laufzeitanalyse I

- Wie können wir die Laufzeit des Algorithmus analysieren?
  - Ist auf jedem Computer unterschiedlich…
  - Hängt vom Compiler, Programmiersprache, etc. Ab

- Wir benötigen ein abstraktes Mass, um die Laufzeit zu messen

- **Idee**: Zähle Anzahl (Grund-)Operationen
  - Anstatt direkt die Zeit zu messen
  - Ist unabhängig von Computer, Compiler
  - Ein gutes Mass für die Laufzeit, falls alle Grundoperationen etwa gleich lange brauchen:

Grundoperation

Was ist eine Grundoperation?

- Einfache arithmetische Operationen (auf Integers)
  - +, -, *, /, % (mod), …
- Ein Speicherzugriff
  - Variable auslesen, Variablenzuweisung
  - Ist das wirklich eine Grundoperation?

$x = 7$

$y = 2 \cdot x$

- Ein Funktionsaufruf
  - Natürlich nur das Springen in die Funktion
- **Intuitiv:** eine Zeile Programmcode
- **Besser:** eine Zeile Maschinencode
- **Noch besser (?):** ein Prozessorzyklus

- **Wir werden sehen:** Es ist nur wichtig, dass die Anzahl Grundoperation ungefähr proportional zur Laufzeit ist.

RAM Modell

UNI FREIBURG

RAM = Random Access Machine

- Standardmodell(e), um Algorithmen zu analysieren!
- Grundoperationen (wie "definiert") benötigen alle eine Zeiteinheit
- Insbesondere sind alle Speicherzugriffe gleich teuer:

  Jede Speicherzelle (1 Maschinenwort) kann in 1 Zeiteinheit gelesen, bzw. beschrieben werden
  - ignoriert insbesondere Speicherhierarchien
  - Ist aber in den meisten Fällen eine vernünftige Annahme

- Alternative abstrakte Modelle existieren:
  - um Speicherhierarchien explizit abzubilden
  - bei riesigen Datenmengen (vgl. «Buzzword» Big Data)
    - z.B.: Streaming-Modelle: Speicher muss sequentiell gelesen werden
  - für verteilte/parallele Architekturen
    - Speicherzugriff kann lokal oder über's Netzwerk sein...

Laufzeitanalyse II

Bisher: Anzahl Grundoperationen ist proportional zur Laufzeit

- Das können wir auch erreichen, ohne die Anzahl Grundoperationen genau zu zählen!

Vereinfachung 1: Wir berechnen nur eine obere Schranke (bzw. eine untere Schranke) an die Anzahl Grundoperationen

  – So, dass die obere/untere Schranke immer noch proportional ist…

- Anz. Grundop. kann von div. Eigenschaften der Eingabe abhängen
  – Länge der Eingabe, aber auch z.B. bei Sortieren: zufällig, vorsortiert, …

Vereinfachung 2: Wichtigster Parameter ist Grösse der Eingabe n
Wir betrachten daher die Laufzeit $T(n)$ als Funktion von n.

  – Und ignorieren weitere Eigenschaften der Eingabe

Selection Sort: Analyse

```
SelectionSort(A):
  for i=0 to n-2 do
     minIdx = i
     for j=i to n-1 do (*)
        if A[j] < A[minIdx] then
           minIdx = j
     swap(A[i], A[minIdx])
```

$\leq c_1$

$\leq c_2 \qquad \geq c_2'$

$\leq c_3$

$c > 0$

$$\#op \leq c \cdot \underbrace{\#\,\text{Schleifendurchl. } (*)}_{x(n)}$$

$$T(n) \leq c \cdot x(n) \leq c \cdot n^2$$

$$x(n) = \sum_{i=0}^{n-2} (n-i) = \sum_{j=2}^{n} j \leq \sum_{j=1}^{n} j = \frac{n(n+1)}{2} \leq n^2$$

$$T(n) \geq c' x(n) \geq \frac{c'}{2} n^2$$

Selection Sort: Analyse

$\boldsymbol{T(n)}$: Anzahl Grundop. von Selection Sort bei Arrays der Länge n

Lemma: *Es gibt eine* ***Konstante*** $\boldsymbol{c_U > 0}$, *so dass* $\boldsymbol{T(n) \leq c_U \cdot n^2}$

Lemma: *Es gibt eine* ***Konstante*** $\boldsymbol{c_L > 0}$, *so dass* $\boldsymbol{T(n) \geq c_L \cdot n^2}$

Laufzeitanalyse

Zusammenfassung

- Wir können nur eine Grösse berechnen, welche proportional zur Laufzeit ist
- Wir wollen auch gar nichts anderes berechnen:
  - Analyse sollte unabhängig von Computer / Compiler / etc. sein
  - Wir wollen Aussagen, welche auch in 10/100/... Jahren noch Gültigkeit haben
- Wir werden immer Aussagen der folgenden Art haben:

  Es gibt eine Konstante C, so dass

$$T(n) \leq C \cdot f(n) \quad \text{oder} \quad T(n) \geq C \cdot f(n)$$

- Um dies zu vereinfachen / verallgemeinern gibt's die O-Notation...

Landau-Symbole ("O-Notation")

- Formalismus, um das asymptotische Wachstum von Funktionen zu beschreiben.
  - Formale Definitionen: siehe nächste Folie...

- Es gibt eine Konst. C, so dass $T(n) \leq C \cdot f(n)$ wird zu:

$$T(n) \in O(f(n))$$

- Es gibt eine Konst. C, so dass $T(n) \geq C \cdot g(n)$ wird zu:

$$T(n) \in \Omega(g(n))$$

- Bei Selection Sort: $T(n) \in O(n^2)$, $T(n) \in \Omega(n^2)$

Landau-Symbole : Definitionen

$$\boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n})) \coloneqq \{\boldsymbol{f}(\boldsymbol{n}) \mid \exists \boldsymbol{c}, \boldsymbol{n_0} > \mathbf{0}\ \forall \boldsymbol{n} \geq \boldsymbol{n_0} : \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c} \cdot \boldsymbol{g}(\boldsymbol{n})\}$$

- Funktion $f(n) \in O(g(n))$, falls es Konstanten c und n_0 gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$

$$\boldsymbol{\Omega}(\boldsymbol{g}(\boldsymbol{n})) \coloneqq \{\boldsymbol{f}(\boldsymbol{n}) \mid \exists \boldsymbol{c}, \boldsymbol{n_0} > \mathbf{0}\ \forall \boldsymbol{n} \geq \boldsymbol{n_0} : \boldsymbol{f}(\boldsymbol{n}) \geq \boldsymbol{c} \cdot \boldsymbol{g}(\boldsymbol{n})\}$$

- Funktion $f(n) \in \Omega(g(n))$, falls es Konstanten c und n_0 gibt, so dass $f(n) \geq c \cdot g(n)$ für alle $n \geq n_0$

$$\boldsymbol{\Theta}(\boldsymbol{g}(\boldsymbol{n})) \coloneqq \boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n})) \cap \boldsymbol{\Omega}(\boldsymbol{g}(\boldsymbol{n}))$$

- Funktion $f(n) \in \Theta(g(n))$, falls es Konstanten c_1, c_2 und n_0 gibt, so dass $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ für alle $n \geq n_0$, resp. falls $f(n) \in O(n)$ und $f(n) \in \Omega(n)$

Landau-Symbole : Definitionen

$$\boldsymbol{o\big(g(n)\big) \coloneqq \{f(n) \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}}$$

- Funktion $f(n) \in o(g(n))$, falls für alle Konstanten $c > 0$ gilt, dass $f(n) \leq c \cdot g(n)$ (für genug grosse n, abhängig von c)

$$\boldsymbol{\omega\big(g(n)\big) \coloneqq \{f(n) \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}}$$

- Funktion $f(n) \in \omega(g(n))$, falls für alle Konstanten $c > 0$ gilt, dass $f(n) \geq c \cdot g(n)$ (für genug grosse n, abhängig von c)

Insbesondere gilt:

$$f(n) \in o\big(g(n)\big) \quad \Longrightarrow \quad f(n) \in O\big(g(n)\big)$$

$$f(n) \in \omega\big(g(n)\big) \quad \Longrightarrow \quad f(n) \in \Omega\big(g(n)\big)$$

Landau-Symbole : Intuitiv

$f(n) \in O(g(n))$:

- $f(n)$ " $\leq$ " $g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch nicht schneller als $g(n)$

$f(n) \in \Omega(g(n))$:

- $f(n)$ " $\geq$ " $g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch mindestens so schnell, wie $g(n)$

$f(n) \in \Theta(g(n))$:

- $f(n)$ " $=$ " $g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch gleich schnell, wie $g(n)$

Landau-Symbole : Intuitiv

$f(n) \in o(g(n))$:

- $f(n)$ " $<$ " $g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch langsamer als $g(n)$

$f(n) \in \omega(g(n))$:

- $f(n)$ " $>$ " $g(n)$, asymptotisch gesehen...
- $f(n)$ wächst asymptotisch schneller als $g(n)$

Falls $f(n)$ und $g(n)$ monoton wachsen, gilt:

$$f(n) \in o(g(n)) \quad \Leftrightarrow \quad f(n) \notin \Omega(g(n))$$

$$f(n) \in \omega(g(n)) \quad \Leftrightarrow \quad f(n) \notin O(g(n))$$

Definition über Grenzwerte (vereinfacht)

Folgende Definitionen gelten für monoton wachsende Funktionen

$$f(n) \in O\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) \in \Theta\big(g(n)\big), \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in o\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \in \omega\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Landau-Notation : Bemerkungen

Schreibweise:

- $O(g(n))$, $\Omega(g(n))$, … sind Mengen (von Funktionen)
- Korrekte Schreibweise ist deshalb eigentlich: $f(n) \in O(g(n))$
- Sehr verbreitete Schreibweise: $f(n) = O(g(n))$

Asymptotisches Verhalten für allgemeine Grenzwerte:

- gleiche Schreibweise auch für Verhalten von z.B. $f(x)$ für $x \to 0$
- z.B. Taylor-Reihen: $e^x = 1 + x + O(x^2)$, bzw. $e^x = 1 + x + o(x)$

Alternative Definition für $\boldsymbol{\Omega(g(n))}$:

$$\boldsymbol{\Omega(g(n)) := \{f(n) \mid \exists c, n_0 > 0\ \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}}$$

$$\boldsymbol{\Omega(g(n)) := \{f(n) \mid \exists c > 0\ \forall n_0 > 0\ \exists n \geq n_0 : f(n) \geq c \cdot g(n)\}}$$

- Wir verwenden die 1. Definition
- Macht nur bei nicht-monotonen Funktionen einen Unterschied

$g(n) = n^2$ $\quad f(n) = \begin{cases} n^2 & \text{falls } n \text{ gerade} \\ 1 & \text{fall } n \text{ ungerade} \end{cases}$

Landau-Notation : Beispiele

Selection Sort:

- Laufzeit $T(n)$, es gibt Konstanten $c_1, c_2 : c_1 n^2 \leq T(n) \leq c_2 n^2$

$$T(n) \in O(n^2), \qquad T(n) \in \Omega(n^2), \qquad T(n) \in \Theta(n^2)$$

- $T(n)$ wächst schneller als linear: $T(n) \in \omega(n)$

Weitere Beispiele:

- $f(n) = 10n^3,\ g(n) = n^3/1000$: $f(n) \in \Theta(g(n))$
- $f(n) = e^n,\ g(n) = n^{100}$: $f(n) \in \omega(g(n))$
- $f(n) = n/\log_2 n,\ g(n) = \sqrt{n}$: $f(n) \in \omega(g(n))$
- $f(n) = n^{1/256},\ g(n) = 10 \ln n$: $f(n) \in \omega(g(n))$
- $f(n) = \log_{10} n, g(n) = \log_2 n$: $f(n) \in \Theta(g(n))$
- $f(n) = n^{\sqrt{n}}, g(n) = 2^{\mathrm{n}}$: $f(n) \in o(g(n))$

$\log(n^{\sqrt{n}}) = \sqrt{n} \cdot \log(n)$ $\qquad \log(2^n) = n$

$f(n) = \Theta(\log n)$

$\lim_{n \to \infty} \frac{e^n}{n^{100}} \to \infty$

$\frac{n/\log n}{\sqrt{n}} = \frac{\sqrt{n}}{\log n} = \frac{2^{x/2}}{x}$

$\log_a n = \frac{\ln n}{\ln a}$

Analyse Bubble Sort

```
BubbleSort(A):
  for i=0 to n-2 do          // need to repeat n-1 times
    for j=0 to n-2-i do (*)
      if (A[j] > A[j+1]) then
        swap(A[j], A[j+1])
```

$x(n)$: # Schleifendurchl. (*)

$T(n) \in \Theta(x(n))$

$$x(n) = \sum_{i=0}^{n-2} (n-i) \in \Theta(n^2)$$

Analyse Insertion Sort

```
InsertionSort(A):
  for i = 1 to n-1 do
    // prefix A[1..i] is already sorted
    pos = i
    while (pos > 0) and (A[pos] < A[pos-1]) do
      swap(A[pos], A[pos−1])
      pos = pos - 1
```

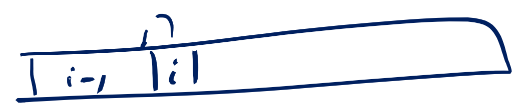
$O(1)$

$(*)$

$X(n) = \ldots$

$x(n) \leq \sum_{i=1}^{n-1} i \in O(n^2)$

$x(n) \geq \sum_{i=1}^{n-1} 1 \geq n-1 \qquad T(n) \in \Omega(n)$



Worst case, best case, average case

Worst Case Analyse

- Analysiere Laufzeit $T(n)$ für eine schlechtestmögliche Eingabe der Grösse n
- Wichtigste / Standard- Art der Algorithmenanlyse

Best Case Analyse

- Analysiere Laufzeit $T(n)$ für eine bestmögl. Eingabe der Grösse n
- Meistens uninteressant…

Average Case Analyse

- Analysiere Laufzeit $T(n)$ für eine typische Eingabe der Grösse n
- Problem: was ist eine typische Eingabe?
  - Standardansatz: zufällige Eingabe
  - nicht klar, wie nahe tatsächliche Instanzen bei uniform zufälligen sind…
  - eine mögl. Alternative: smoothed analysis (werden wir nicht anschauen)

Wie gut ist quadratische Laufzeit?

Quadratisch = 2x so grosse Eingabe → 4x so grosse Laufzeit

- das wächst für grosse n schon ziemlich schnell…

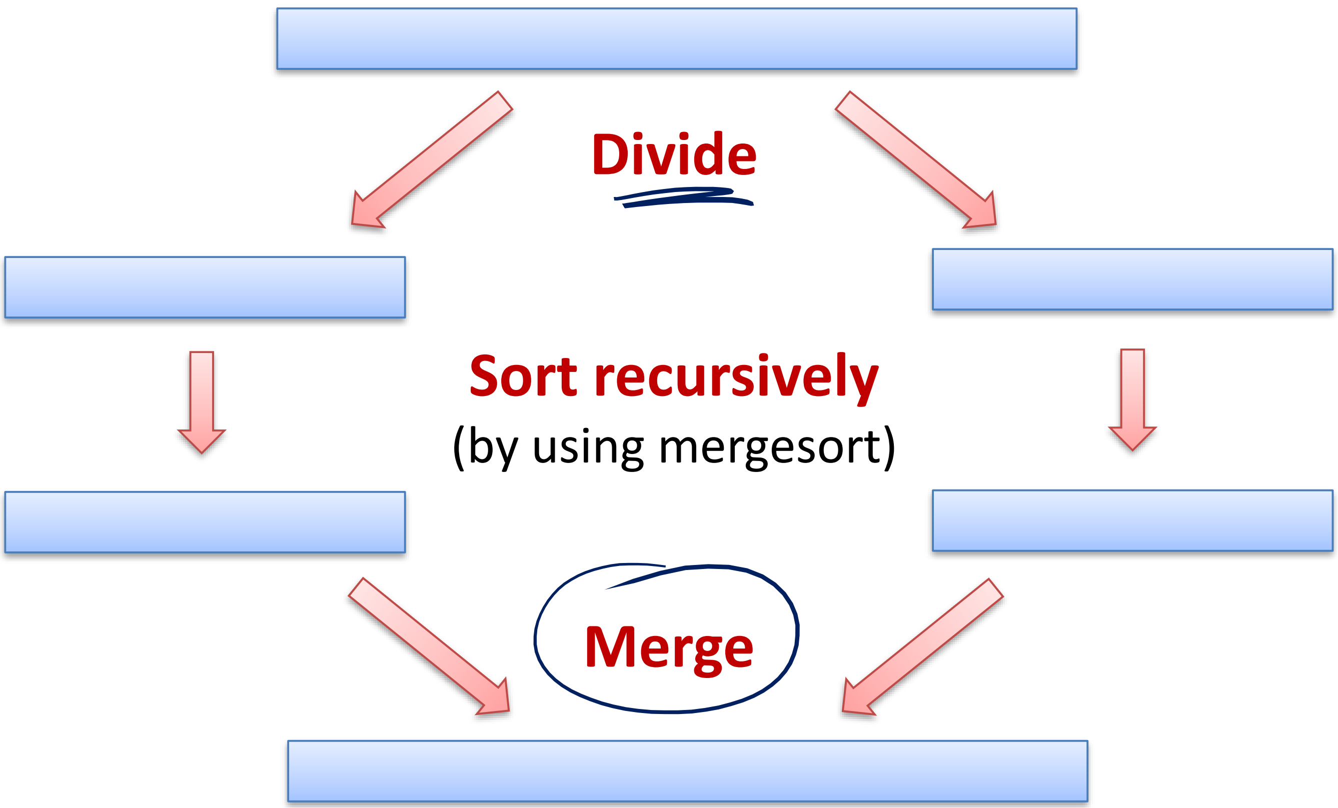
Beispielrechnung:

- Nehmen wir an, Anz. Grundop. $T(n) = n^2$
- Nehmen wir zudem an, 1 Grundop. pro Rechnerzyklus
- Bei einem 1Ghz-Rechner gibt das 1 ns pro Grundop.

Eingabegrösse n	4 Bytes pro Zahl	Laufzeit $T(n)$
10^3 Zahlen	$\approx 4\text{KB}$	$10^{3\cdot 2} \cdot 10^{-9}\ \text{s} = 1\ \text{ms}$
10^6 Zahlen	$\approx 4\text{MB}$	$10^{6\cdot 2} \cdot 10^{-9}\ \text{s} = 16.7\ \text{min}$
10^9 Zahlen	$\approx 4\text{GB}$	$10^{9\cdot 2} \cdot 10^{-9}\ \text{s} = 31.7\ \text{Jahre}$

für grosse Probleme zu langsam!

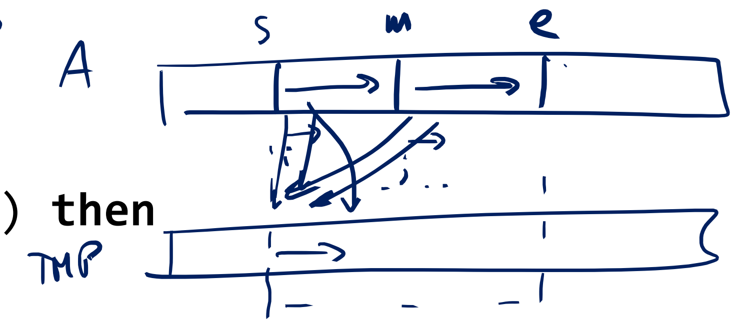
Analyse Merge Sort



- Divide ist trivial → Kosten: $O(1)$
- Rekursives Sortieren: Werden wir gleich noch anschauen...
- Merge: Das werden wir uns zuerst anschauen...

Analyse Merge-Schritt

```
MergeSortRecursive(A, start, end, tmp)                 // sort A[start..end-1]
    ⋮
                     O(1)
    pos = start; i = start; j = middle
  while (pos < end) do
          if (i < middle) and (A[i] < A[j]) then
   O(1)     tmp[pos] = A[i]; pos++; i++
          else
            tmp[pos] = A[j]; pos++; j++
  for i = start to end-1 do A[i] = tmp[i]
                                   O(1)
```



$k := end - start$

#Schleifendurchläufe: $O(end - start) = O(k)$

Merge-Schritt für Teilarray der Länge k : $O(k)$

Analyse Merge Sort

Laufzeit $T(n)$ setzt sich zusammen aus:

- Divide und Merge: $O(n)$
- 2 rekursive Aufrufe zum Sortieren von $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Elementen

Rekursive Formulierung von $T(n)$:

- Es gibt eine Konstante $b > 0$, so dass

$$T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + b \cdot n, \qquad \mathrm{T}(1) \leq b$$

- Wir machen uns das Leben ein bisschen einfacher und ignorieren das Auf- und Abrunden:

$$\boldsymbol{T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \qquad T(1) \leq b}$$