Algorithmen und Datenstrukturen

Vorlesung 1

Sortieren I

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Sortieren

UNI FREIBURG

Problemdefinition

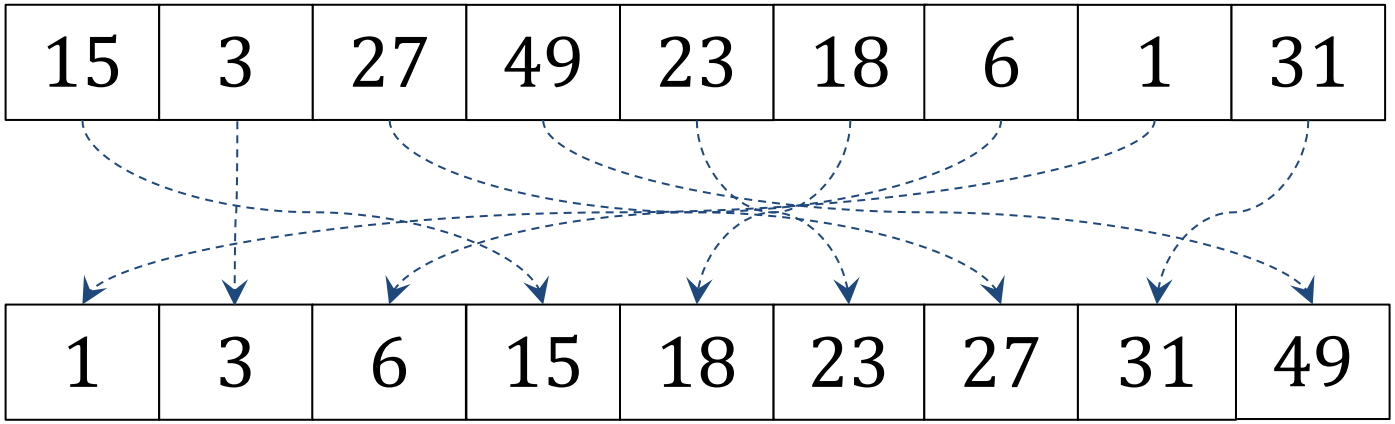
- **Eingabe**: Sequenz von n Elementen $x_1, \dots, x_n$
- Ordnungsrelation $\leq$ auf den Elementen
  - Vergleichsoperation, mit welcher zwei beliebige Elemente miteinander verglichen werden können
  - Bsp. 1: Sequenz von Zahlen mit üblicher $\leq$-Relation
  - Bsp. 2: Sequenz von Strings mit lexikographischer (alphabetischer) Ordnung
- **Ausgabe**: Gemäss $\leq$-Ordnung sortierte Sequenz der n Elemente
- **Beispiel**:
  - Eingabe: $[15, 3, 27, 49, 23, 18, 6, 1, 31]$
  - Ausgabe: $[1, 3, 6, 15, 18, 23, 27, 31, 49]$
- Sortieren wird in (fast) allen grösseren Programmen gebraucht!

Algorithmus zum Sortieren?

Aufgabe: Sortiere Array A (z.B. $A = [15, 3, 27, 49, 23, 18, 6, 1, 31]$)

Einfache Idee:

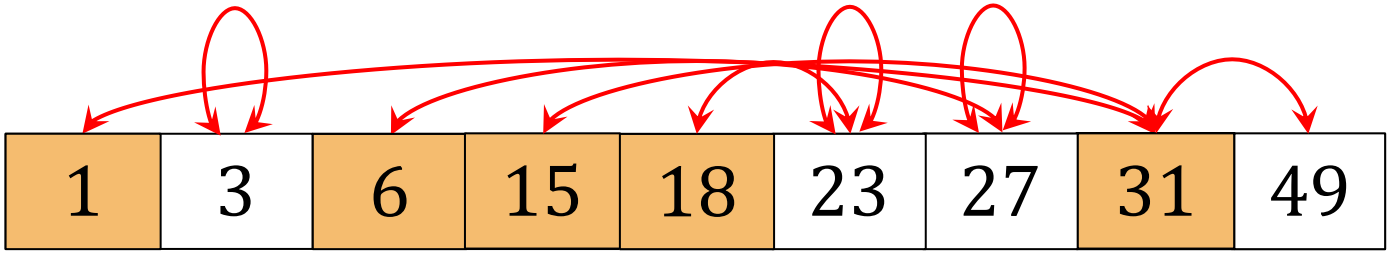
- Finde kleinstes Element und setze es an den Anfang
- Finde kleinstes Element unter den restlichen Elementen
- etc.



Selection Sort Algorithmus

SelectionSort (etwas genauer):

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. ...



Selection Sort im Detail

UNI FREIBURG

Eingabe: Array A der Grösse n

- In der Vorlesung und der Übung unterstützen wir offiziell Python als Programmiersprache.
- Um ein Beispiel zu sehen, schauen wir uns an, wie man den besprochenen Algorithmus konkret in Python implementieren kann.

Selection Sort: Pseudocode

Eingabe: Array A der Grösse n

```
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+1 to n-1 do
 5:     if A[j] < A[minIdx] then
 6:       minIdx = j

 7:   // swap A[i] with min of A[i..n-1]
 8:   tmp = A[i]
 9:   A[i] = A[minIdx]
10:   A[minIdx] = tmp
```

Selection Sort: Analyse

Algorithmenanalyse:

1. Algorithmus berechnet für jede Eingabe die korrekte Ausgabe
2. Algorithmus terminiert (Totale Korrektheit)

1. Zeige, dass der Algorithmus korrekt ist
   - Dazu braucht es einen formalen (mathematischen) Beweis
   - Meistens macht man das nicht ganz formal
   - Wie man sauber formal beweisen kann , dass ein Algorithmus / Programm das Richtige tut, werden wir später noch sehen.

2. Analysiere die Laufzeit und evtl. andere Eigenschaften
   - Wie man das macht, werden noch anschauen
   - Vorerst werden wir einfach messen und versuchen dadurch einen Eindruck zu erhalten, wie schnell ein Algorithmus ist

In der Vorlesung wird die Korrektheit meistens intuitiv klar sein und der Fokus wird eher auf der Analyse der Effizienz der Algorithmen liegen.

Selection Sort: Korrektheit

Eigenschaften nach Schleifendurchlauf i:

a) A enthält die gleichen Werte wie am Anfang

b) A[0..i] ist korrekt sortiert

c) "Werte in A[0..i]" $\leq$ "Werte in A[i+1..n-1]"

```
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+1 to n-1 do
 5:     if A[j] < A[minIdx] then
 6:       minIdx = j
 7:   swap(A[i], A[minIdx])
```

Beweis durch vollständige Induktion:

- **Induktionsverankerung ($i = 0$):**
  - a) folgt, da wir nur Werte vertauschen
  - b) trivial
  - c) folgt, da A[0] nach dem Vertauschen das kleinste Element in A enthält

Selection Sort: Korrektheit

Eigenschaften nach Schleifendurchlauf i:

a) A enthält die gleichen Werte wie am Anfang

b) A[0..i] ist korrekt sortiert

c) "Werte in A[0..i]" $\leq$ "Werte in A[i+1..n-1]"

```
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+1 to n-1 do
 5:     if A[j] < A[minIdx] then
 6:       minIdx = j
 7:   swap(A[i], A[minIdx])
```

Beweis durch vollständige Induktion:

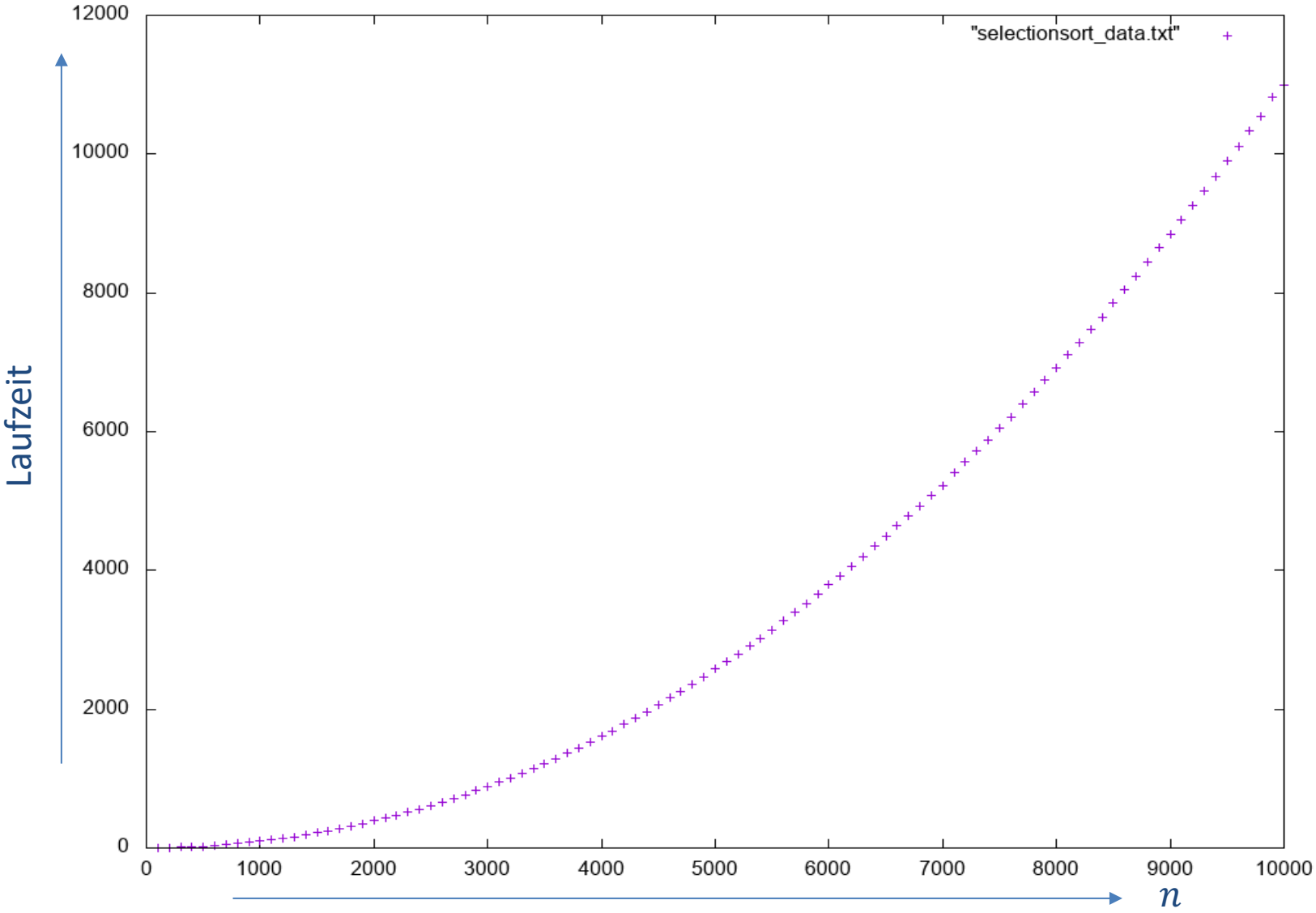
- **Induktionsschritt ($i > 0$):**
  - Induktionshypothese $\Longrightarrow$ a) gilt, A[0..i-1] ist korrekt sortiert, Werte in A[0..i-1] $\leq$ Werte in A[i..n-1]
  - a) folgt wieder, da nur vertauscht wird
  - Werte in A[0..i-1] werden in Schleifendurchlauf i nicht verändert, b) folgt daher direkt
  - c) folgt, da A[i] kleinstes Element in A[i..n-1] enthält

Messung der Laufzeit

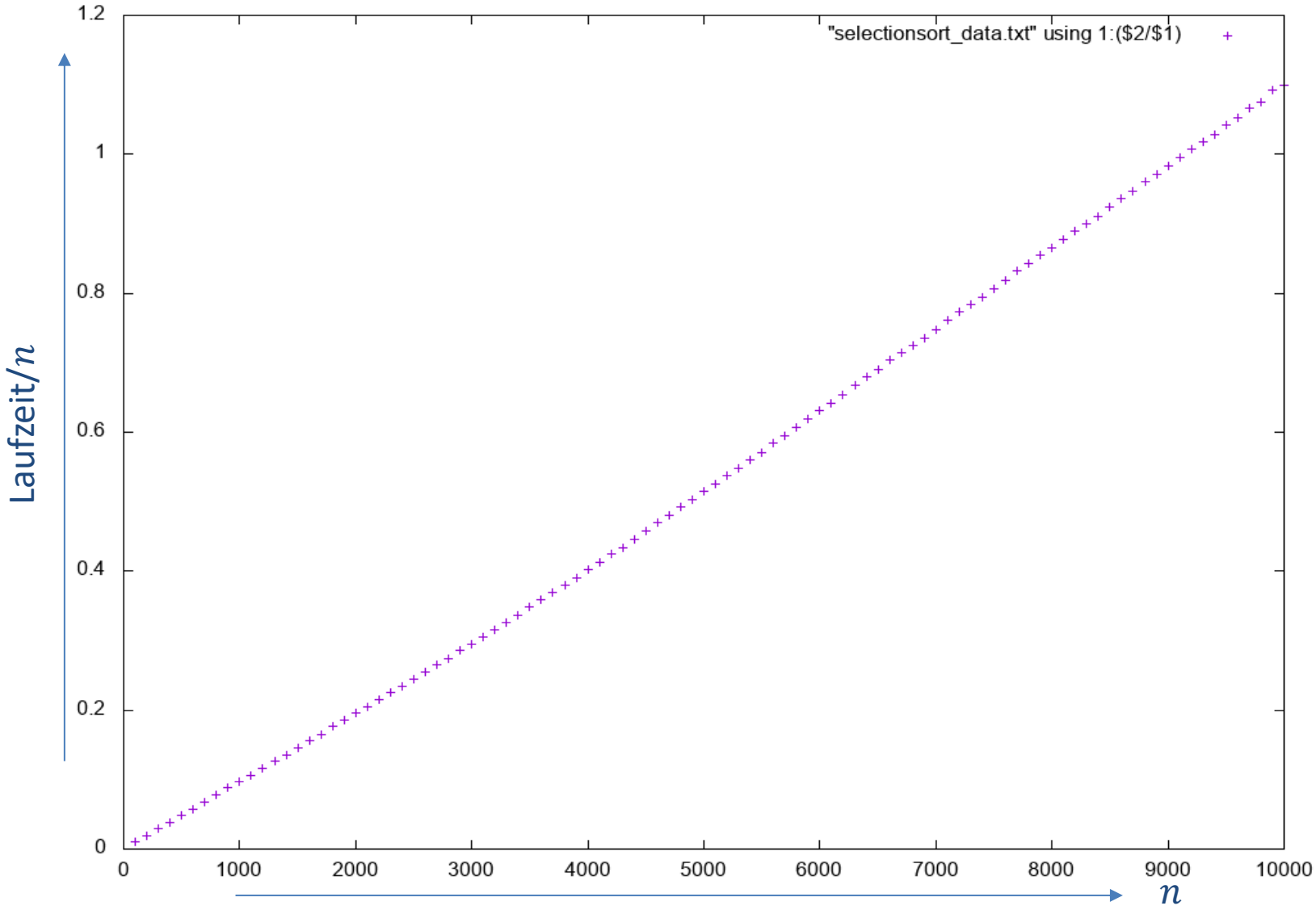
In Python:

```
import time

…

start_time = time.time();
// code segment for which you want to measure
// the running time.
run_time = (time.time() – startTime) * 1000;
```

Zeitmessung SelectionSort



Zeitmessung SelectionSort



Beobachtungen Selection Sort

UNI FREIBURG

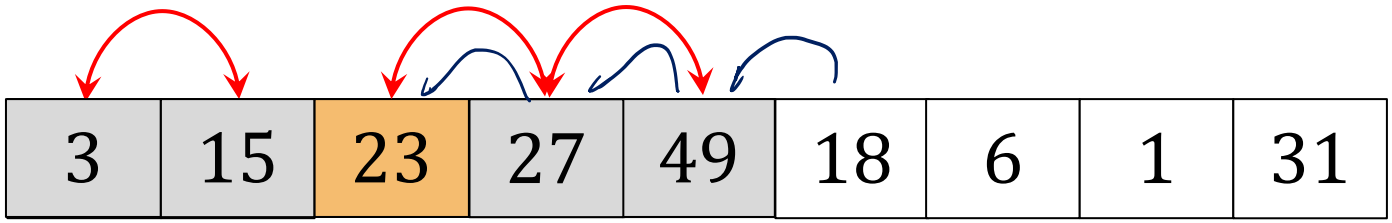
Zeitmessung Selection Sort:

- 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:



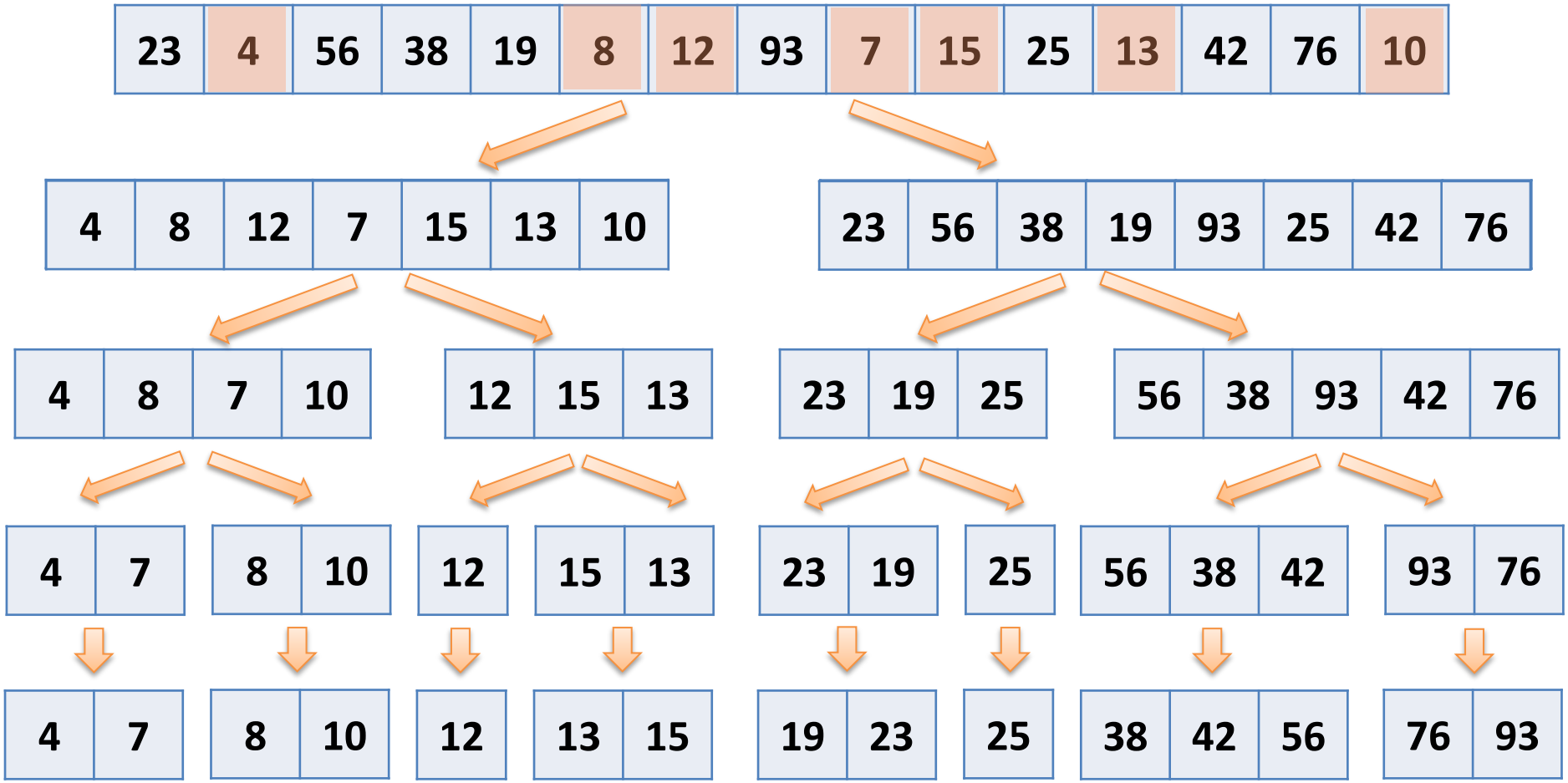
Insertion Sort: Pseudocode

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
```

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”

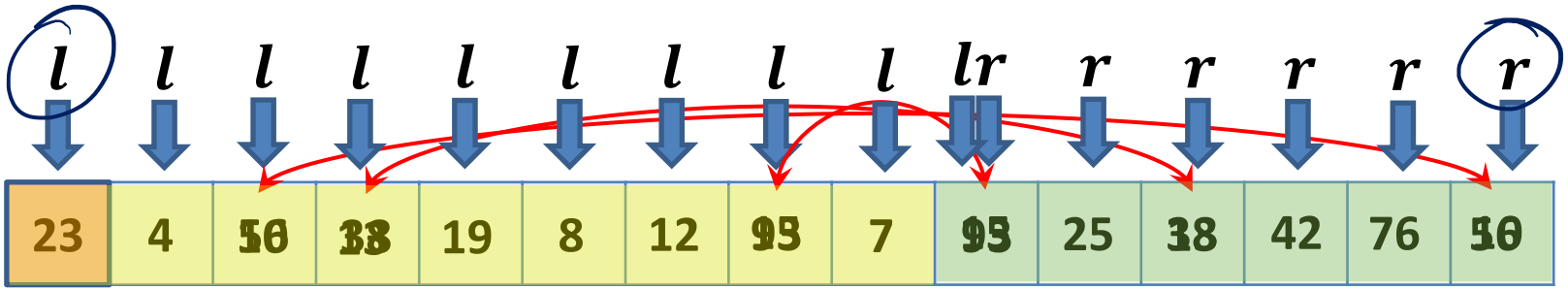
- Sobald die Teilprobleme so klein werden, dass Sortieren trivial wird, endet die Rekursion
  – spätestens wenn die Teile nur noch aus einem Element bestehen

QuickSort : Aufteilen des Arrays

- 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… (im nachfolgenden Bsp. nach links)



pivot $= 23$

1. Inkrementiere l solange $A[l] \leq \text{pivot}$
2. Dekrementiere r solange $A[r] > \text{pivot}$
3. Vertausche $A[l]$ und $A[r]$

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's Aufteilen** (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's Aufteilen (etwas formaler)**:**

- Aufgabe: Teile 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 : Wahl des Pivots

UNI FREIBURG

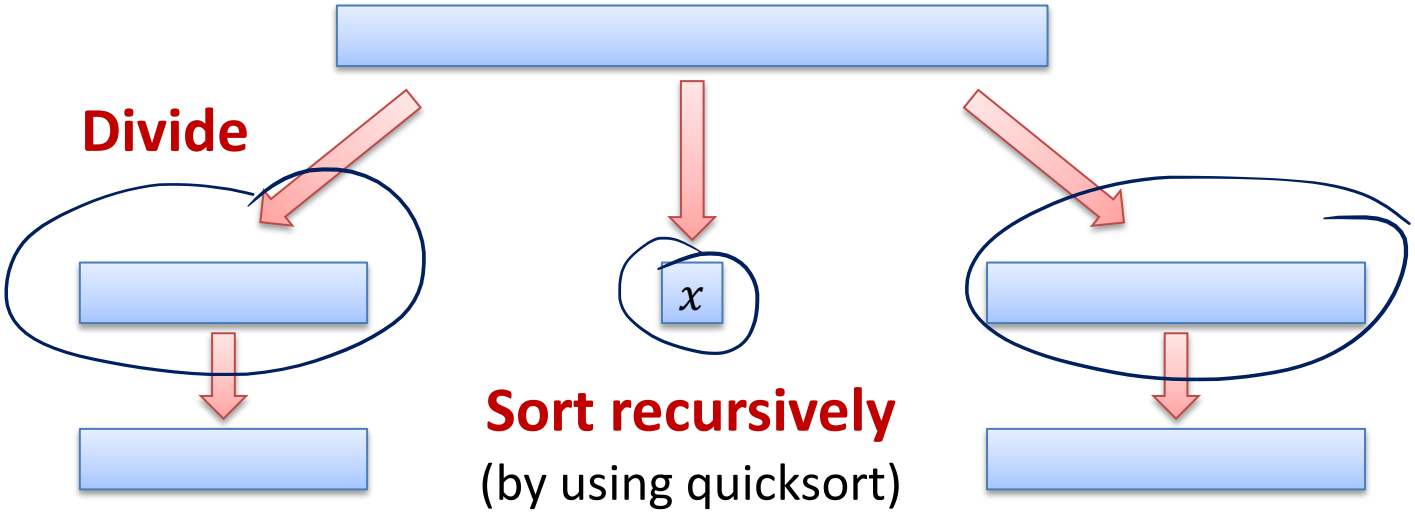
- Wie gross die zwei Teile beim Divide werden, hängt von der Wahl des Pivots ab…
- Wir werden sehen: Der Algorithmus ist am schnellsten, wenn die zwei Teile möglichst gleich gross sind.

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:



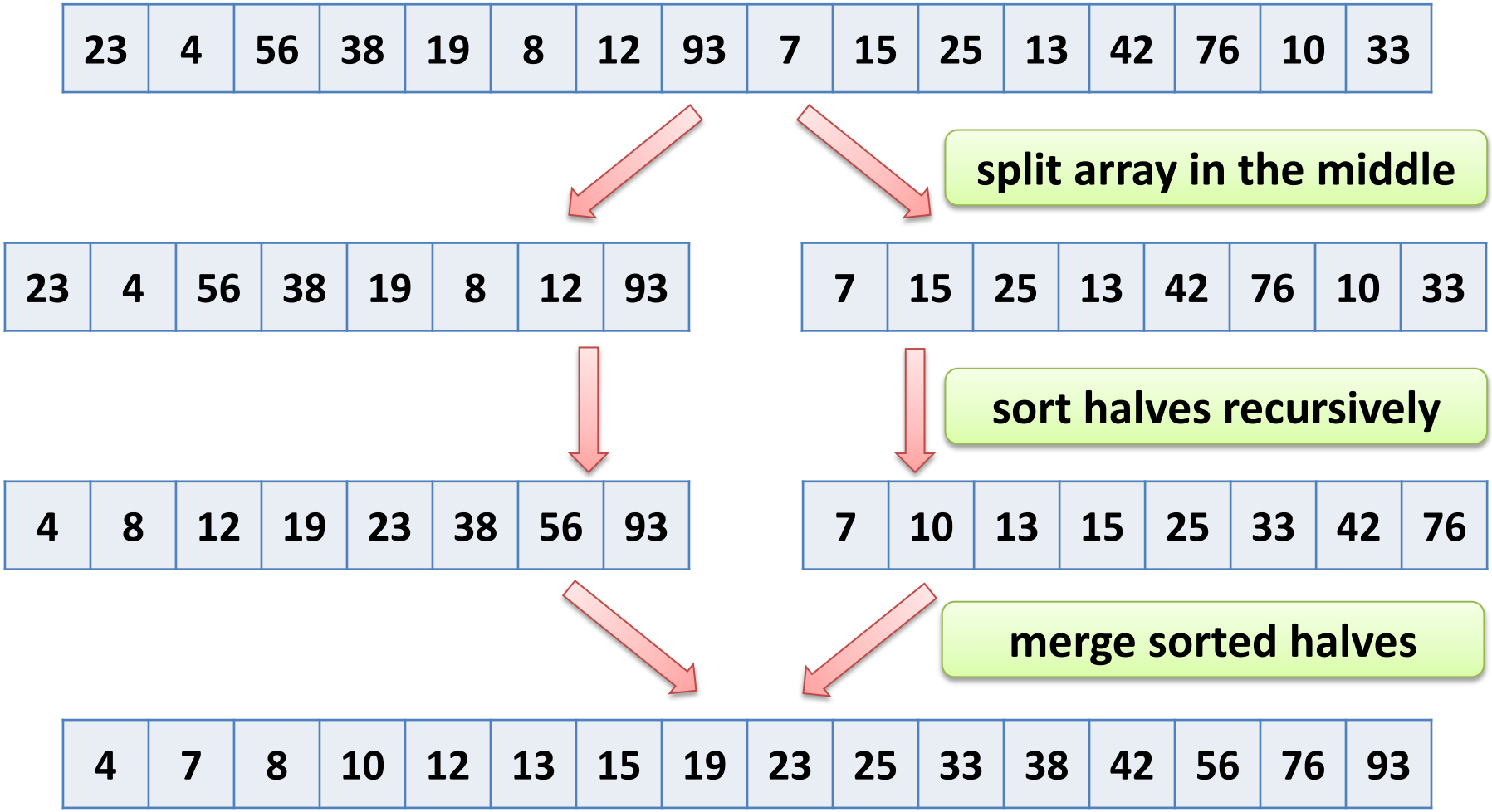
Divide and Conquer:

- Verbreitetes Prinzip für den Algorithmenentwurf

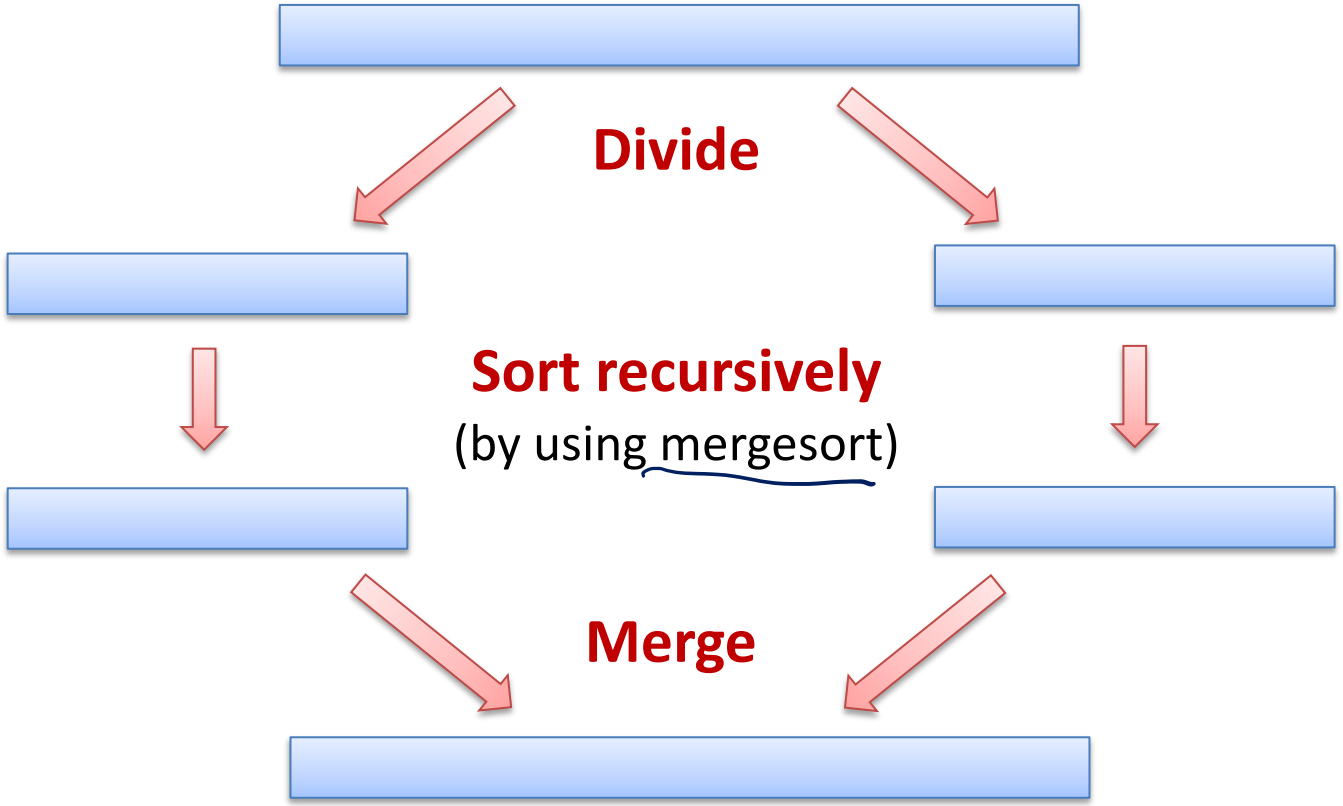
1. Teile Eingabe in 2 oder mehrere kleinere Teilprobleme
2. Löse die Teilprobleme rekursiv
3. Kombiniere die Teillösungen zur Gesamtlösung

MergeSort

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



Übersicht MergeSort



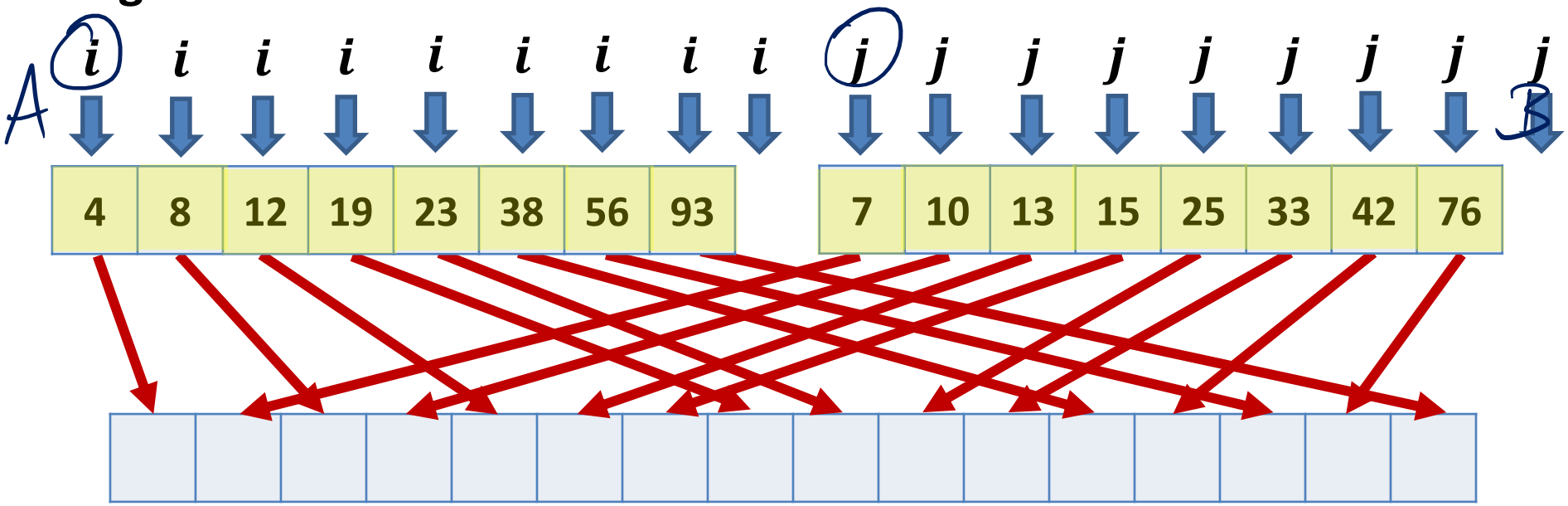
- Divide ist bei MergeSort trivial
- Merge (kombinieren der Lösungen) benötigt dafür Arbeit...

MergeSort: Merge-Schritt

Verschmelzen (merge) von zwei sortierten Arrays:

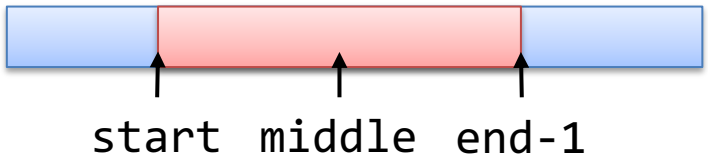
- Gegeben: sortierte Arrays A und B der Länge n und m
- Ausgabe: sortiertes Array C mit den Elementen von A und B

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        // integer 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 i < middle and (j >= right or 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]
```

Zusammenfassung Sortieralgorithmen

UNI FREIBURG

- Wir haben vier verschiedene Sortieralgorithmen gesehen:

Einfache Sortieralgorithmen: Selection Sort und Insertion Sort

- Selection Sort scheint recht langsam zu sein, unsere Messungen ergaben, dass die Laufzeit wohl quadratisch mit n steigt.
- Wir werden sehen, dass das für beide Alg. tatsächlich so ist.

Rekursive Sortieralgorithmen: QuickSort und MergeSort

- Beide sind nach dem Divide-and-Conquer Prinzip aufgebaut: Das Array der Grösse n wird in zwei kleinere Teilarrays aufgeteilt, die zwei Teilarrays werden rekursiv sortiert und die Lösungen werden zu einer sortierten Gesamtlösung zusammengefügt.
- Wir werden sehen, dass sich die zusätzliche Komplexität lohnt, beide Algorithmen sind viel besser als die einfachen zwei.