Algorithms and Data Structures Conditional Course

Lecture 1

Sorting I

UNI FREIBURG

Fabian Kuhn

Algorithms and Complexity

Sorting

UNI FREIBURG

Problem Definition

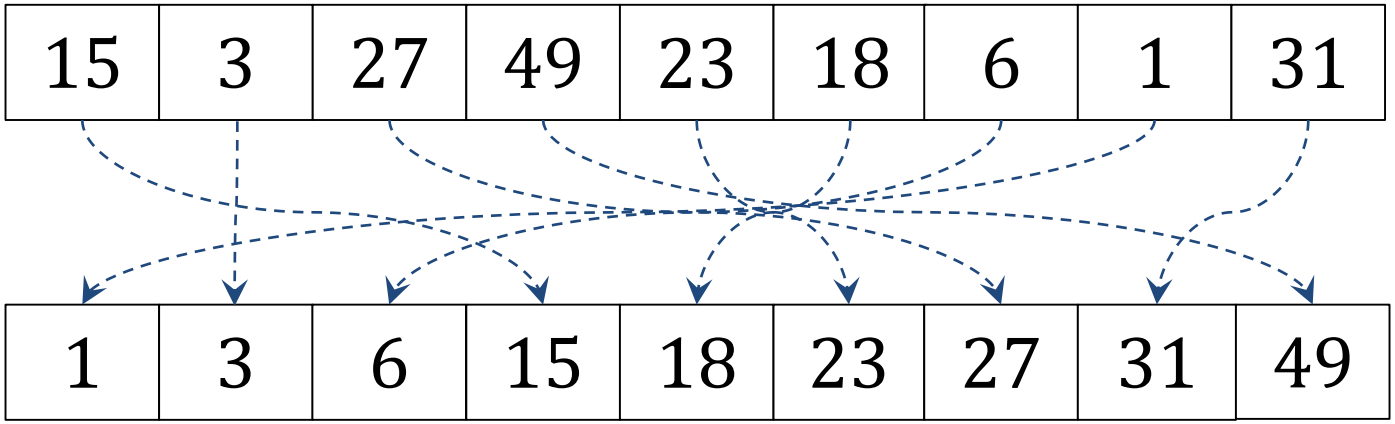
- **Input**: Sequence of n elements $x_1, \dots, x_n$
- Ordering relation $\leq$ on elements
  - Comparison operation that allows to compare two arbitrary elements
  - Ex. 1: Sequence of numbers with the usual $\leq$ relation
  - Ex. 2: Sequence of strings with lexicographic (alphabetical) order
- **Output**: Sorted sequence of the n elements according to order $\leq$
- **Example**:
  - Input: $[15, 3, 27, 49, 23, 18, 6, 1, 31]$
  - Output: $[1, 3, 6, 15, 18, 23, 27, 31, 49]$
- Sorting is used in (almost) all larger programs!

Algorithm for Sorting?

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

Simple Idea:

- Find smallest element und put it to the beginning
- Find smallest element among the remaining elements
- etc.



Selection Sort Algorithm

SelectionSort ():

1. Find smallest element in array, swap it to 1st position
2. Find smallest element in rest, swap it to 2nd position
3. Find smallest element in rest, swap it to 3rd position
4. ...

15	3	27	49	23	18	6	1	31

Selection Sort in Detail

Input: Array A of size n

- In the lecture and exercises, we officially support Python as a programming language.
- To see an example, we will next show how one can implement the discussed algorithm in Python.

Selection Sort: Pseudocode

Input: Array A of size 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: Analysis

Algorithm Analysis:

1. Algorithm computes correct output for every possible input
2. Algorithm terminates (total correctness)

1. Show that the algorithm is correct
   - This requires a formal (mathematical) proof
   - Usually this is not done completely formal
   - We will see later how one can rigorously prove that an algorithm / computer program does what it is supposed to do.

2. Analyze the running time and possibly other properties
   - We will see next week how this is done
   - For now, we will just measure the performance and try to get an impression of how fast an algorithm is.

In this lecture, correctness will mostly be clear intuitively and we will mostly focus on analyzing an understanding the efficiency of algorithms.

Selection Sort: Correctness

Properties after iteration i

a) A contains the same values as in the beginning

b) A[0..i] is sorted correctly

c) "values in A[0..i]" $\leq$ "values 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])
```

Proof by induction:

- **Induction basis ($i = 0$):**
  - a) follows because we only swap values
  - b) trivial
  - c) follows because after swapping in iteration $i = 0$, A[0] contains the smallest element of A

Selection Sort: Correctness

Properties after iteration i:

a) A contains the same values as in the beginning

b) A[0..i] is sorted correctly

c) "values in A[0..i]" $\leq$ "values 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])
```

Proof by induction:

- **Induction step ($i > 0$):**
  - Induction hypothesis $\Longrightarrow$ a) holds, A[0..i-1] is sorted correctly, values in A[0..i-1] $\leq$ values in A[i..n-1]
  - a) again follows because we only swap values
  - Values in A[0..i-1] are not changed in iteration i, b) thus follows
  - c) follows because A[i] contains smallest element of A[i..n-1]

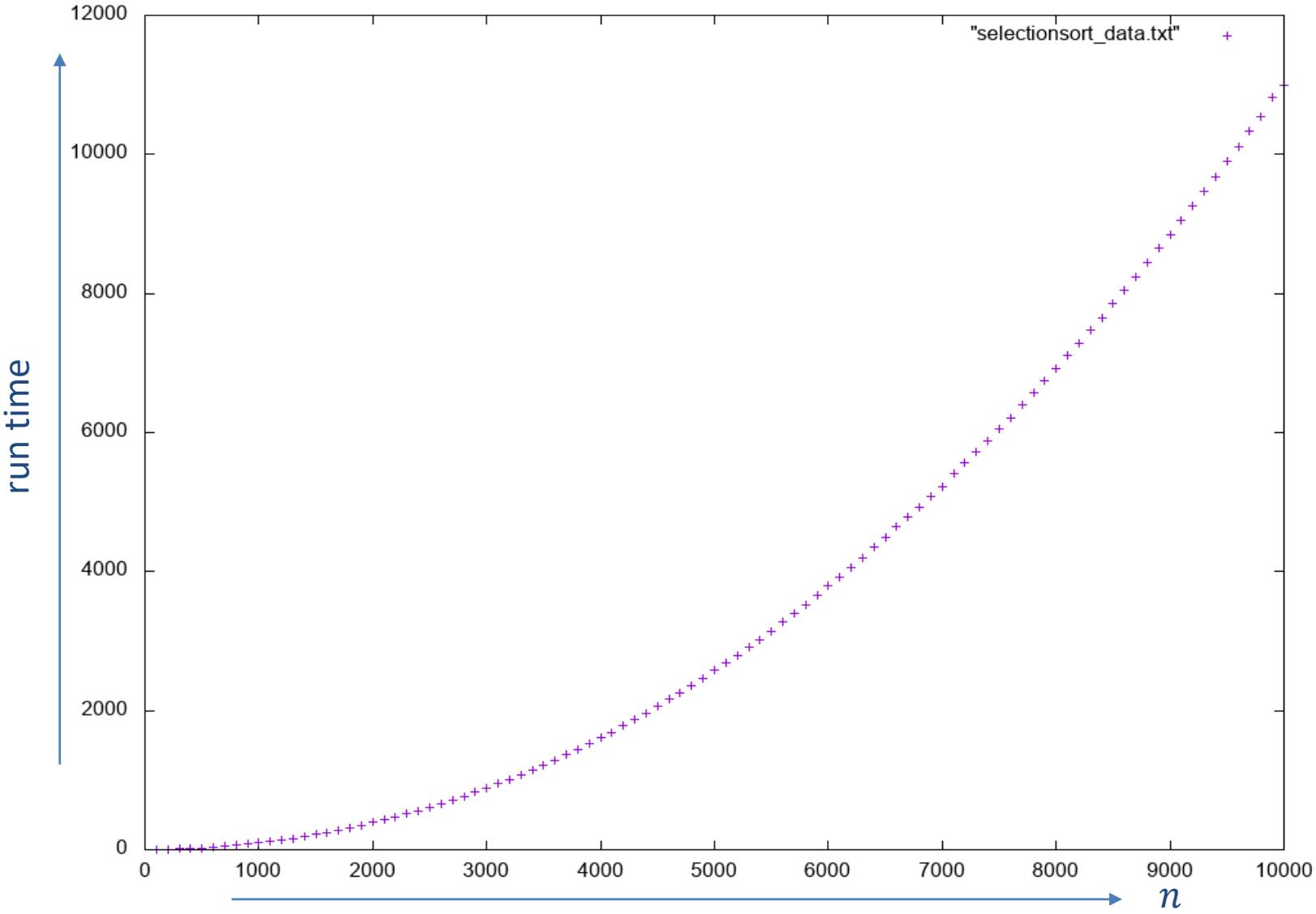
Measuring the Running Time

UNI FREIBURG

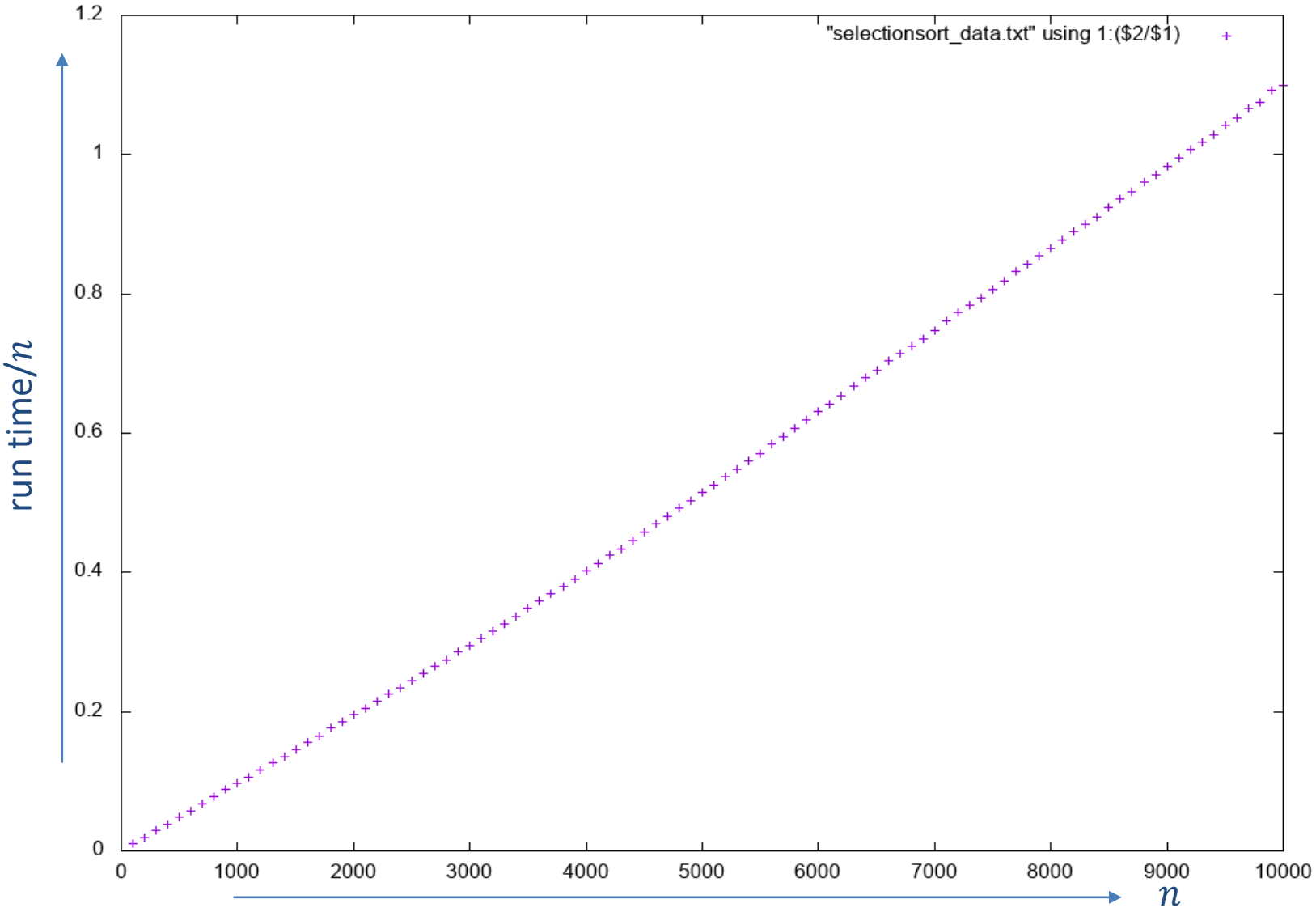
In Python:

```
import time

…

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

Time Measurement Selection Sort



Time Measurement Selection Sort



Observations Selection Sort

UNI FREIBURG

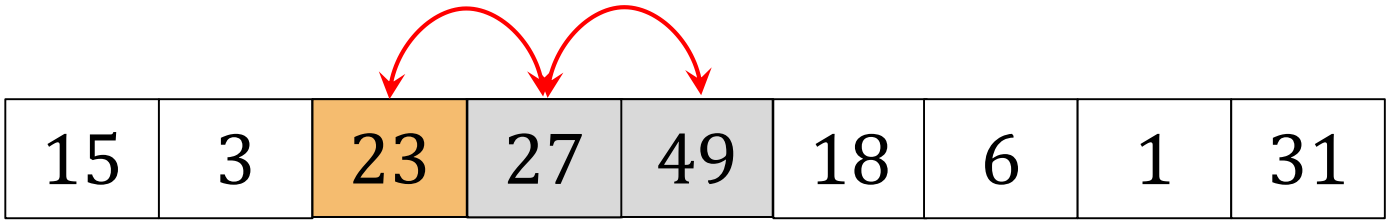
Time Measurements Selection Sort:

- Seems to become slower over-proportionally when the size of the arrow increases
- The time seems to grow roughly quadratically with the size of the array.
  - Array 2x the size → 4 x so lange Laufzeit
  - Array 3x the size → 9 x so lange Laufzeit
  - …
- We will see that the running time is indeed quadratic
  - and how this is analyzed and expressed properly in a formal way
- Let us first think about other ways to sort...

Insertion Sort - Idea

- Beginning (prefix) of array is sorted
  - At the beginning only the first element, later more...
- Step-by-step, always insert the next element into the always sorted part of the array.

Example:



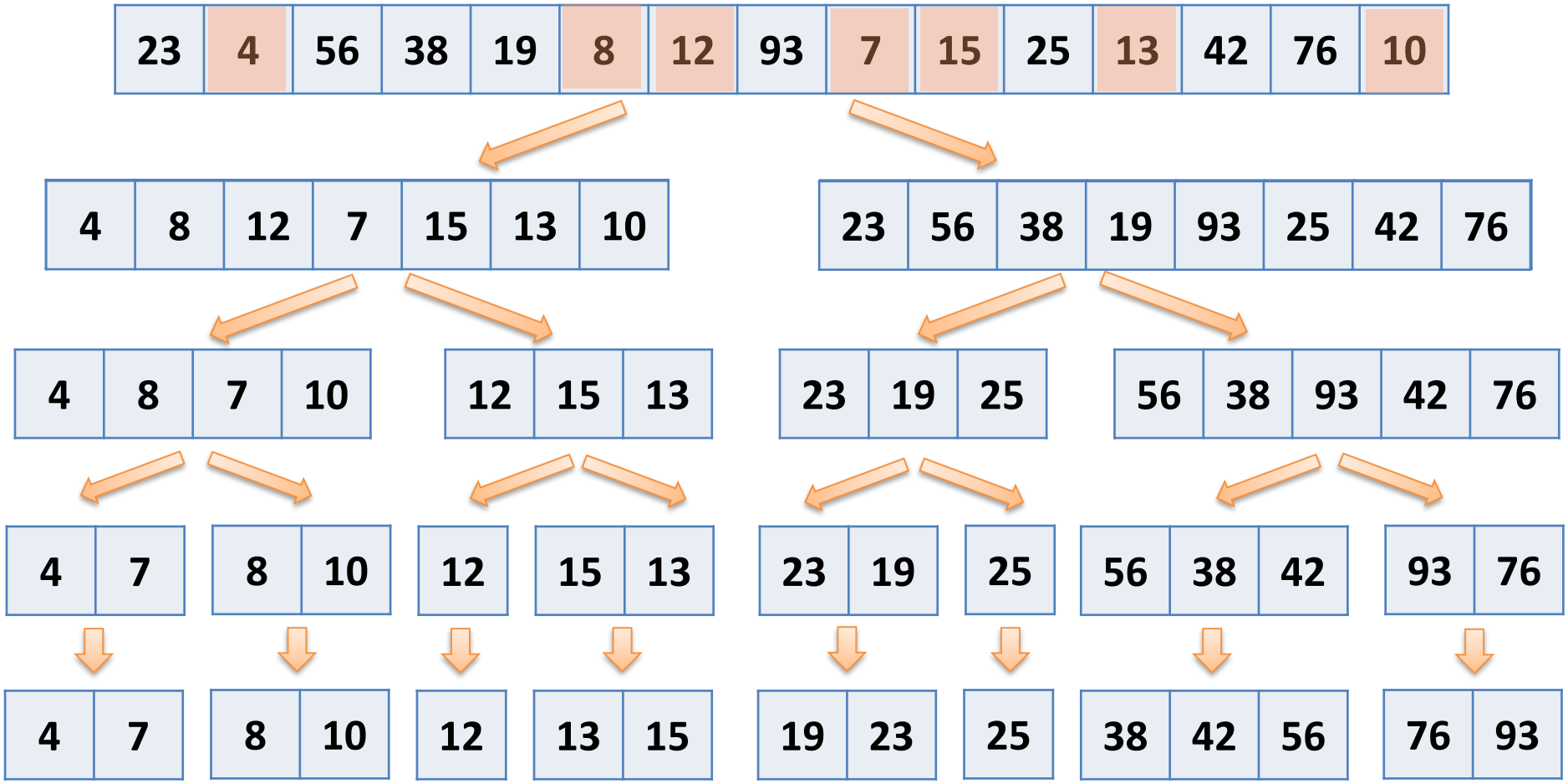
Insertion Sort: Pseudocode

Input: Array A of size 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 : Idea

1. Divide array into two parts
   - left part: small elements, right part: large elements
2. Sort the two parts recursively!



QuickSort Algorithm

UNI FREIBURG

Informal Description:

1. Divide array into left and right part such that

$$\textbf{elements left} \leq \textbf{elements right}$$

   – Remark: The elements in both parts do not need to be sorted

2. a) Sort elements in left part **recursively**
   b) Sort elements in right part **recursively**

   – Recursion: "solve a smaller sub problem of the same kind and with the same method as the main problem"

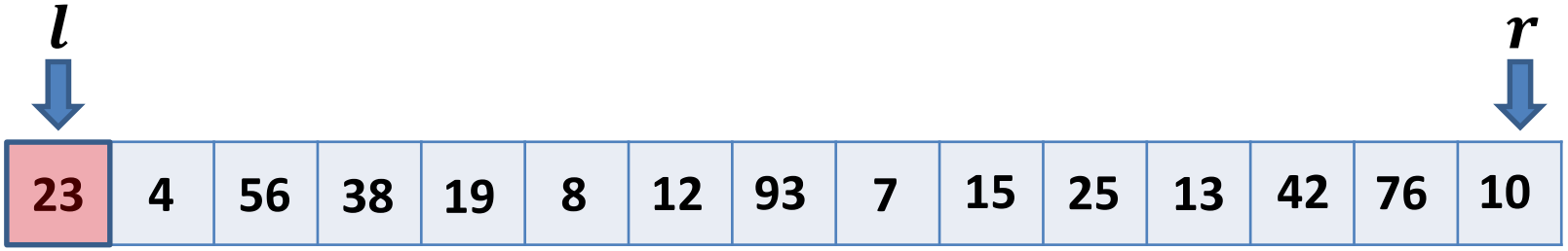
- As soon as the sub problems become small enough such that sorting becomes trivial, the recursion ends
  – at the latest if the parts consist of a single element

QuickSort : Partitioning the Array

- We have to partition such that

$$\text{Elements in left part} \leq \text{Elements in right part}$$

- **Idea:** Choose a **pivot x** that determines where to separate
  - elements $< x$ have to go to the left
  - elements $> x$ have to go to the right
  - for elements $= x$ it doesn't matter… (in the following to the left)



pivot $= 23$

1. Increment l as long as $A[l] \leq \text{pivot}$
2. Decrement r as long as $A[r] > \text{pivot}$
3. Swap $A[l]$ and $A[r]$

QuickSort : Partitioning the Array

UNI FREIBURG

- We have to partition such that

$$\text{Elements in left part} \leq \text{Elements in right part}$$

- **Idea:** Choose a **pivot x** that determines where to separate
  - elements $< x$ have to go to the left
  - elements $> x$ have to go to the right
  - for elements $= x$ it doesn't matter… (in the following to the left)

- **Algorithmus for divide** (often also called partition)**:**
  - Idea: Iterate from left and from right over array
  - If encounterinig an element that is on the correct side, one does not need to do anything.
  - If encountering an element that has to switch to the other side, the element can be swapped with an element on the other side that has to switch to the other side.

QuickSort : Partitioning the Array

Algorithmus for partitioning (somewhat more formally)**:**

- Task: Divide array A of length n according to pivot x
  - Assumption: Elements $\leq x$ go to the left, elements $> x$ go to the right

- General Procedure:
  - Two variables l and r to iterate over the array from the left and the right
  - Increment l until $A[l] > x$ (element has to go to the right)
  - Dekrementiere r bis $A[r] \leq x$ (element has to go to the left)
  - Swap $A[l]$ and $A[r]$, increment l, decrement r ($l \mathrel{+}= 1$, $r \mathrel{-}= 1$)
  - Divide is done as soon as l und r meet

- You will need to figure out the details in the exercises…

QuickSort : Choice of Pivot

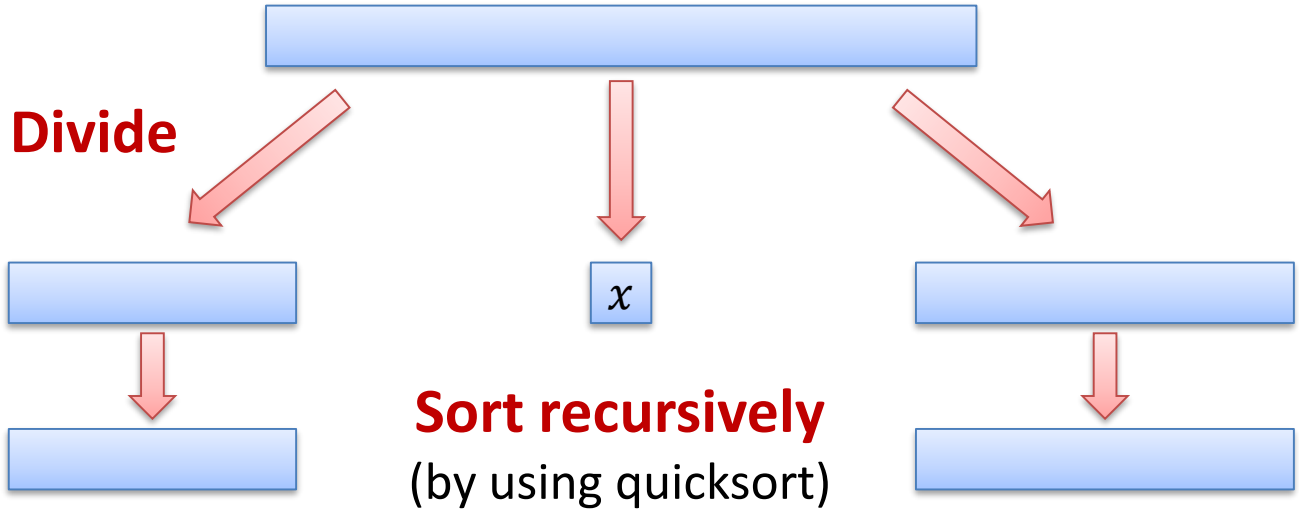
- Choice of the pivot determines how large the two parts become into which the array is divided…
- We will see: The algorithm is fastest, if the sizes of the two parts are as equal as possible

Strategies to determine the pivot:

- Median would be ideal → cannot be found easily…
  - We will see this later...
- A fixed element of the array (e.g., always the first of current part)
  - can lead to a very uneven partition...
- An element at a random position (inside the current part)
  - randomized QuickSort usually refers to exactly this strategy
  - Usually works fairly well and gives roughly equal parts
- Median of three (or more) random elements
  - somewhat more “expensive”, but somewhat more “equal” parts

Divide and Conquer

Overview QuickSort:



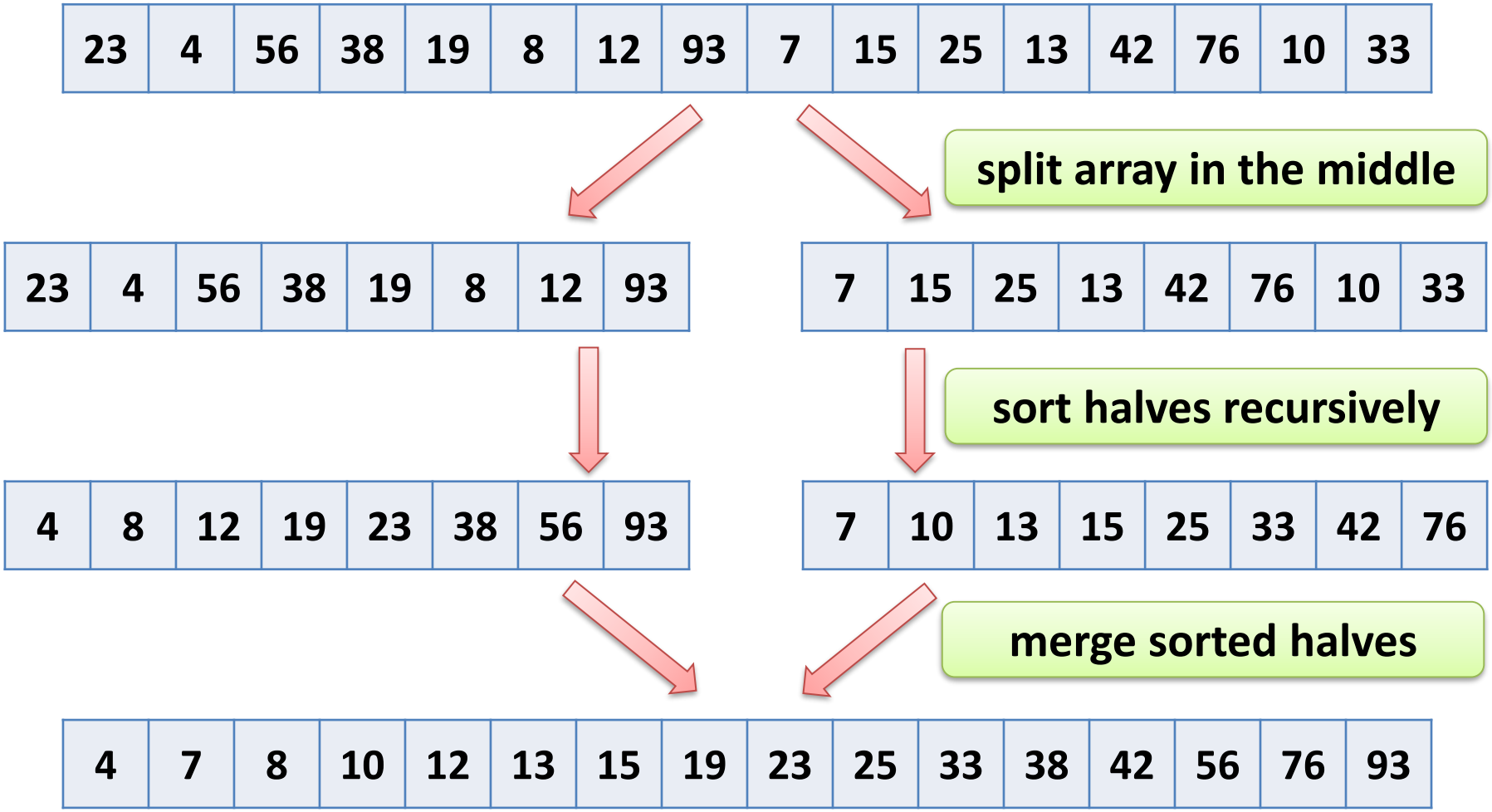
Divide and Conquer:

- Widely used algorithm design principle:

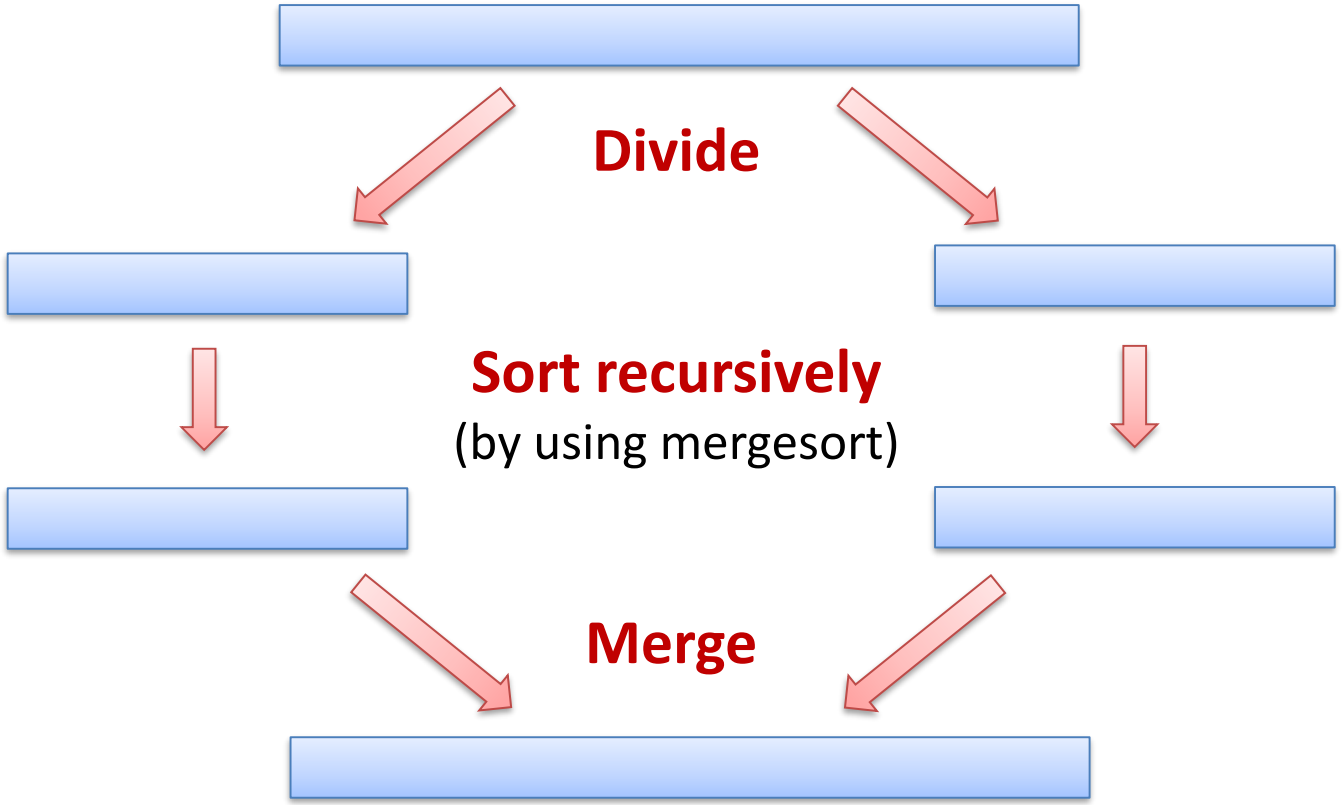
1. Divide input into 2 or more smaller sub problems
2. Solve sub problems recursively
3. Combine solutions of sub problems to solution of original problem.

MergeSort

- Another sorting algorithm that is based on the divide-and-conquer principle.



Overview MergeSort



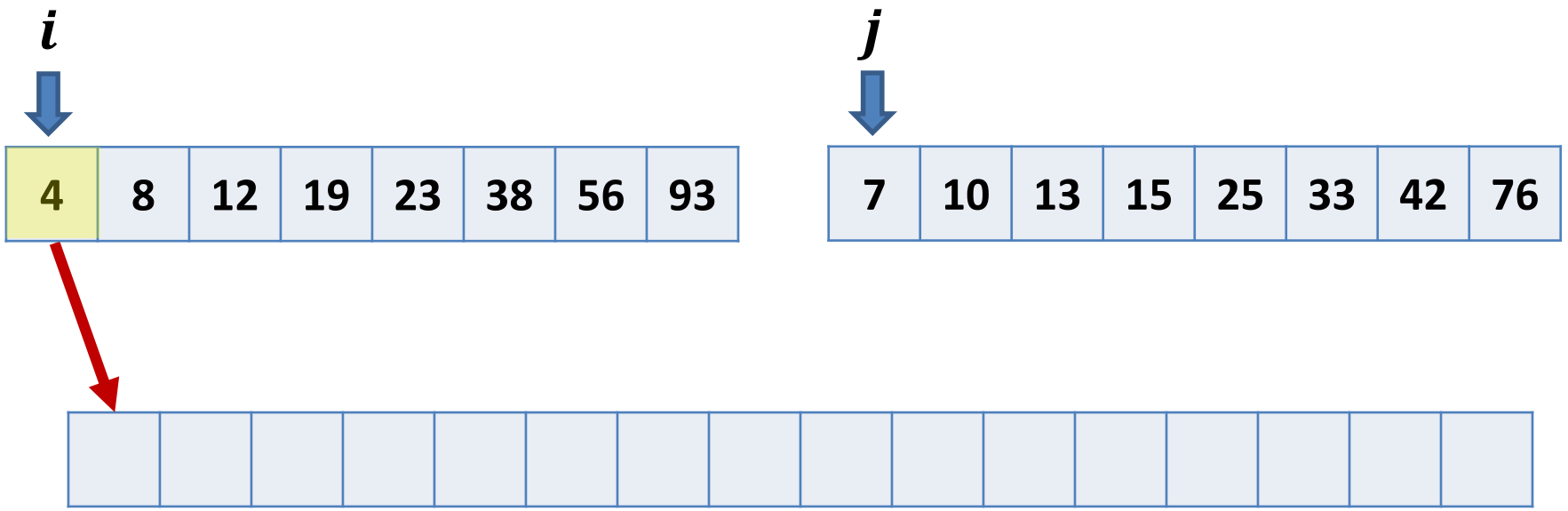
- Divide trivial for MergeSort
- Merge (combining the solutions) requires more work…

MergeSort: Merge Step

Merging of two sorted arrays:

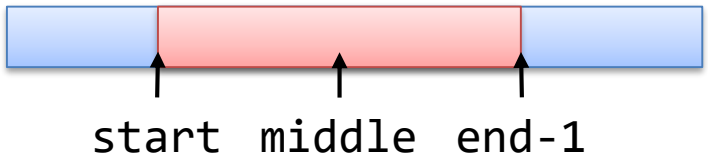
- Given: sorted arrays A und B of lengths n and m
- Output: sorted array C containing the elements of A and B

Merging Algorithm:



MergeSort: Pseudocode

Eingabe: Array A of size 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]
```

Summary Sorting Algorithms

UNI FREIBURG

- We have seen 4 different sorting algorithms:

Simple Sorting Algorithms: Selection Sort and Insertion Sort

- Selection Sort seems to be quite slow. Our measurements suggest that the running time is quadratic in the size n of the array
- We will see that both algorithms indeed have quadratic run time.

Recursive Sorting Algorithms: QuickSort and MergeSort

- Both algorithm use the divide-and-conquer design principle: The array of size n is split into two smaller sub-arrays that are afterwards solve recursively. The recursive solutions are then combined to a solution of the original sorting problem.
- We will see that the added complexity of the two algorithms pays off. Both algorithms are much faster than the two simple ones.