

# Informatik II - SS 2016

## (Algorithmen & Datenstrukturen)

Vorlesung 3 (27.4.2014)

O-Notation, Asymptotische Analyse,  
Sortieren III



**UNI  
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

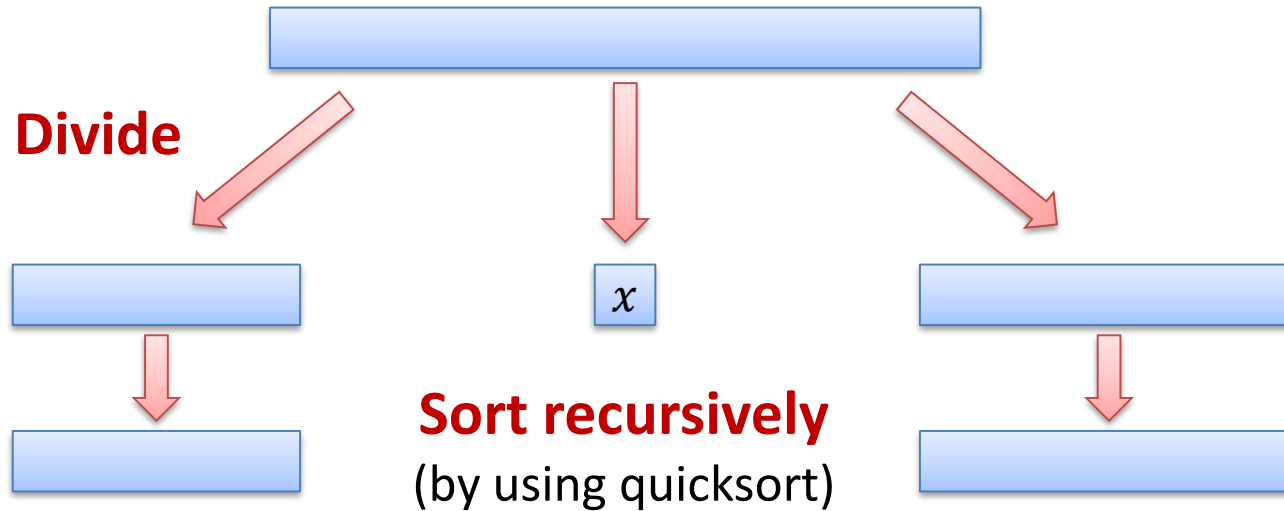
## SelectionSort (informell):

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):

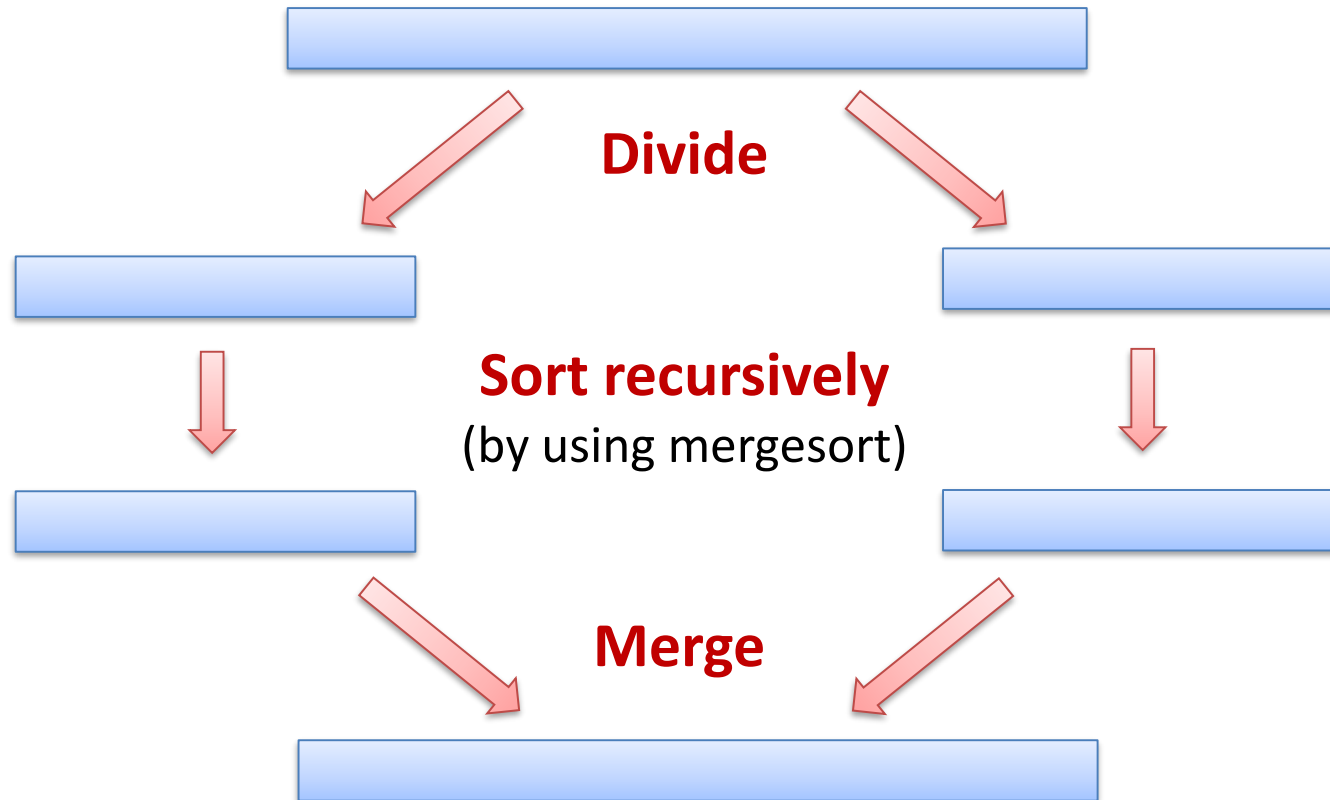
```
1: for i=0 to n-2 do
2:   // find min in A[i..n-1]
3:   minIdx = i
4:   for j=i to n-1 do
5:     if A[j] < A[minIdx] then
6:       minIdx = j
8:   swap(A[i], A[minIdx])
```

## Übersicht QuickSort:



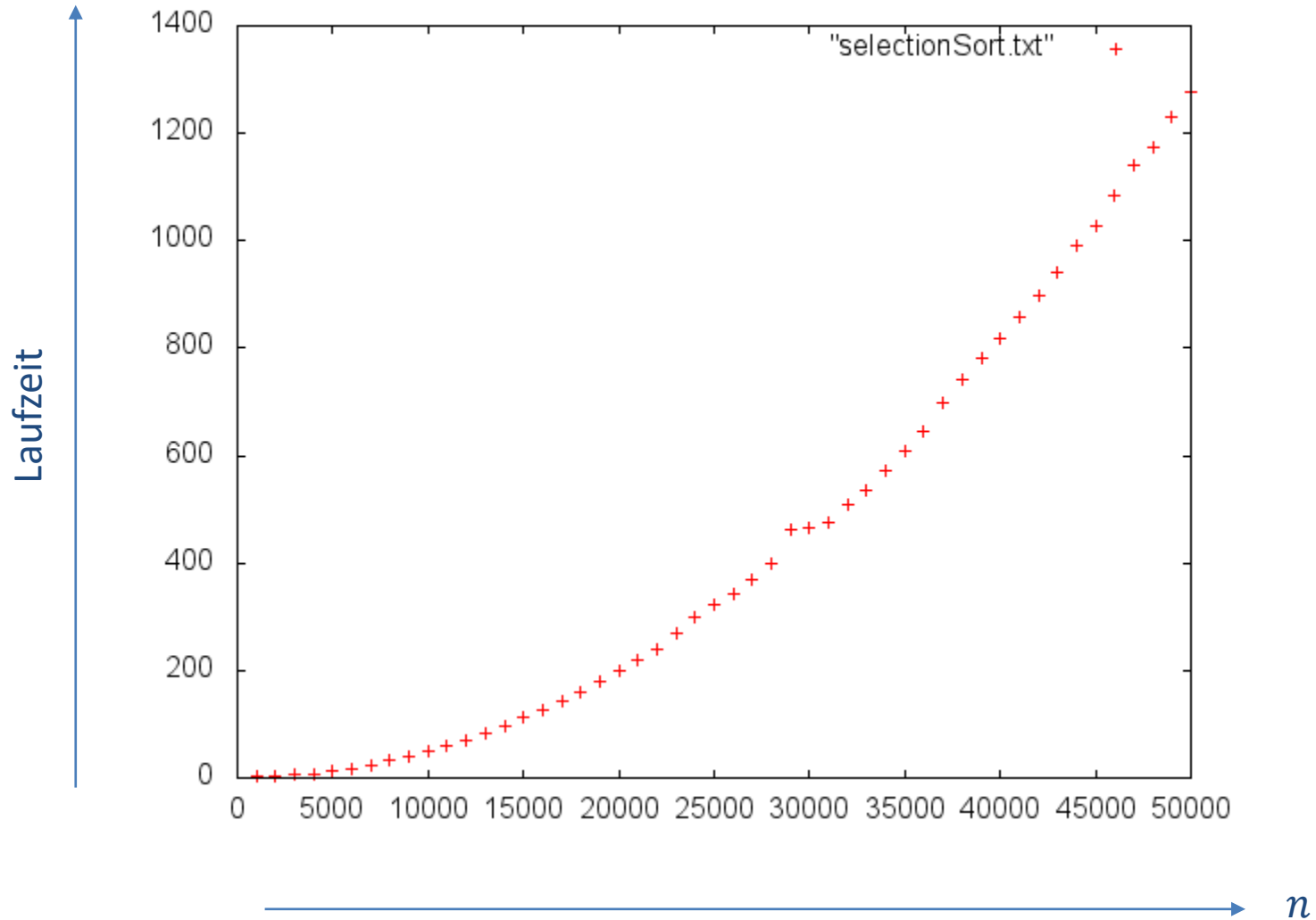
## Divide and Conquer:

- 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

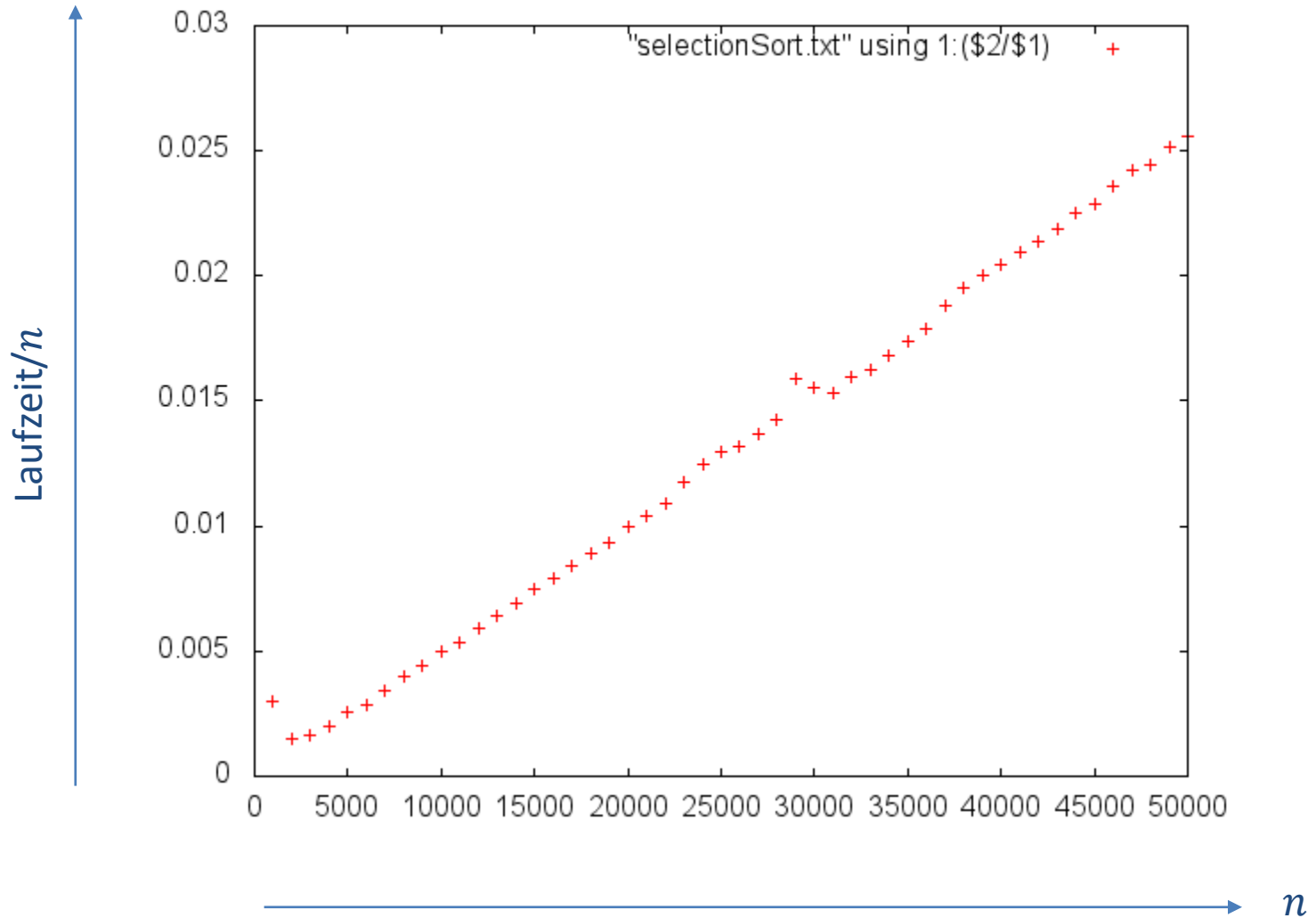


- Divide ist bei MergeSort trivial
- Merge (kombinieren der Lösungen) benötigt dafür Arbeit...

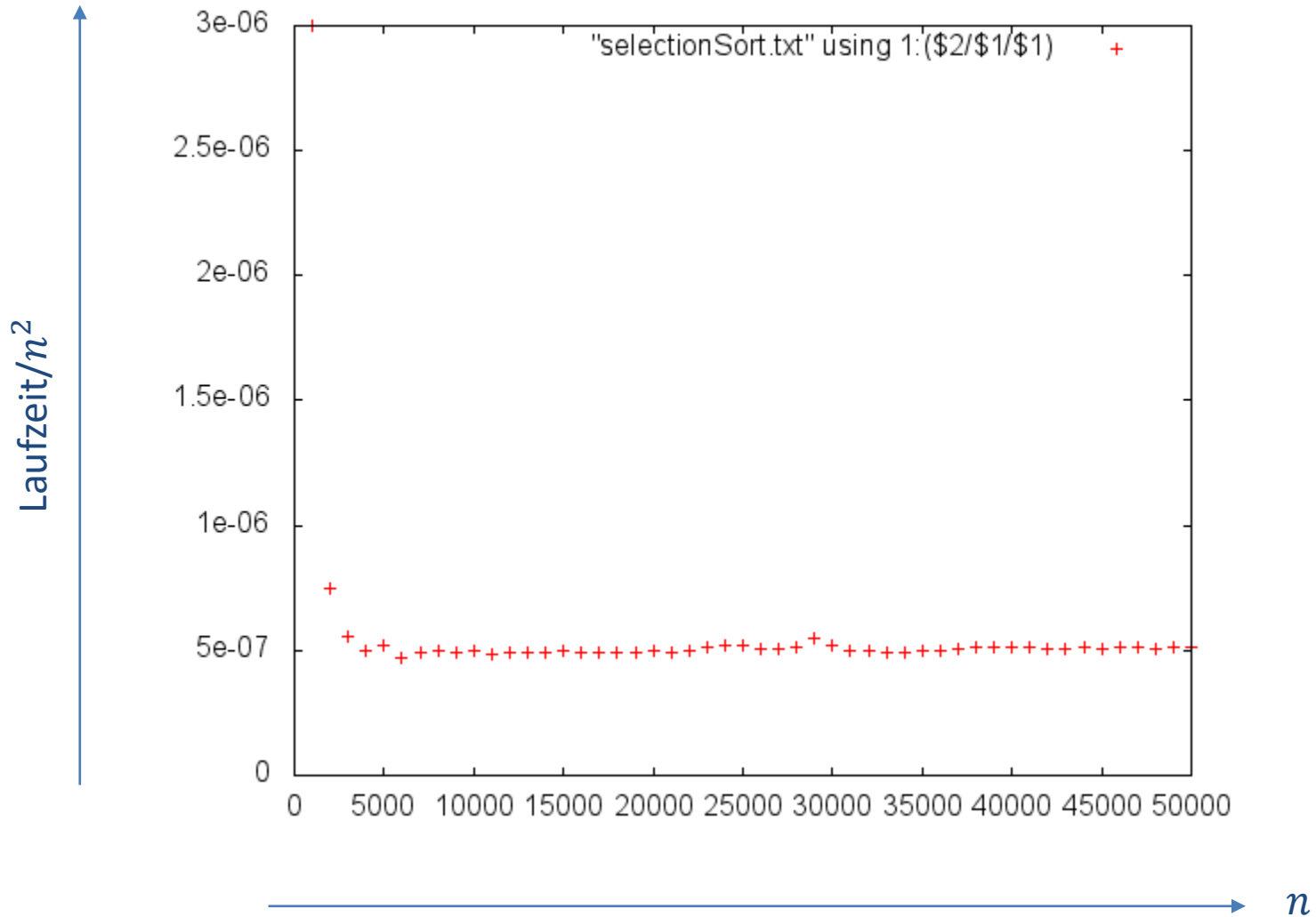
# Zeitmessung SelectionSort



# Zeitmessung SelectionSort



# Zeitmessung SelectionSort



- 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:



## Was ist eine Grundoperation?

- Einfache arithmetische Operationen
  - $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$  (mod), ...
- Ein Speicherzugriff
  - Variable auslesen, Variablenzuweisung
  - Ist das wirklich eine Grundoperation?
- 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 = 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...

**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):

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

# Selection Sort: Analyse

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

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

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

## 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:

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

- 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  $n \geq n_0$

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

- 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  $n \geq n_0$

$$\Theta(g(n)) := O(g(n)) \cap \Omega(g(n))$$

- 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  $f(n) \in \Omega(n)$



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

- 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  $c$ )

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

- 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:

$$f(n) \in o(g(n)) \implies f(n) \in O(g(n))$$

$$f(n) \in \omega(g(n)) \implies f(n) \in \Omega(g(n))$$

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

- $f(n) \leq g(n)$ , 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  $g(n)$

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

- $f(n) \ll g(n)$ , asymptotisch gesehen...
- $f(n)$  wächst asymptotisch langsamer als  $g(n)$

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

- $f(n) \gg g(n)$ , asymptotisch gesehen...
- $f(n)$  wächst asymptotisch schneller als  $g(n)$

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

$$f(n) \in o(g(n)) \iff f(n) \notin \Omega(g(n))$$

$$f(n) \in \omega(g(n)) \iff f(n) \notin O(g(n))$$

# Definition über Grenzwerte (vereinfacht)

Folgende Definitionen gelten für monoton wachsende Funktionen

$$f(n) \in O(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) \in \Theta(g(n)), \quad 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in o(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \in \omega(g(n)), \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

## Schreibweise:

- $O(g(n)), \Omega(g(n)), \dots$  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  $x \rightarrow 0$
- z.B. Taylor-Reihen:  $e^x = 1 + x + O(x^2)$ , bzw.  $e^x = 1 + x + o(x)$

## Alternative Definition für $\Omega(g(n))$ :

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

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

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

## Selection Sort:

- Laufzeit  $T(n)$ , es gibt Konstanten  $c_1, c_2 : c_1 n^2 \leq T(n) \leq c_2 n^2$   
 $T(n) \in O(n^2), \quad T(n) \in \Omega(n^2), \quad T(n) \in \Theta(n^2)$
- $T(n)$  wächst schneller als linear:  $T(n) \in \omega(n)$

## Weitere Beispiele:

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

# Analyse Bubble Sort

BubbleSort(A):

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

# Analyse Insertion Sort

InsertionSort(A):

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



## Worst Case Analyse

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

## 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**

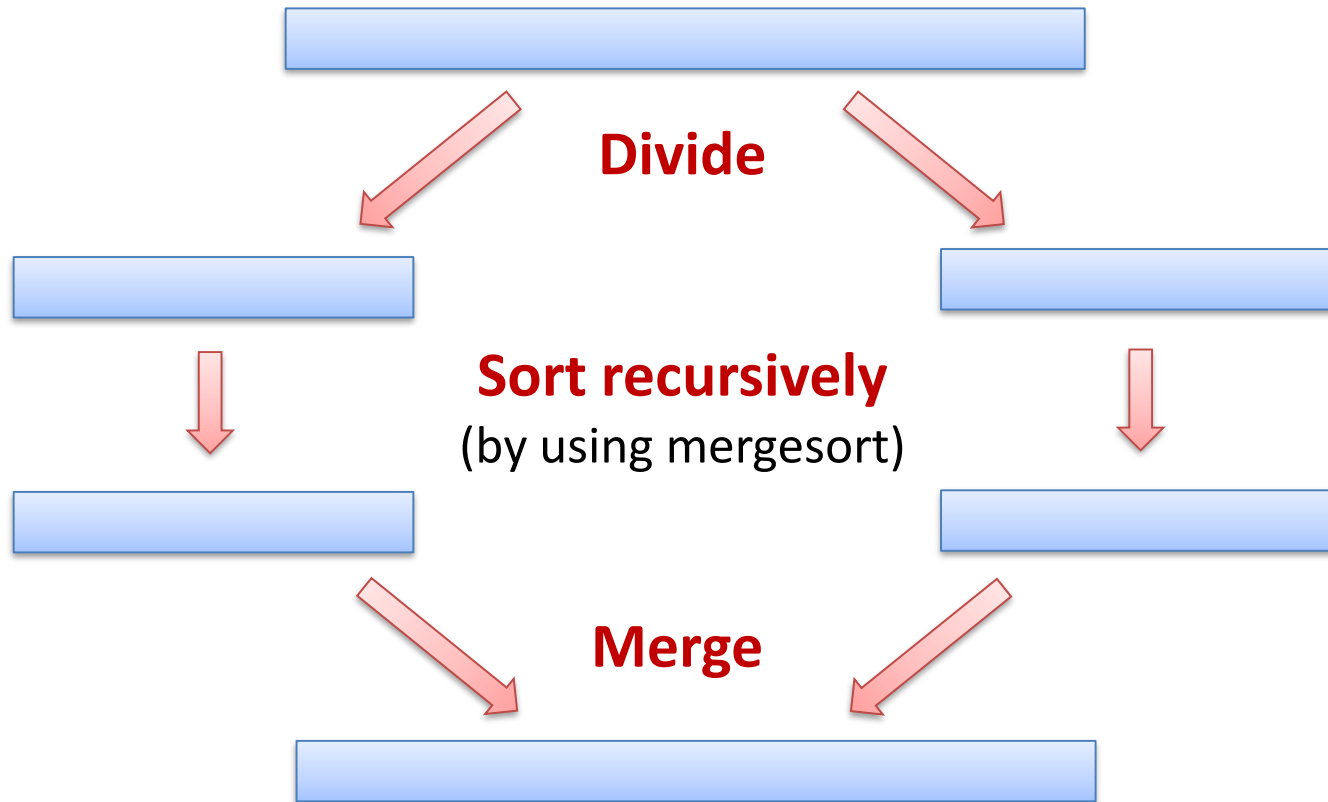
– das wächst für grosse  $n$  schon ziemlich schnell...

## 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     | $\approx$ 4KB    | $10^{3 \cdot 2} \cdot 10^{-9} \text{ s} = 1 \text{ ms}$       |
| $10^6$ Zahlen     | $\approx$ 4MB    | $10^{6 \cdot 2} \cdot 10^{-9} \text{ s} = 16.7 \text{ min}$   |
| $10^9$ Zahlen     | $\approx$ 4GB    | $10^{9 \cdot 2} \cdot 10^{-9} \text{ s} = 31.7 \text{ Jahre}$ |

**für grosse Probleme zu langsam!**



- Divide ist trivial  $\rightarrow$  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]
  :
5:   pos = start; i = start; j = middle
6:   while (pos < end) do
7:     if (i < middle) and (A[i] < A[j]) then
8:       tmp[pos] = A[i]; pos++; i++
9:     else
10:      tmp[pos] = A[j]; pos++; j++
11:   for i = start to end-1 do A[i] = tmp[i]
```

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

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

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

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