Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 2 (23.4.2018)

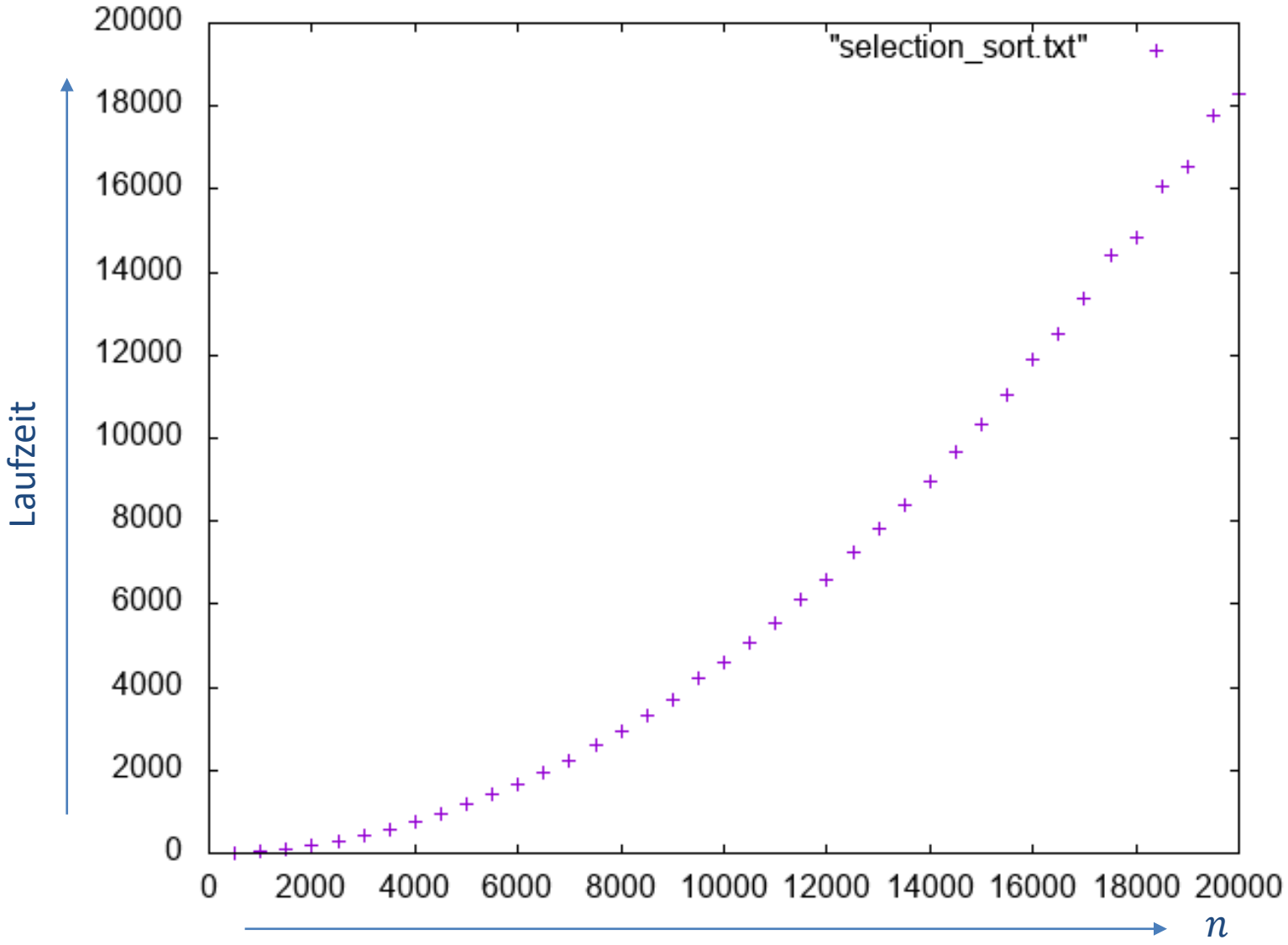
Sortieren II

UNI FREIBURG

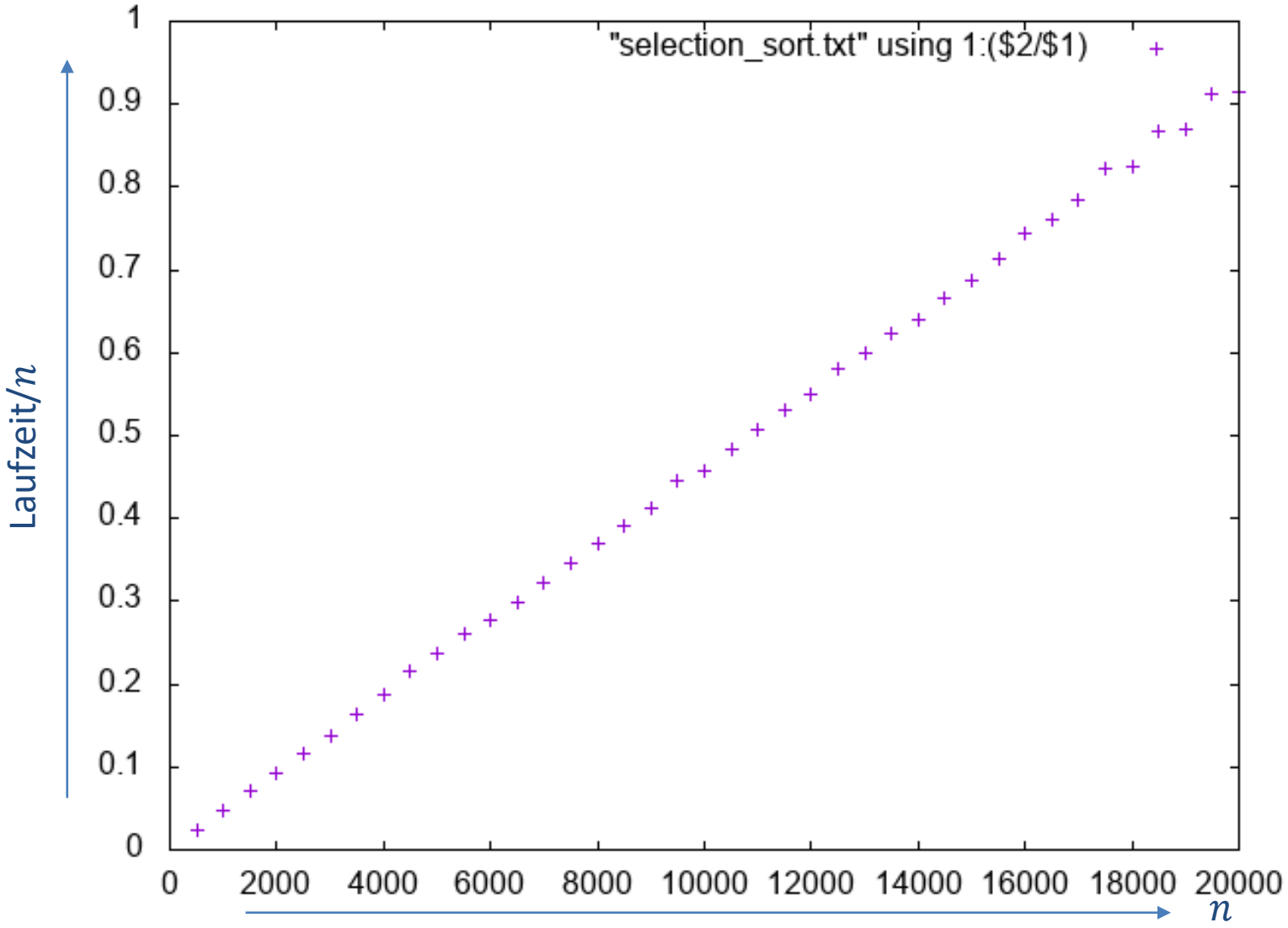
Fabian Kuhn

Algorithmen und Komplexität

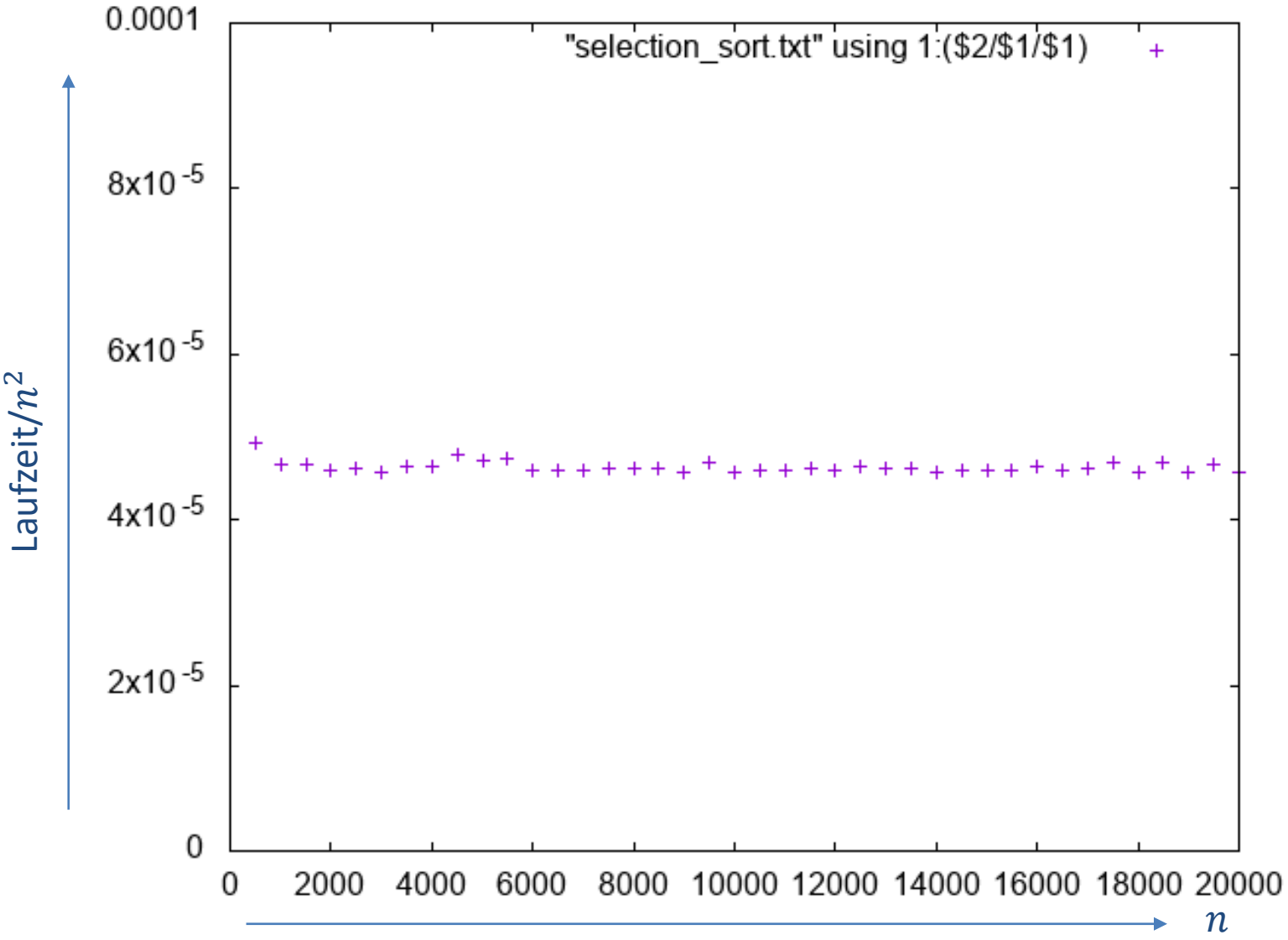
Zeitmessung SelectionSort



Zeitmessung SelectionSort



Zeitmessung SelectionSort



Beobachtungen Selection Sort

UNI FREIBURG

Zeitmessung Selection Sort:

- Scheint mit wachsender Grösse des Arrays unverhältnismässig langsamer zu werden
- Die Zeit scheint etwa quadratisch mit der Grösse des Arrays zu wachsen
  - doppelt so grosses Array → 4 x so lange Laufzeit
  - dreimal so grosses Array → 9 x so lange Laufzeit
  - …
- Wir werden sehen, dass die Laufzeit tatsächlich quadratisch ist
  - und wie man das formal korrekt analysiert und ausdrückt
- Zuerst überlegen wir, wie man sonst noch sortieren kann...

Insertion Sort - Idee

- Anfang (Präfix) des Arrays ist sortiert
  - Am Anfang nur das erste Element, mit der Zeit mehr...
- Füge schrittweise immer das nächste Element in den bereits sortierten Teil ein.

Beispiel: $A = [15,\ \ 3,\ \ 27,\ \ 49,\ \ 23,\ \ 18,\ \ 6,\ \ 1,\ \ 31]$

Insertion Sort: Pseudocode

Eingabe: Array A der Grösse n

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

Bubble Sort

- Gehe durch's Array, vertausche benachbarte Elemente, falls sie in der falschen Reihenfolge sind
- Wiederhole bis das Array sortiert ist...

Beispiel: $A = [15,\ \ 3,\ \ 27,\ \ 49,\ \ 23,\ \ 18,\ \ 6,\ \ 1,\ \ 31]$

Bubble Sort: Pseudocode

Eingabe: Array A der Grösse n

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

- Wir werden sehen: Insertion Sort und Bubble Sort sind auch nicht besser als Selection Sort …

QuickSort : Idee

1. Teile Array in zwei Teile auf
   - linker Teil: kleine Elemente, rechter Teil: grosse Elemente
2. Sortiere die beiden Teile rekursiv!

QuickSort Algorithmus

UNI FREIBURG

Informelle Beschreibung:

1. Teile Array in linken und rechten Teil, so dass

$$\textbf{Elemente links} \leq \textbf{Elemente rechts}$$

   – Bemerkung: Die Elemente in beiden Teilen müssen noch nicht sortiert sein

2. a) Sortiere die Elemente im linken Teil **rekursiv**
   b) Sortiere die Elemente im rechten Teil **rekursiv**

   – Rekursion: “löse ein kleineres Teilproblem der gleichen Art mit der gleichen Methode wie das Hauptproblem”

- Sobald die Teilprobleme so klein werden, dass Sortieren trivial wird, endet die Rekursion
  – spätestens wenn die Teile nur noch aus einem Element bestehen

QuickSort : Beispiel

Beispiel: $A = [15,\ 3,\ 27,\ 49,\ 23,\ 18,\ 6,\ 1,\ 31]$

QuickSort : Aufteilen des Arrays

UNI FREIBURG

- Wir müssen so aufteilen, dass

  $$\text{Elemente im linken Teil} \leq \text{Elemente im rechten Teil}$$

- **Idee:** Wähle ein **Pivot x**, welches bestimmt wo die Mitte ist
  - Elemente $< x$ müssen nach links
  - Elemente $> x$ müssen nach rechts
  - Bei Elementen $= x$ ist's egal…

- **Algorithmus für Divide** (oft auch Partition genannt)**:**
  - Idee: Iteriere von links und von rechts über's Array
  - Wenn man ein Element trifft, das auf der richtigen Seite ist, muss man nichts tun und kann weiter gehen.
  - Bei einem Element, welches die Seite wechseln muss, kann man mit einem Element auf der anderen Seite vertauschen, welches auch die Seite wechseln muss.

QuickSort : Aufteilen des Arrays

UNI FREIBURG

Algorithmus für Divide (etwas formaler)**:**

- Aufgabe: Divide array A der Länge n anhand von Pivot x
  - Annahme: Elem. $\leq x$ gehen nach links, elem. $> x$ gehen nach rechts

- Generelles Vorgehen:
  - Zwei Variablen l und r, um von links und rechts durch's Array zu gehen
  - Inkrementiere l bis $A[l] > x$ (Element muss nach rechts)
  - Dekrementiere r bis $A[r] \leq x$ (Element muss nach links)
  - Vertausche, und stelle l und r eins vor ($l \mathrel{+}= 1$, $r \mathrel{-}= 1$)
  - Divide fertig, sobald sich l und r treffen

- Die Details müssen Sie in der Übung selbst ausarbeiten…

QuickSort Divide : Beispiel

Beispiel: $A = [15,\ 17,\ 3,\ 22,\ 27,\ 49,\ 9,\ 23,\ 18,\ 6,\ 1,\ 31]$

QuickSort : Wahl des Pivots

UNI FREIBURG

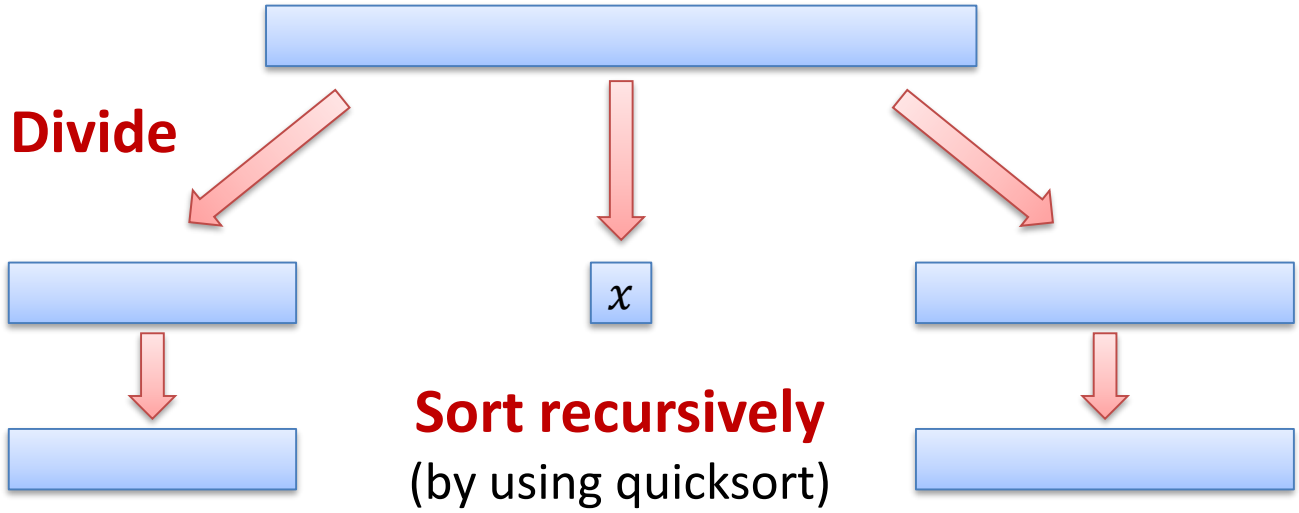
- Wie gross die zwei Teile beim Divide werden, hängt von der Wahl des Pivots ab…
- Wir werden sehen: Der Algorithmus ist am schnellsten, wenn die zwei Teile möglichst gleich gross sind.

Strategien zur Bestimmung des Pivots:

- Ideal wäre der Median → kann nicht einfach gefunden werden…
  - werden wir noch anschauen...
- Ein fixes Element des Arrays (z.B. immer das erste des Bereichs)
  - kann zu sehr ungleichen Teilen führen...
- Ein Element an einer zufälligen Position (innerhalb des Bereichs)
  - Randomized QuickSort meint meistens genau das
  - Wird meistens vernünftig grosse Teile liefern
- Median von drei (oder mehr) zufälligen Elementen
  - etwas “teurer”, dafür werden die Teile “gleicher”

Divide and Conquer

Übersicht QuickSort:



Divide and Conquer:

- Verbreitetes Prinzip für den Algorithmenentwurf

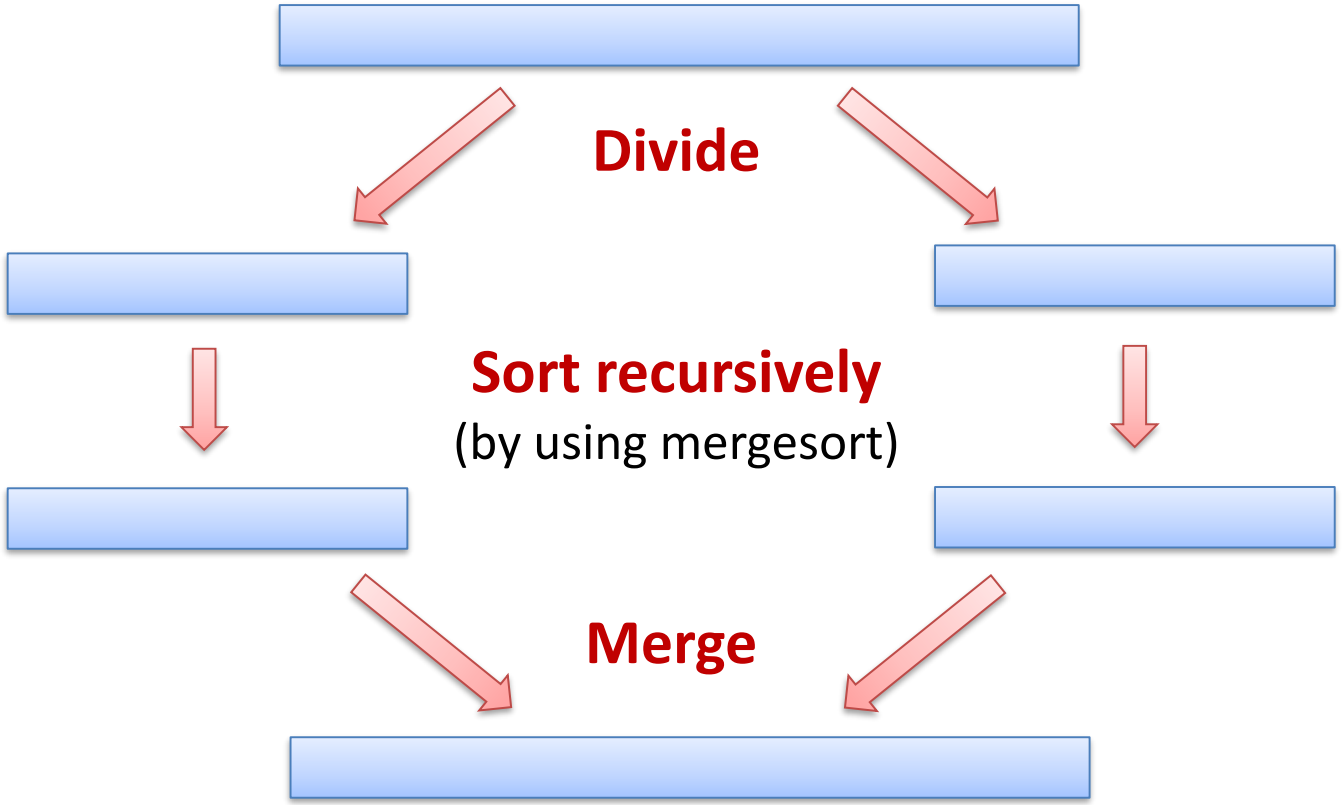
1. Teile Eingabe in 2 oder mehrere kleinere Teilprobleme
2. Löse die Teilprobleme rekursiv
3. Kombiniere die Teillösungen zur Gesamtlösung

MergeSort

- Ein weiterer Sortieralgorithmus, welcher auf dem Divide-and-Conquer Prinzip basiert

Beispiel: $A = [15,\ 17,\ \ 3,\ 22,\ 27,\ 49,\ 9,\ 23,\ 18,\ 6,\ 1,\ 31]$

Übersicht MergeSort



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

MergeSort: Merge-Schritt

Verschmelzen (merge) von zwei sortierten Arrays:

- Gegeben: sortierte Arrays A und B der Länge n und m
- Ausgabe: sortiertes Array C mit den Elementen von A und B

Vorgehen:

MergeSort: Pseudocode

Eingabe: Array A der Grösse n

```
MergeSort(A):
 1: allocate array tmp to store intermediate results
 2: MergeSortRecursive(A, 0, n, tmp)

MergeSortRecursive(A, start, end, tmp)          // sort A[start..end-1]
 1: if end – start > 1 then
 2:   middle = start + (end – start) / 2        // int. division
 3:   MergeSortRecursive(A, start, middle, tmp)
 4:   MergeSortRecursive(A, middle, end, tmp)
 5:   pos = start; i = start; j = middle
 6:   while pos < end do
 7:     if 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]
```

Laufzeitanalyse I

UNI FREIBURG

- 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

UNI FREIBURG

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 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

UNI FREIBURG

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: Obere Schranke

UNI FREIBURG

$\boldsymbol{T(n)}$**:** 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

UNI FREIBURG

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”)

UNI FREIBURG

- 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:

Landau-Symbole : Definitionen

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

- 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 $n \geq n_0$

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

- 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 $n \geq n_0$

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

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

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:

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

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

Landau-Symbole : Intuitiv

UNI FREIBURG

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

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