Informatik II
Prüfungsvorbereitungskurs

Tag 3, 8.6.2017

Giuseppe Accaputo

g@accaputo.ch

Aufbau des PVK

- **Tag 1**: Java Teil 1
- **Tag 2**: Java Teil 2
- **Tag 3**: Algorithmen & Komplexität
- **Tag 4**: Dynamische Datenstrukturen, Datenbanksysteme

Programm für heute

- Repetition
- Rekursion
- Komplexität
- Algorithmen

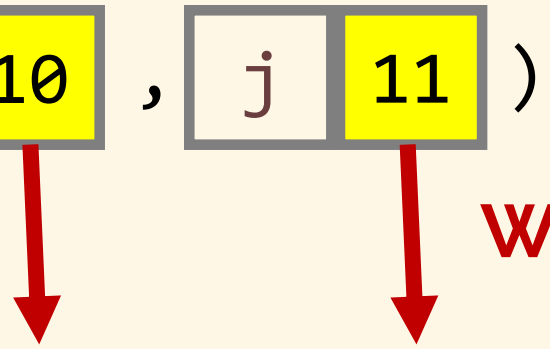
Repetition Tag 2

Pass by Value

- Argumentwerte einer Methode werden beim Methodenaufruf in die Parameter kopiert
- Primitive Datentypen:
  Wert wird in Parameter kopiert
- Nicht-primitive Datentypen:
  Referenz wird in Parameter kopiert

Pass by Value: primitive Datentypen

```
void test(){
      int i = 10, j = 11;

      do( i  10 , j  11 );

}

void do( a  10 , b  11 )
      …
}
```



Werte werden kopiert

- Funktion **do** hat zwei Parameter mit primitiven Datentypen

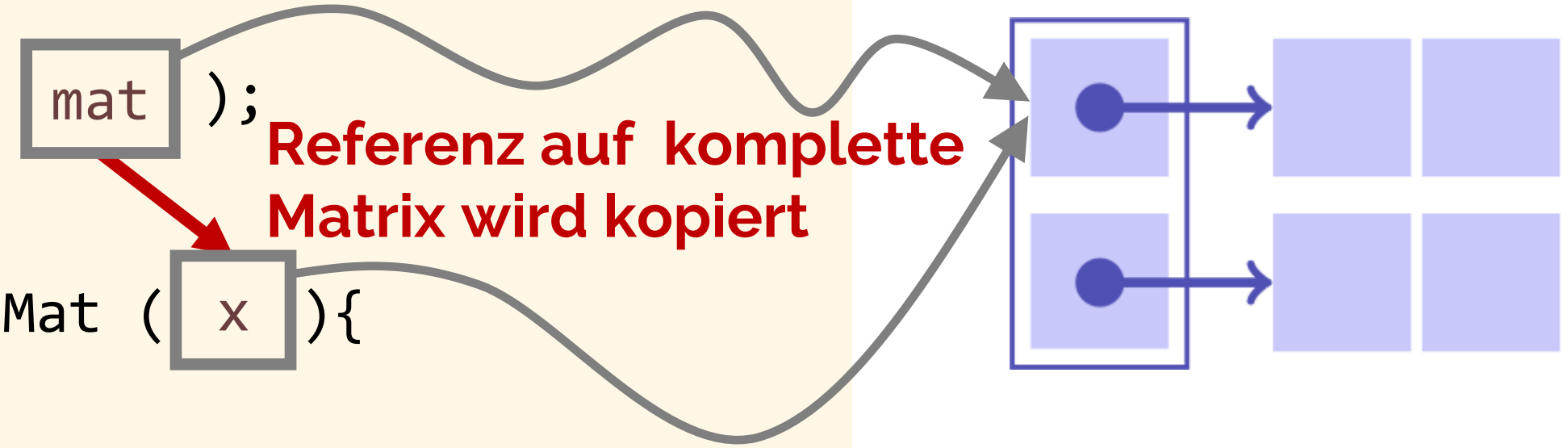
Pass by Value: nicht-primitive Datentypen

```
void test(){
      int[] arr = new int[2];
      arr[0] = 10;arr[1] = 11;

      fn( arr );

}

void fn( x ){
…}
```

- Funktion `fn` hat ein Parameter mit nicht-primitivem Datentyp

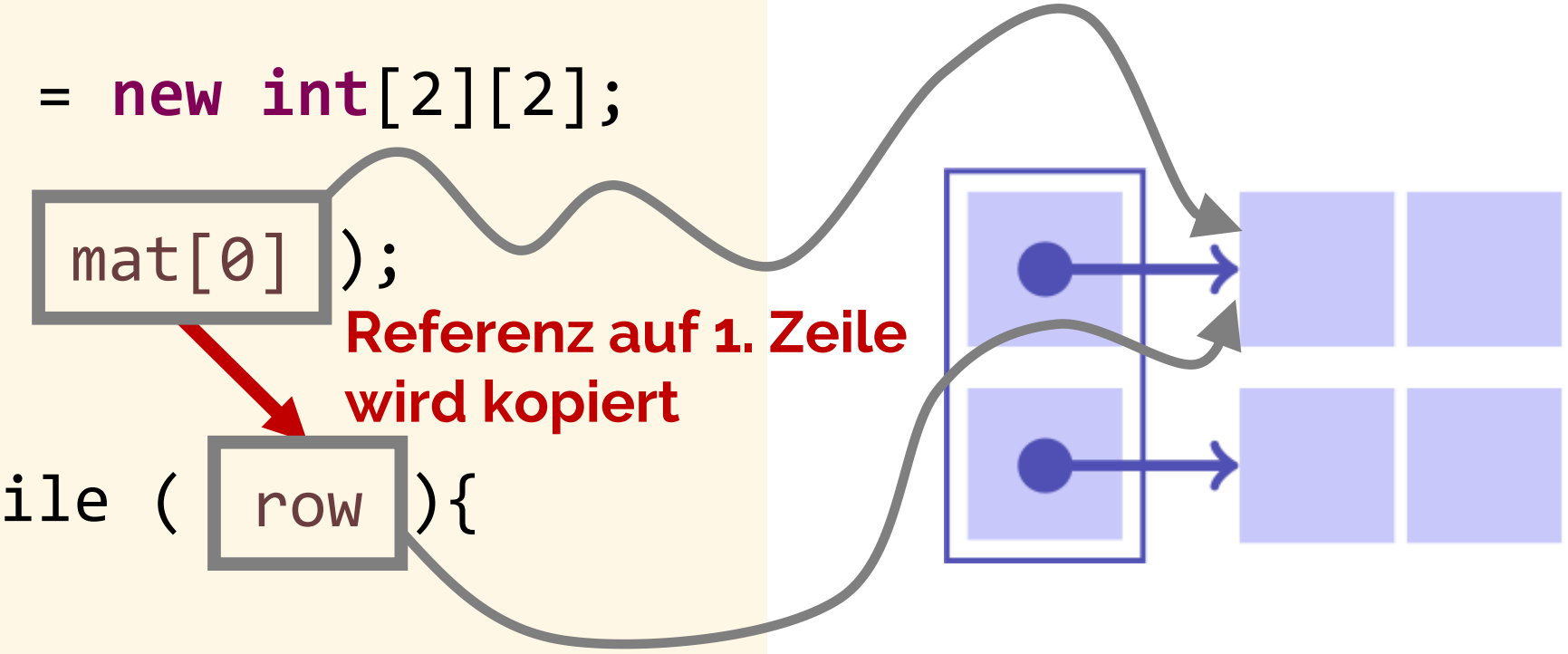
Pass by Value: Mehrdimensionales Array

```
void test(){
  int[][] mat = new int[2][2];
    …
  fnMat( mat );
}

void fnMat ( x ){
    …
}
```



Pass by Value: Mehrdimensionales Array

```
void test(){
 int[][] mat = new int[2][2];
  …
 fnMatZeile( mat[0] );
}

void fnMatZeile ( row ){
  …
}
```

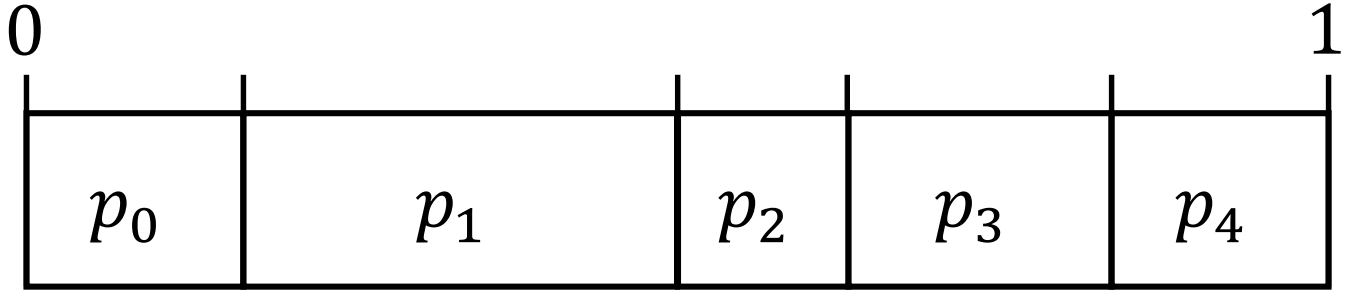


MCMC Simulationen

- Seien n mögliche Zustände gegeben (z.B. Wetterlage, Schrittlängen die eine Person machen kann, Zimmer in welchem sich die Maus befindet, etc.)
- Sei ein Wahrscheinlichkeitsvektor oder eine Wahrscheinlichkeitsmatrix gegeben
- Verwende den W-keitsvektor oder die W-keitsmatrix um zu bestimmen, welches der nächste Schritt oder Zustand ist

Nächsten Schritt bestimmen («unfairer Würfel»)

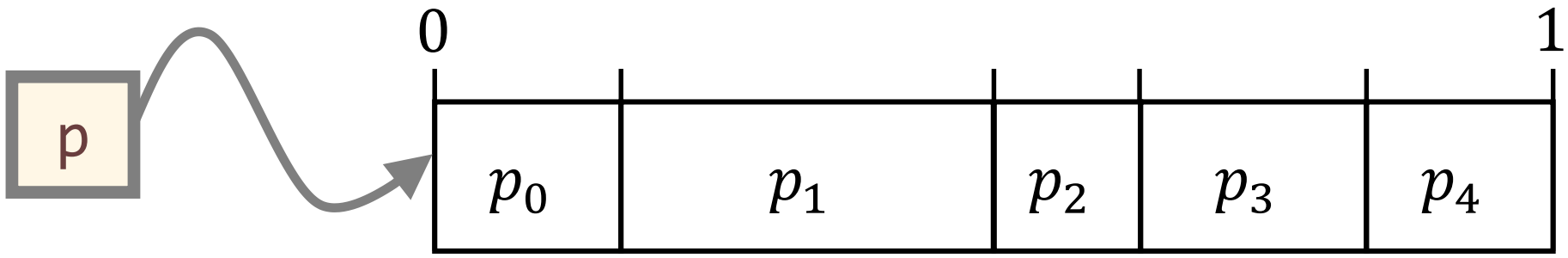
- **Unfair**: Zahlen können mit verschiedenen Wahrscheinlichkeiten gewürfelt werden
- **Gegeben**: W-keitsvektor $p = (p_0, p_1, \dots, p_{n-1})$ mit $\sum_{i=0}^{n-1} p_i = 1$



- **Gesucht**: `sample(p)` soll ein $j \in [0, n-1]$ zurückgeben mit Wahrscheinlichkeit p_j

Nächsten Schritt bestimmen («unfairer Würfel»)

```
int sample(double p[]){
   …
}
```



Wahrscheinlichkeitsmatrix

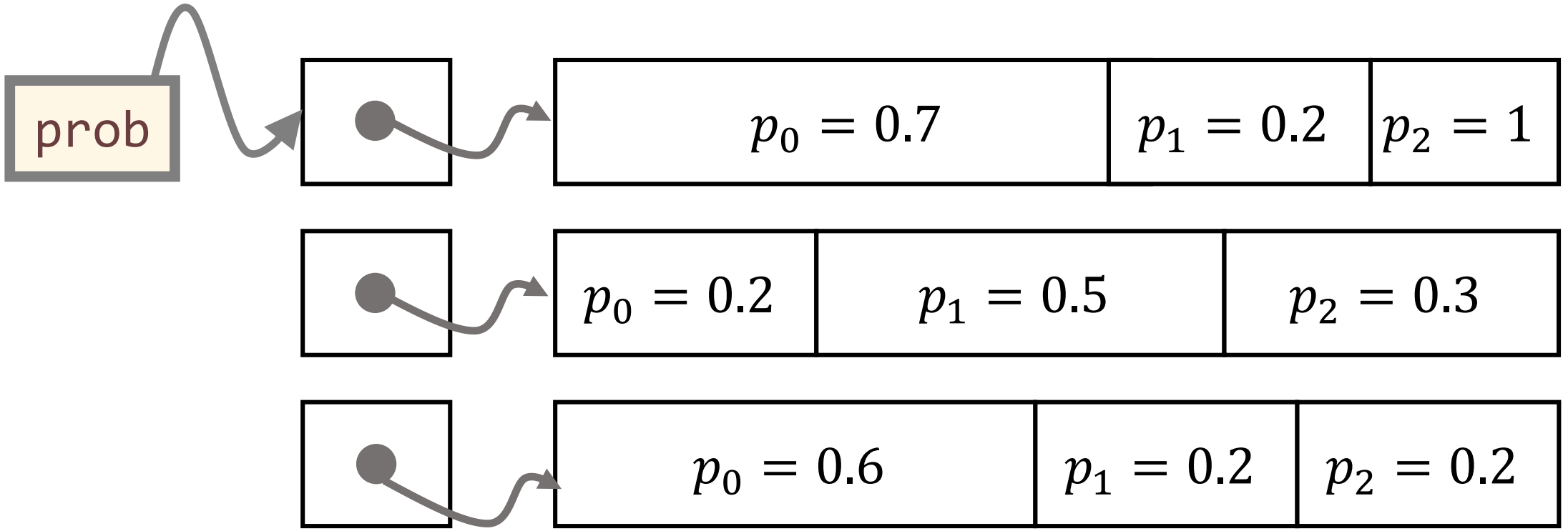
- Modell: Wandtafel

	Sonne	Wolken	Regen
Sonne	0.7	0.2	0.1
Wolken	0.2	0.5	0.3
Regen	0.6	0.2	0.2

Quelle: Informatik II Vorlesung

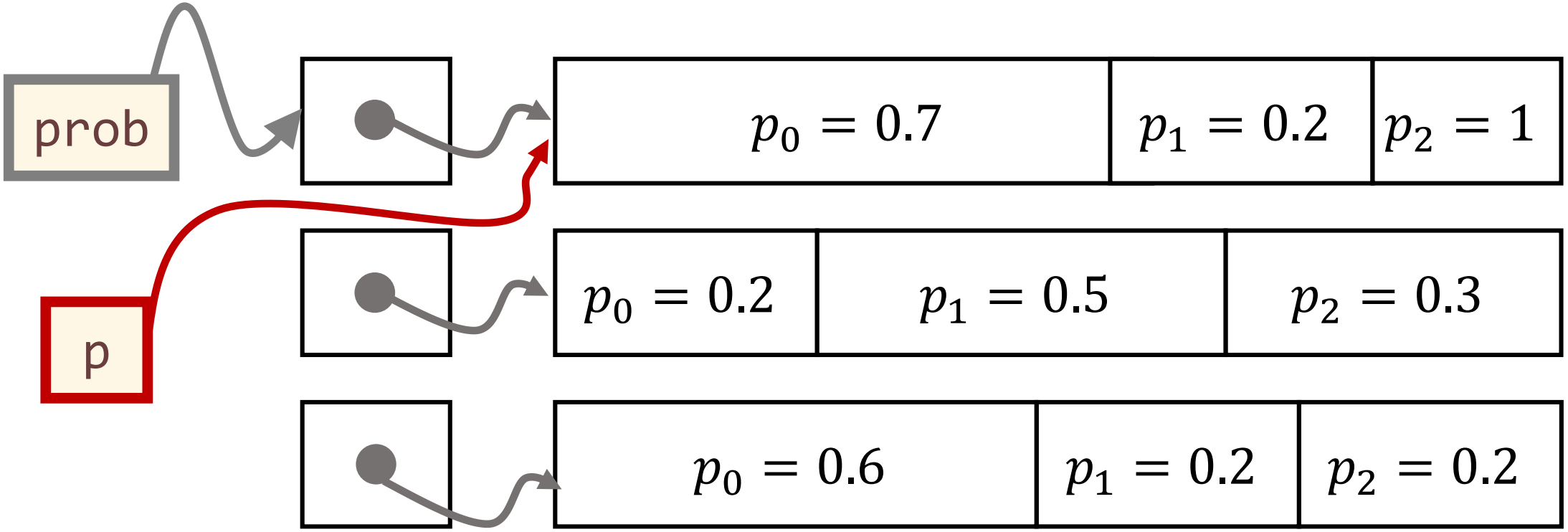
Wahrscheinlichkeitsmatrix

```
double[][] prob = new double[3][3];
```



```
int sample(double p[]){
   …
}
```

```
int ind = sample(prob[0]);
```



Prüfung 08.2016 Aufgabe 2

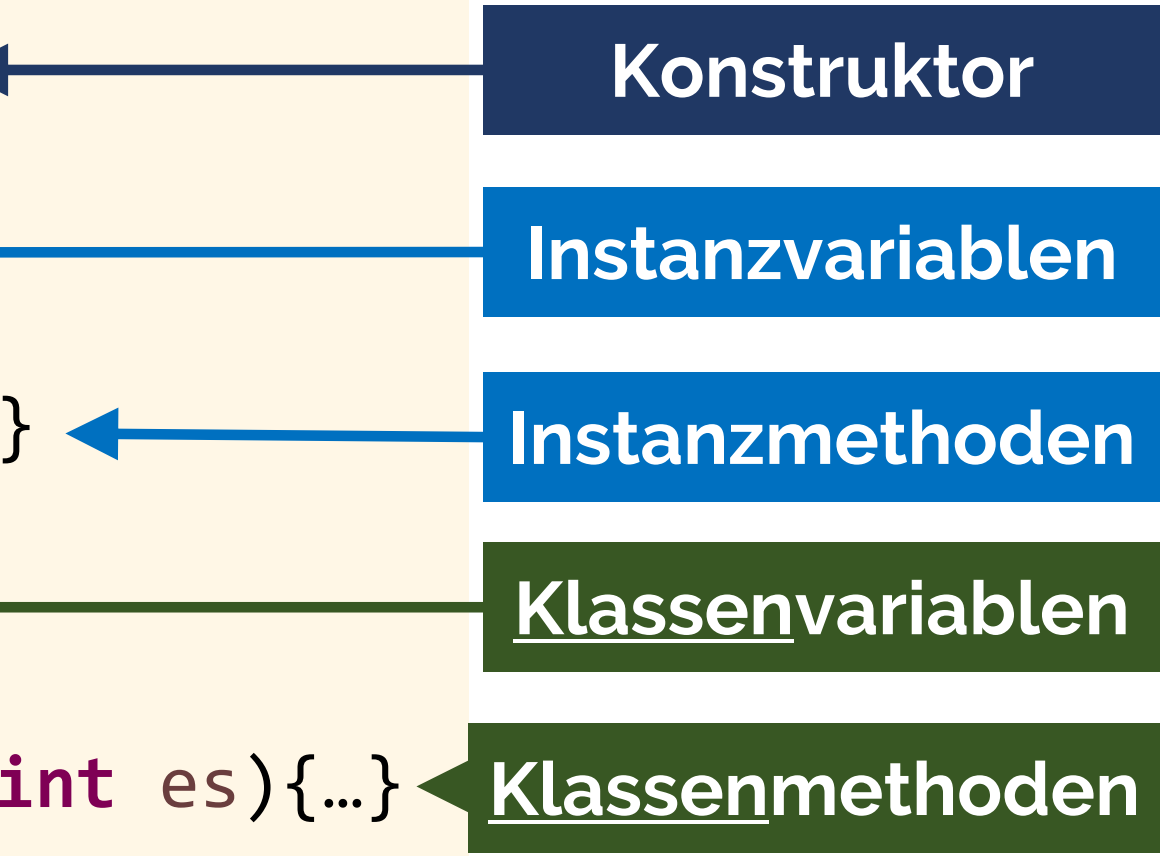
- Wettersimulation
- Wandtafel

Prüfung 08.2014 Aufgabe 10

- Random Walk
- Wandtafel

Aufbau einer Klasse

```
public class MeineKlasse {
    public MeineKlasse(){…}

    private int i;

    public int setI(int i){…}

    public static int s;

    public static void setS(int es){…}
}
```



Instanzen einer Klasse

```
MeineKlasse m1 = new MeineKlasse();
MeineKlasse m2 = new MeineKlasse(10);
```

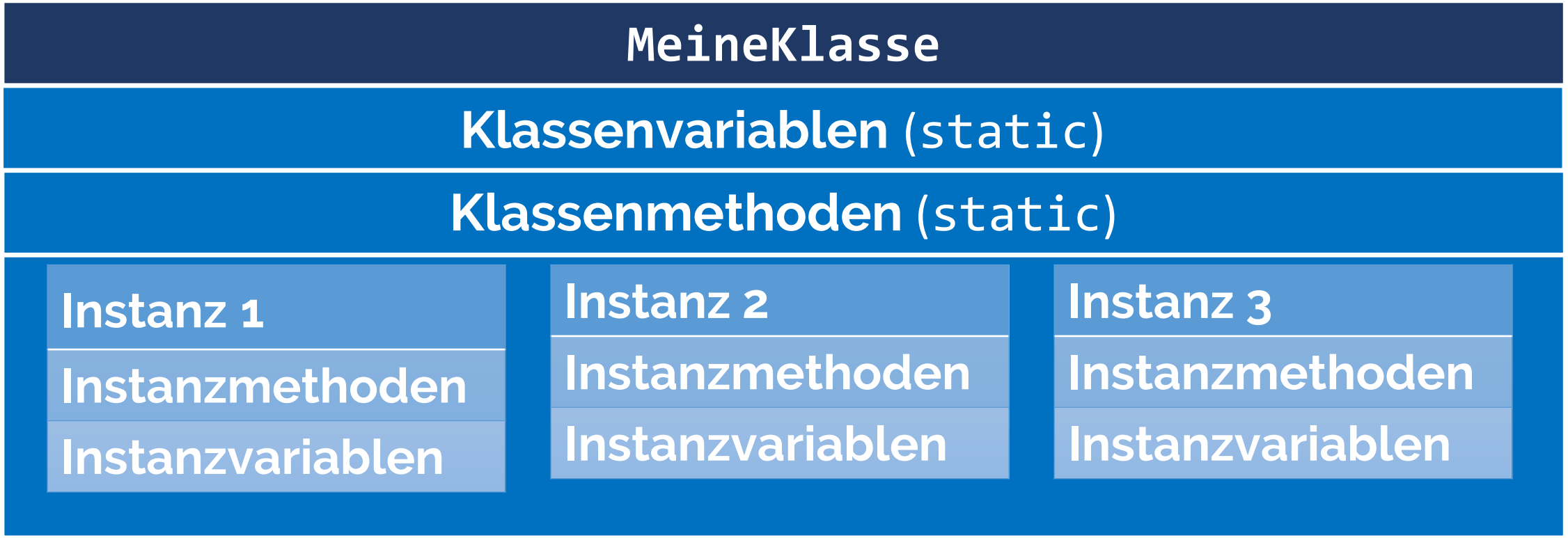
Instanz **m1**
Instanzmethoden Instanzvariablen

Instanz **m2**
Instanzmethoden Instanzvariablen

- Jede Instanz hat eigene Kopie von Instanzmethoden und -variablen
- Ändert Instanz Wert der eigenen Instanzvariablen, so ist die Änderung nur für die Instanz geltend

Klassenvariablen und Klassenmethoden

- Klassenvariablen und Klassenmethoden sind über alle Instanzen verfügbar (sofern Zugriff gewährleistet)



Prüfung 08.2015 Aufgaben 1a & 1c

- Java
- Wandtafel

Rekursion

Rekursion

- Eine Prozedur, die sich selber aufruft, heisst rekursiv.

- “In order to understand recursion, one must first understand recursion”

Rekursion

Jede rekursive Funktion besteht aus den folgenden 3 Bestandteilen:

1. Abbruchbedingung: bestimmt wann die Funktion fertig ist
2. Schritt in Richtung Erfüllung der Abbruchbedingung: wir vereinfachen das Problem, bzw. nähern uns der Terminierung (wie bei `while`-Schleifen)
3. Rekursiver Aufruf: die Funktion sollte sich selbst aufrufen

Beispiel Rekursion

- Rekursive Definition der Fakultät:

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

Beispiel Rekursion

- Rekursive Funktion um Fakultät einer Zahl **n** zu berechnen:

```
public static double fklt(double n){
     if(n <= 1)
          return 1;
     else
          return n * fklt(n-1);
}
```

Beispiel Rekursion

1. Abbruchbedingung:

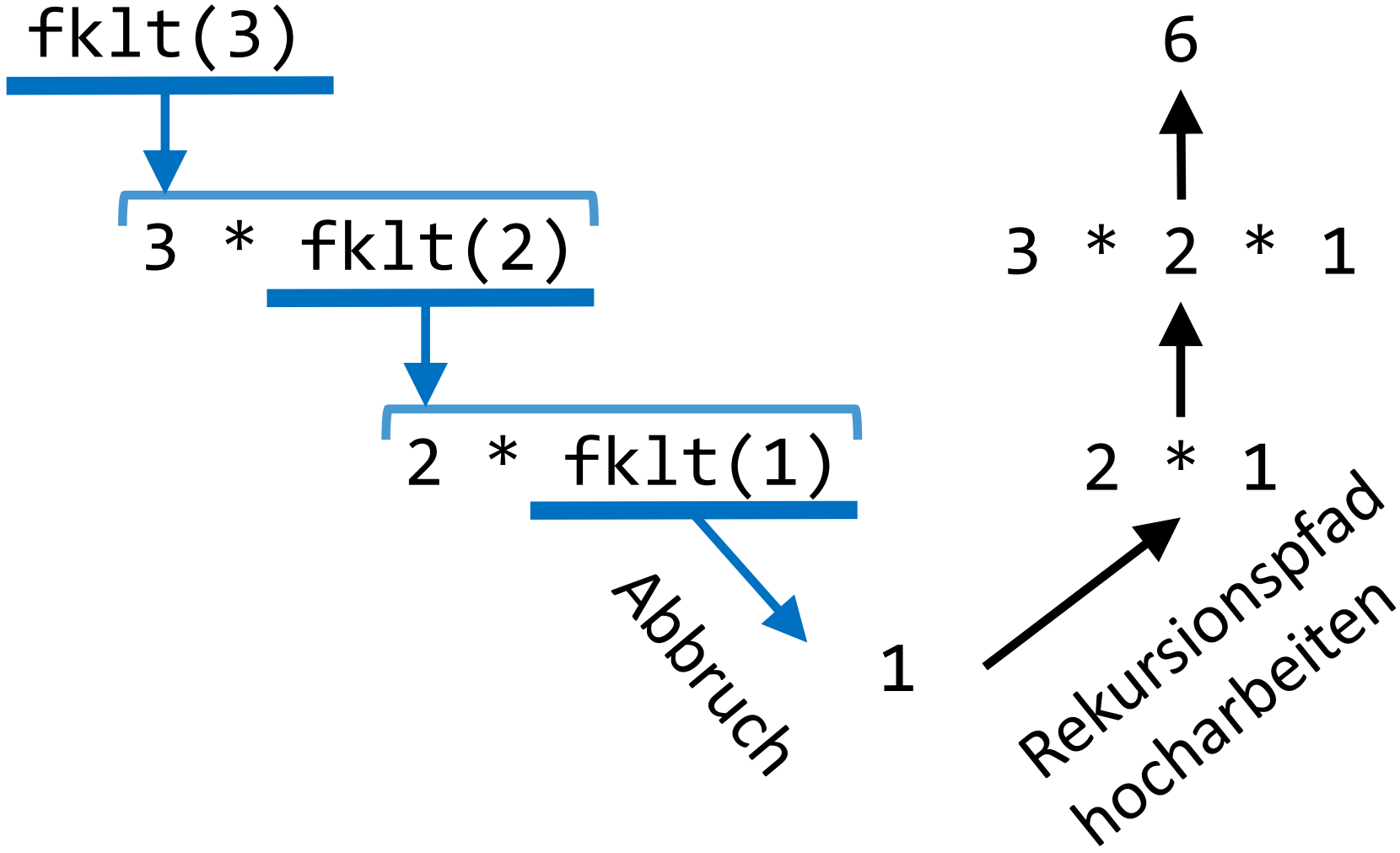
```
if(n <= 1) return 1;
```

   - Hier wird ein konkreter Wert zurückgegeben (falls rekursive Funktion einen Rückgabetypen definiert hat)

2. Rekursiver Aufruf und Fortschritt Richtung Abbruch/Terminierung:

```
return n * fklt(n-1);
```

Beispiel Rekursion



Aufgabe Rekursion

- Terminieren die beiden folgenden Funktionen?
- Wie sieht ein Beispielaufruf aus?

(a)
```
boolean f (const int n) {
    if (n == 0) {
        return false;
    }
    return !f(n-1);
}
```

(b)
```
void g (int n) {
    if (n == 0) {
        System.out.printf("*");
        return;
    }
    g(n-1);
    g(n-1);
}
```

Self-Assessment Test 2 Aufgaben 1a & 4

- Rekursion
- Wandtafel

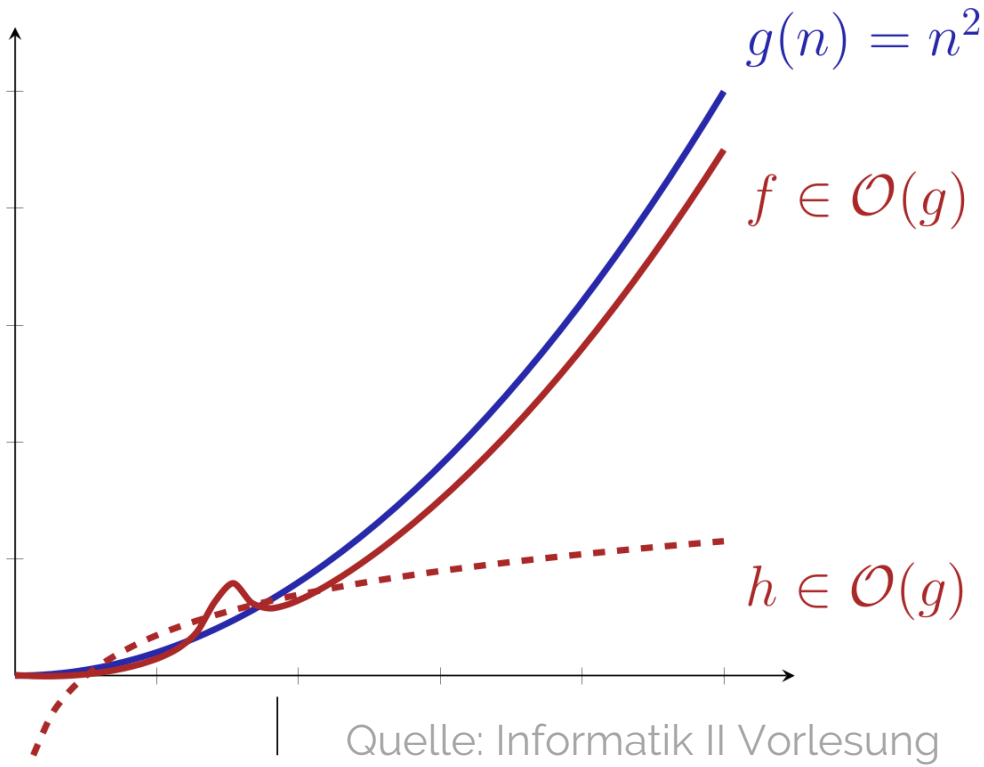
Komplexität

Asymptotische Komplexität

- Effizienz von Algorithmen bestimmen
- Zeitlicher Aufwand in Abhängigkeit der Problemgrösse n
  - Z.B. n = Grösse einer Eingabeliste

Big-O Notation

- Big-O Notation: obere Schranke für die Laufzeit eines Algorithmus (*worst case*)
- Ignoriert konstante Faktoren
  - Beispiel: $3n \in O(n)$, d.h. $3n$ wächst höchstens so schnell wie n



Quelle: Informatik II Vorlesung

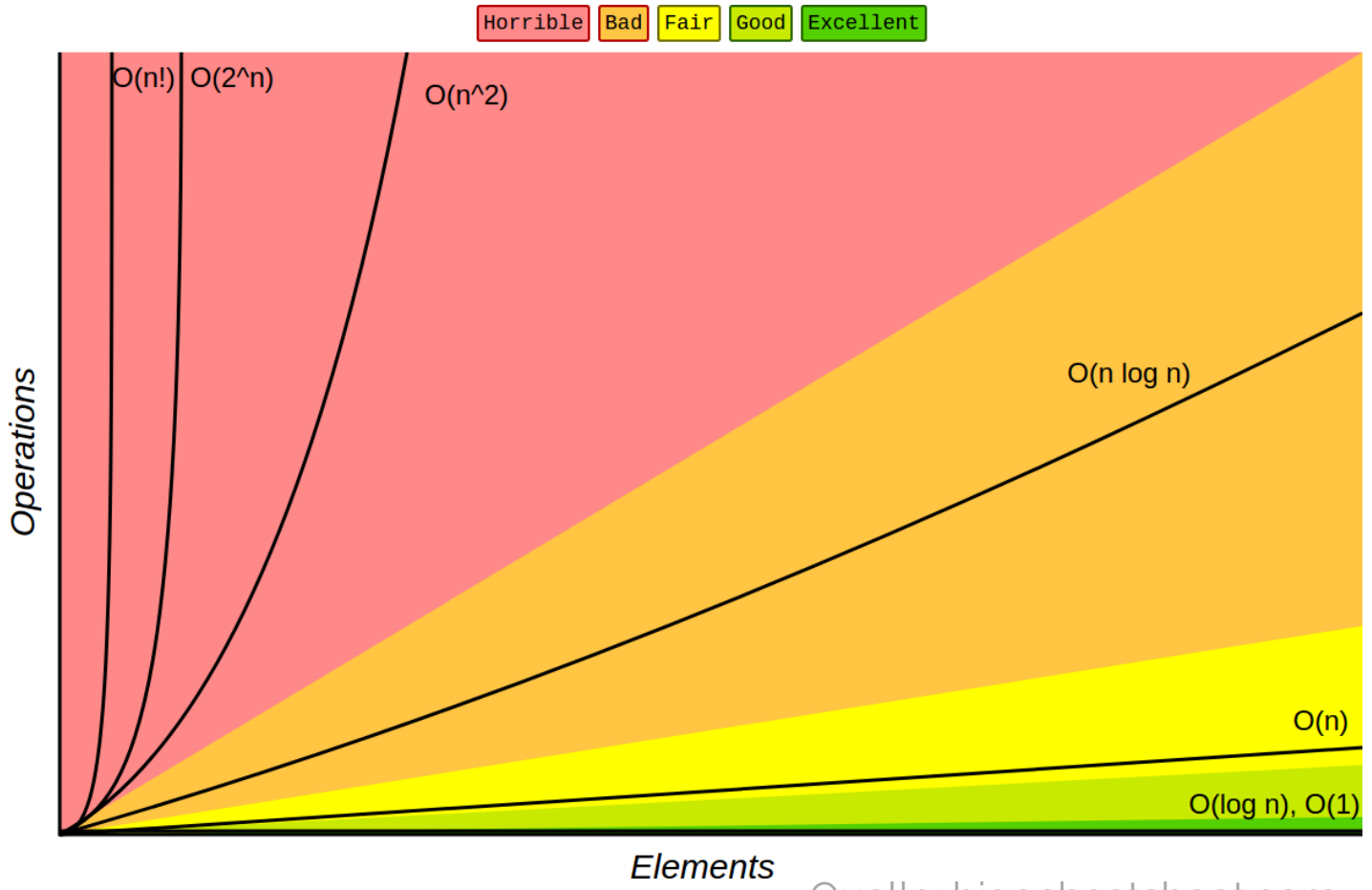
$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \; \forall n \geq n_0\}$$

Asymptotische Komplexität

$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log \log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n \log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(2^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

Quelle: Informatik II Vorlesung

Asymptotische Komplexität



Quelle: bigocheatsheet.com

Aufgabe Asymptotische Komplexität

$f(n)$	$f \in \mathcal{O}(?)$
$3n^2 + 5$	
$7n$	
$3n + 2$	
$log_2(n) + 5$	
$n * n$	
$(n * n + 1) * n * n/2$	

Quelle: Informatik II Vorlesung

Aufgabe Asymptotische Komplexität

Sort the following functions from left to right such that: if function f is left to function g, then $f \in \mathcal{O}(g)$. Example: n^3, n^7, n^9 are in a correct order ($n^3 \in \mathcal{O}(n^7)$, $n^7 \in \mathcal{O}(n^9)$).

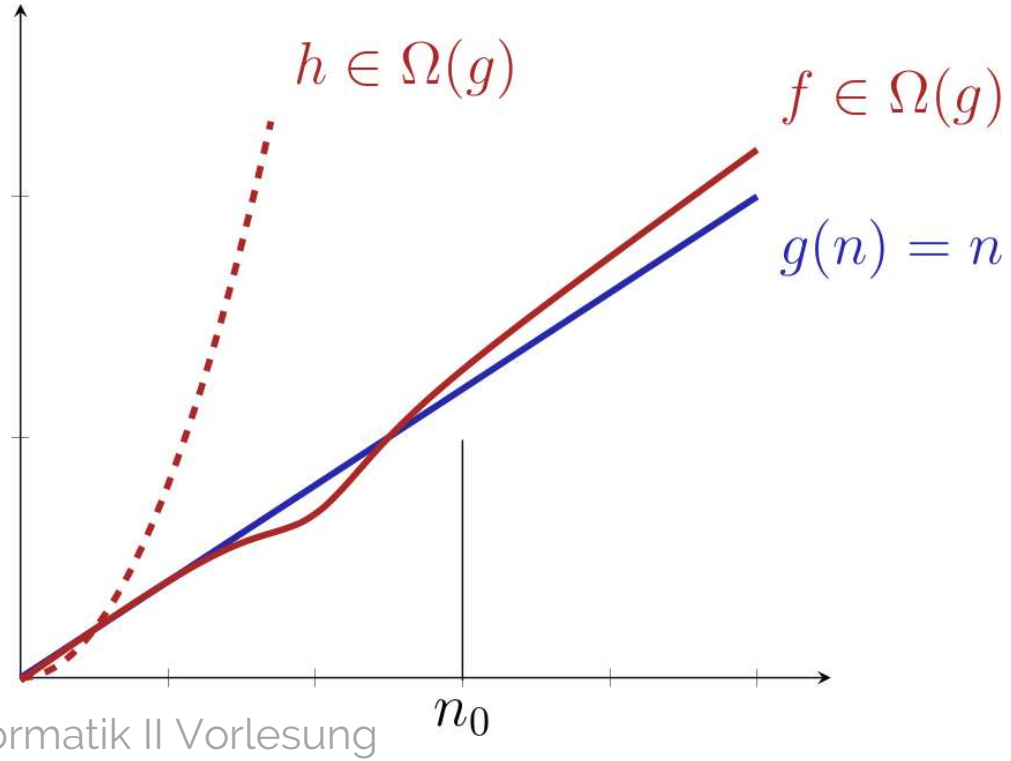
$$n^5 + n,\ \log(n^4),\ \sqrt{n},\ \binom{n}{3},\ 2^{16},\ n^n,\ n!,\ \frac{2^n}{n^2},\ \log^8(n),\ n \log n.$$

Quelle: Informatik II Vorlesung

Big-Omega

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \; \forall n \geq n_0\}$$

- Untere Schranke für die Laufzeit eines Algorithmus

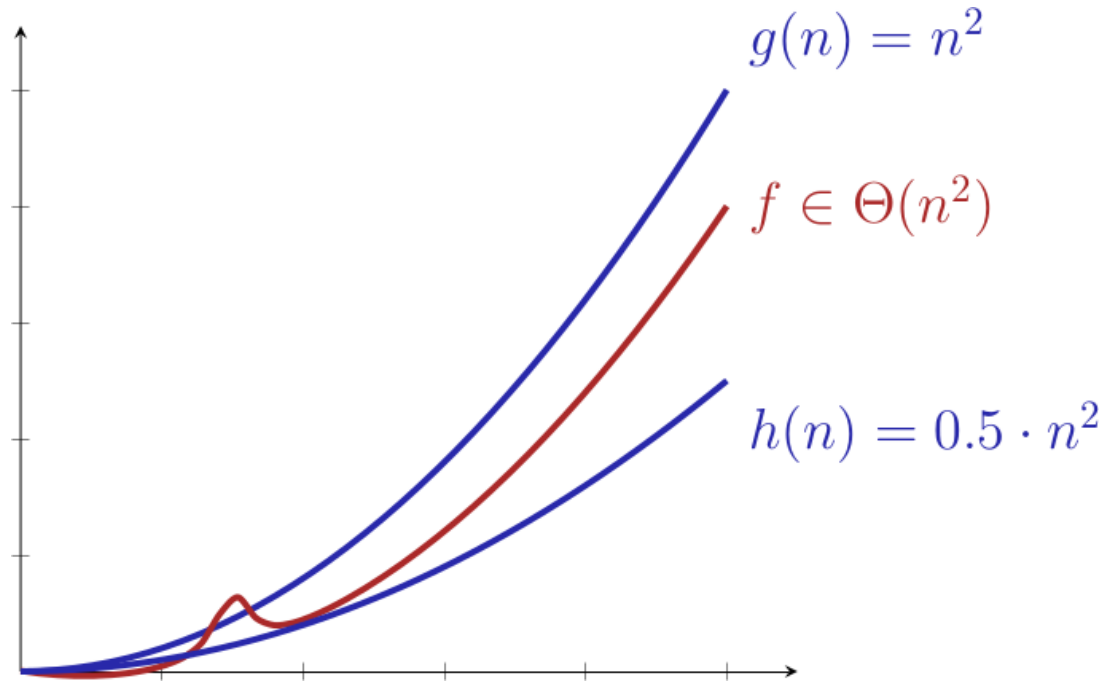


Quelle: Informatik II Vorlesung

Big-Theta

- Asymptotisch scharfe Schranke

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



Quelle: Informatik II Vorlesung

Beispiel Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f2(int n){
     int res = 0;
     for(int i = 0; i<n; i++)
          res += i;
}
```

Beispiel Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f2(int n){
     int res = 0;
     for(int i = 0; i<n; i++)
          res += i;
}
```

$$O(n)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f1(int a[]){
     if(a.length != 0)
          a[0] = 1;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f1(int a[]){
     if(a.length != 0)
          a[0] = 1;
}
```

$$O(1)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f3(int n){
   int res = 0;
   for(int i = 0; i<n; i++)
      for(int j = 0; j<n; j++)
            res += j*i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f3(int n){
   int res = 0;
   for(int i = 0; i<n; i++)
      for(int j = 0; j<n; j++)
            res += j*i;
}
```

$$O(n^2)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f4(int n){
   int res = 0;
   for(int i = 0; i<n; i++)
          res += i;
   for(int j = 0; j<n; j++)
          res += j;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f4(int n){
   int res = 0;
   for(int i = 0; i<n; i++)
          res += i;
   for(int j = 0; j<n; j++)
          res += j;
}
```

$$O(2n) \Longrightarrow O(n)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f5(int n){
   int res = 0;
   for(int i = 0; i<n*n; i++)
            res += i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f5(int n){
   int res = 0;
   for(int i = 0; i<n*n; i++)
            res += i;
}
```

$$O(n^2)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f6(int n){
   int res = 0;
   for(int i = 1; i<n; i*=2)
            res += i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f6(int n){
   int res = 0;
   for(int i = 1; i<n; i*=2)
            res += i;
}
```

$$O(\log(n))$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f7(int n){
   int res = 0;
   for(int i = 0; i<n; i+=2)
            res += i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f7(int n){
   int res = 0;
   for(int i = 0; i<n; i+=2)
            res += i;
}
```

$$O(n/2) \Longrightarrow O(n)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f8(int n){
   int res = 0;
   for(int i = 0; i<n; i++)
      for(int j = i+1; j<n; j++)
            res += j*i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f8(int n){
   int res = 0;
   for(int i = 0; i<n; i++)
      for(int j = i+1; j<n; j++)
            res += j*i;
}
```

$$O(n^2)$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f9(int n){
   int res = 0;
   for(int i = 1; i<n; i*=2)
      for(int j = 0; j<n; j++)
            res += j*i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f9(int n){
   int res = 0;
   for(int i = 1; i<n; i*=2)
      for(int j = 0; j<n; j++)
            res += j*i;
}
```

$$O(n\log(n))$$

Aufgabe Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f10(int n){
     int res = 0;
     for(int i = 2; i<n; i*=i)
          res += i;
}
```

Lösung Asymptotische Komplexität

- Was ist die Komplexität der folgenden Funktion?

```
public static void f10(int n){
     int res = 0;
     for(int i = 2; i<n; i*=i)
          res += i;
}
```

$$O(\sqrt{n})$$

Self-Assessment Test 2 Aufgabe 3

- Asymptotische Komplexität
- Wandtafel

Zeitkomplexität

- **best-case-Laufzeit**:
  wie lange der Algorithmus mindestens braucht
- **worst-case-Laufzeit**:
  wie lange der Algorithmus maximal braucht

Suchen

Suche in unsortiertem Array

- **Gegeben**:
  1. Unsortiertes Array A mit n Elementen $(A[1], \ldots, A[n])$
  2. Schlüssel b
- **Gesucht**:

  Index $k, 1 \leq k \leq n$ mit $A[k] == b$ oder «nicht gefunden»

Lineare Suche in unsortiertem Array

- **Algorithmus**: Iteriere durch das Array und vergleiche jedes Element $A[i]$ mit dem Schlüssel b
- **best-case-Laufzeit**: $\boldsymbol{O(1)}$
  (Zu suchendes Element befindet sich an der ersten Position)
- **worst-case-Laufzeit**: $\boldsymbol{O(n)}$
  (Zu suchendes Element befindet sich an der letzten Position. Wir müssen durch alle n Elemente iterieren)

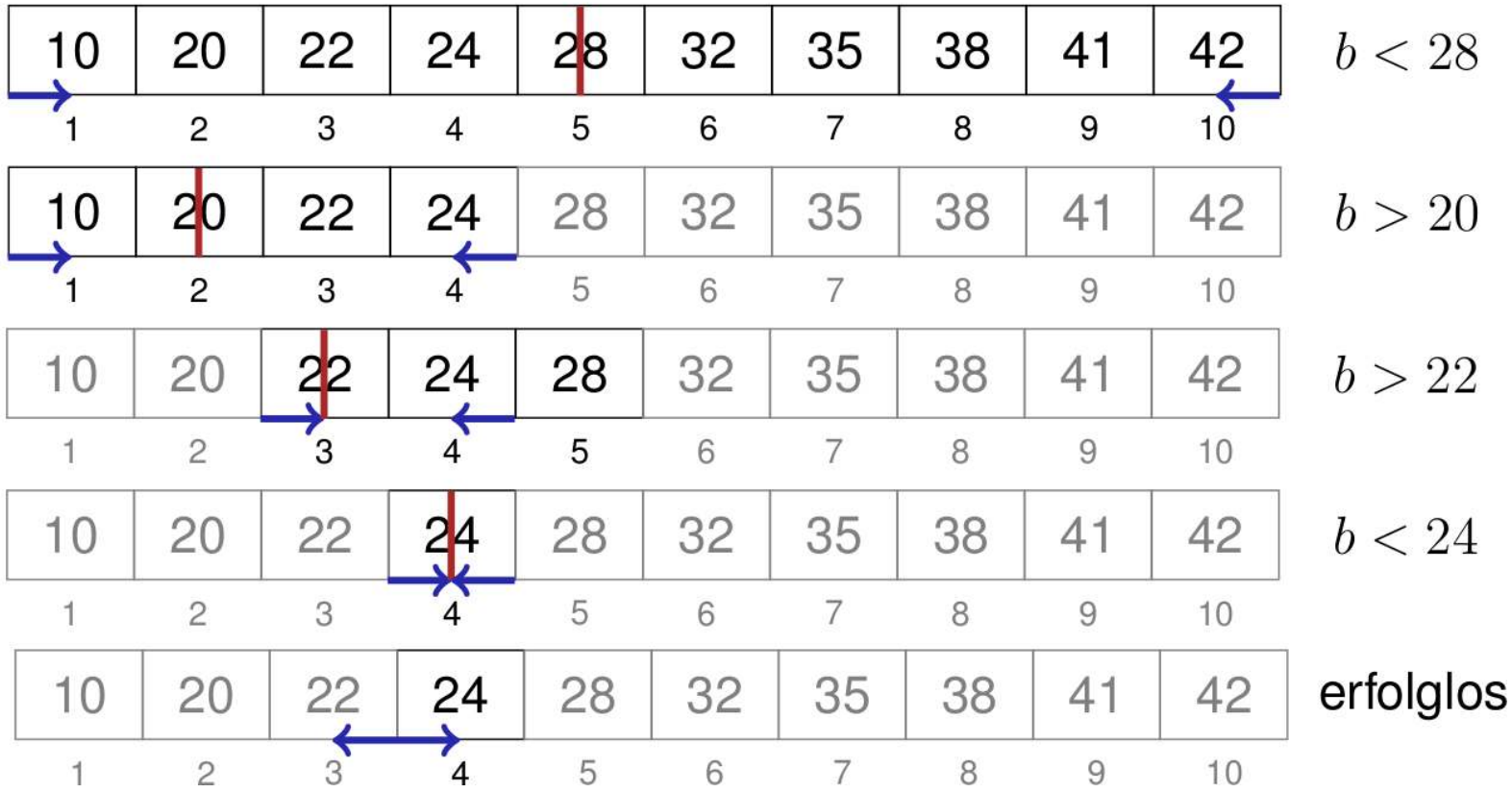
Suche in sortiertem Array

- **Gegeben**:
  1. Sortiertes Array A mit n Elementen $(A[1], \ldots, A[n])$ und $A[1] \leq A[2] \leq \ \ldots \leq A[n]$
  2. Schlüssel b
- **Gesucht**:

  Index $k, 1 \leq k \leq n$ mit $A[k] == b$ oder «nicht gefunden»

Divide and Conquer

Suche $b = 23$.



Quelle: Informatik II Vorlesung

Binäre Suche

Input : Sortiertes Array A von n Schlüsseln. Schlüssel b. Bereichsgrenzen $1 \leq l \leq r \leq n$ oder $l > r$ beliebig.
Output : Index des gefundenen Elements. 0, wenn erfolglos.

$m \leftarrow \lfloor (l + r)/2 \rfloor$
if $l > r$ **then** // erfolglose Suche
    return 0
else if $b = A[m]$ **then**// gefunden
    return m
else if $b < A[m]$ **then**// Element liegt links
    return BSearch$(A, b, l, m - 1)$
else // $b > A[m]$: Element liegt rechts
    return BSearch$(A, b, m + 1, r)$

Quelle: Informatik II Vorlesung

Binäre Suche

- **Algorithmus**: Iteriere durch das Array und vergleiche jedes Element $A[i]$ mit dem Schlüssel b
- **best-case-Laufzeit**: $\boldsymbol{O(1)}$
  (Element befindet sich gleich auf der Position m)
- **worst-case-Laufzeit**: $\boldsymbol{O(\log(n))}$
  (Wir müssen so lange die Menge halbieren, bis nur noch das Element alleine da steht)

Auswählen

Das Auswahlproblem

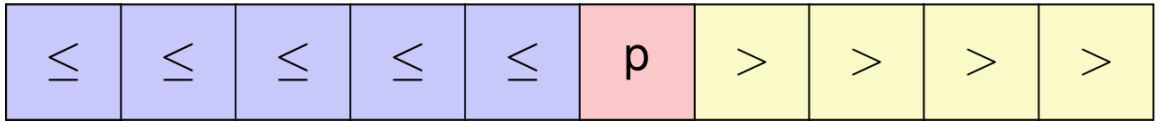
- **Gegeben**:
  1. Unsortiertes Array A mit n Elementen $(A[1], \ldots, A[n])$
  2. Zahl $1 \leq k \leq n$
- **Gesucht**:
  Element $A[i]$ mit $|\{j: A[j] < A[i]\}| = k - 1$, also das k-grösste Element im Array

Das Auswahlproblem

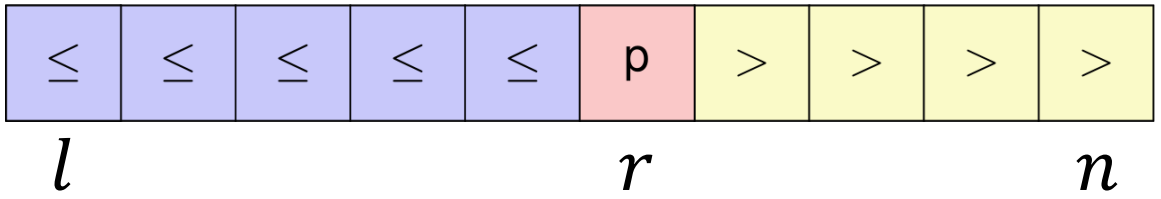
- $k = n$: Minimum
- $k = n$: Maximum
- $k = \lfloor n/2 \rfloor$: Median

Pivotieren

1. Wähle ein Element p als Pivotelement
2. Teile A in zwei Teile auf, so dass



3. Führe Rekursion auf relevantem Teil aus. Falls $k = r$ ist, haben wir das k-grösste Element gefunden



Algorithmus `Partition(A[l..r], p)`

Input : Array A, welches den Sentinel p im Intervall $[l, r]$ mindestens einmal enthält.
Output : Array A partitioniert in $[l..r]$ um p. Rückgabe der Position von p.

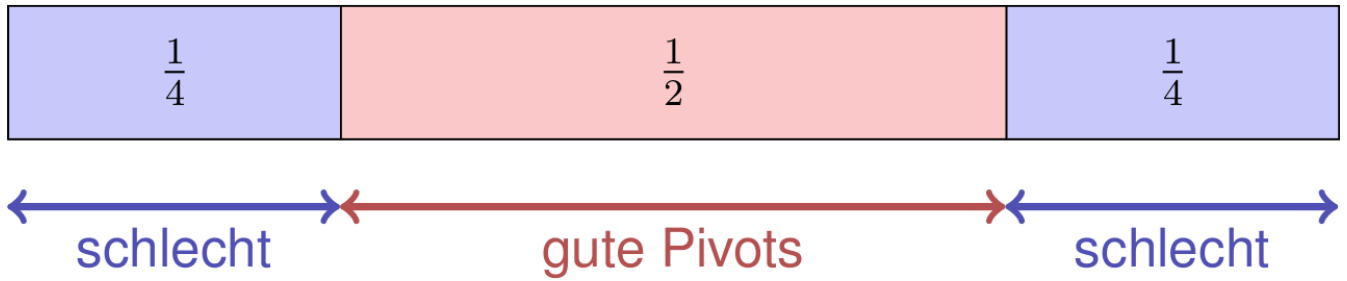
while $l < r$ **do**
    while $A[l] < p$ **do**
        $l \leftarrow l + 1$
    while $A[r] > p$ **do**
        $r \leftarrow r - 1$
    swap$(A[l], A[r])$
    if $A[l] = A[r]$ **then**
        $l \leftarrow l + 1$

return l-1

Quelle: Informatik II Vorlesung

Wahl des Pivots

- Schlechter Pivot: Kleinstes Element
  - **worst-case-Laufzeit**: $O(n^2)$
- Guter Pivot: Zufälligen Pivot wählen



Algorithmus `Quickselect(A[l..r], i)`

Input : Array A der Länge n. Indizes $1 \leq l \leq i \leq r \leq n$, so dass für alle $x \in A[l..r]$ gilt, dass $|\{j|A[j] \leq x\}| \geq l$ und $|\{j|A[j] \leq x\}| \leq r$.
Output : Partitioniertes Array A, so dass $|\{j|A[j] \leq A[i]\}| = i$

```
if l=r then return;
repeat
    wähle zufälligen Pivot x ∈ A[l..r]
    p ← l
    for j = l to r do
        if A[j] ≤ x then p ← p + 1
until (l+r)/4 ≤ p ≤ 3(l+r)/4
m ← Partition(A[l..r] ,p)
if i < m then
    quickselect(A[l..m], i)
else
    quickselect(A[m..r], i)
```

Quelle: Informatik II Vorlesung

Algorithmus `Quickselect(A[l..r], i)`

- **Algorithmus**: Unterteile Array in zwei Teile: Elemente die grösser oder kleiner als Pivot sind (Partitionieren). Wenn Pivot an k-ter Stelle steht, haben wir das k-grösste Element gefunden. Ansonsten: Rekursion in Teil welches das k -grösste Element enthalten kann.
- **best-case-Laufzeit**: $\boldsymbol{O(n)}$
- **worst-case-Laufzeit**: $\boldsymbol{O(n^2)}$

Quelle: Informatik II Vorlesung

Sortieren

Sortieren

- **Gegeben**:
  Array A mit n Elementen $(A[1], \ldots, A[n])$
- **Ausgabe**:
  Permutation A' von A, die sortiert ist:

$$A'[1] \leq A'[2] \leq \ \ldots \leq A'[n]$$

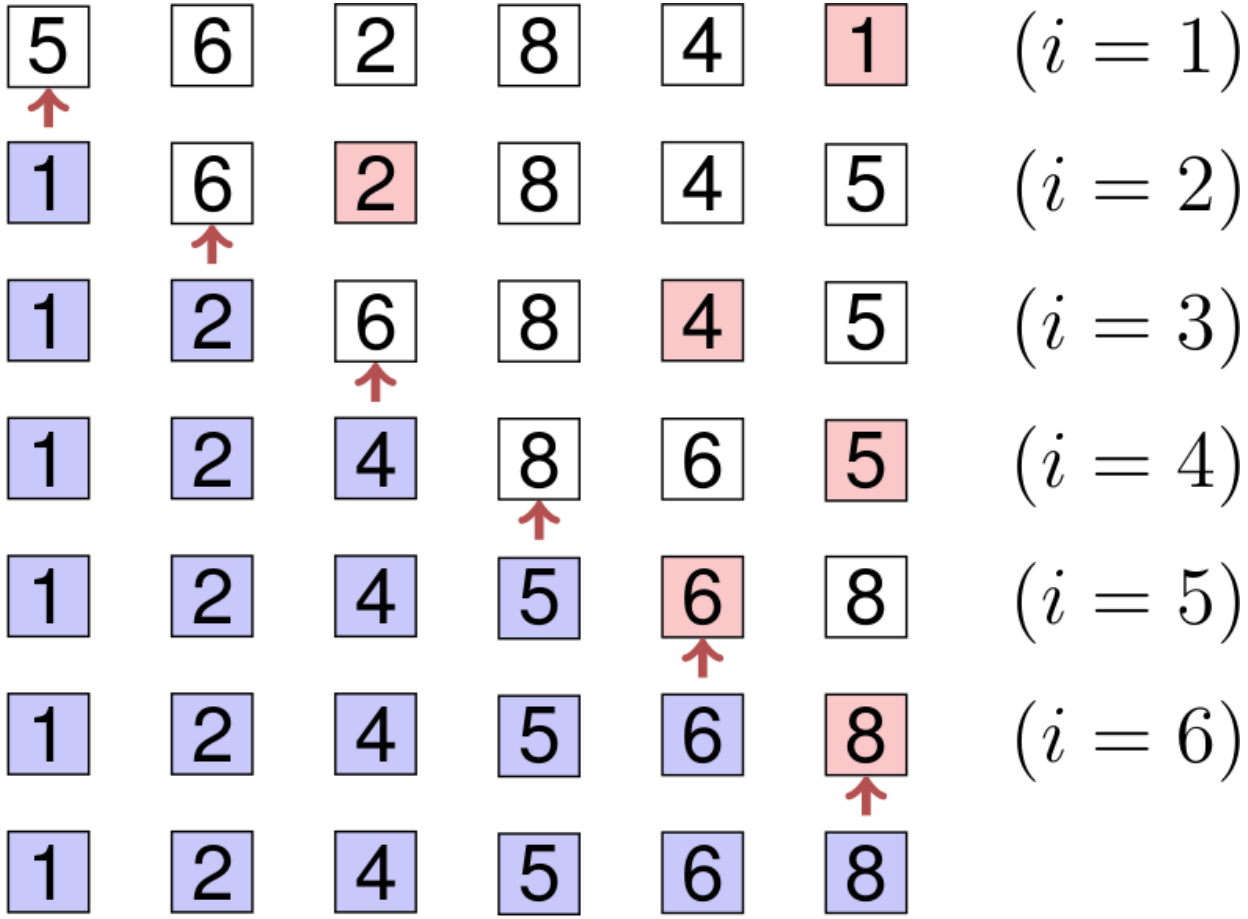
Selection Sort (Sortieren durch Auswahl)

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sortiertes Array A

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

Suche Index p des aktuell kleinsten Elementes im Teilarray $A[i..n]$

Selection Sort (Sortieren durch Auswahl)



Selection Sort (Sortieren durch Auswahl)

- **worst-case Anzahl Vergleiche**: $\boldsymbol{\Theta(n^2)}$
- **best-case Anzahl Vergleiche**: $\boldsymbol{\Theta(n^2)}$

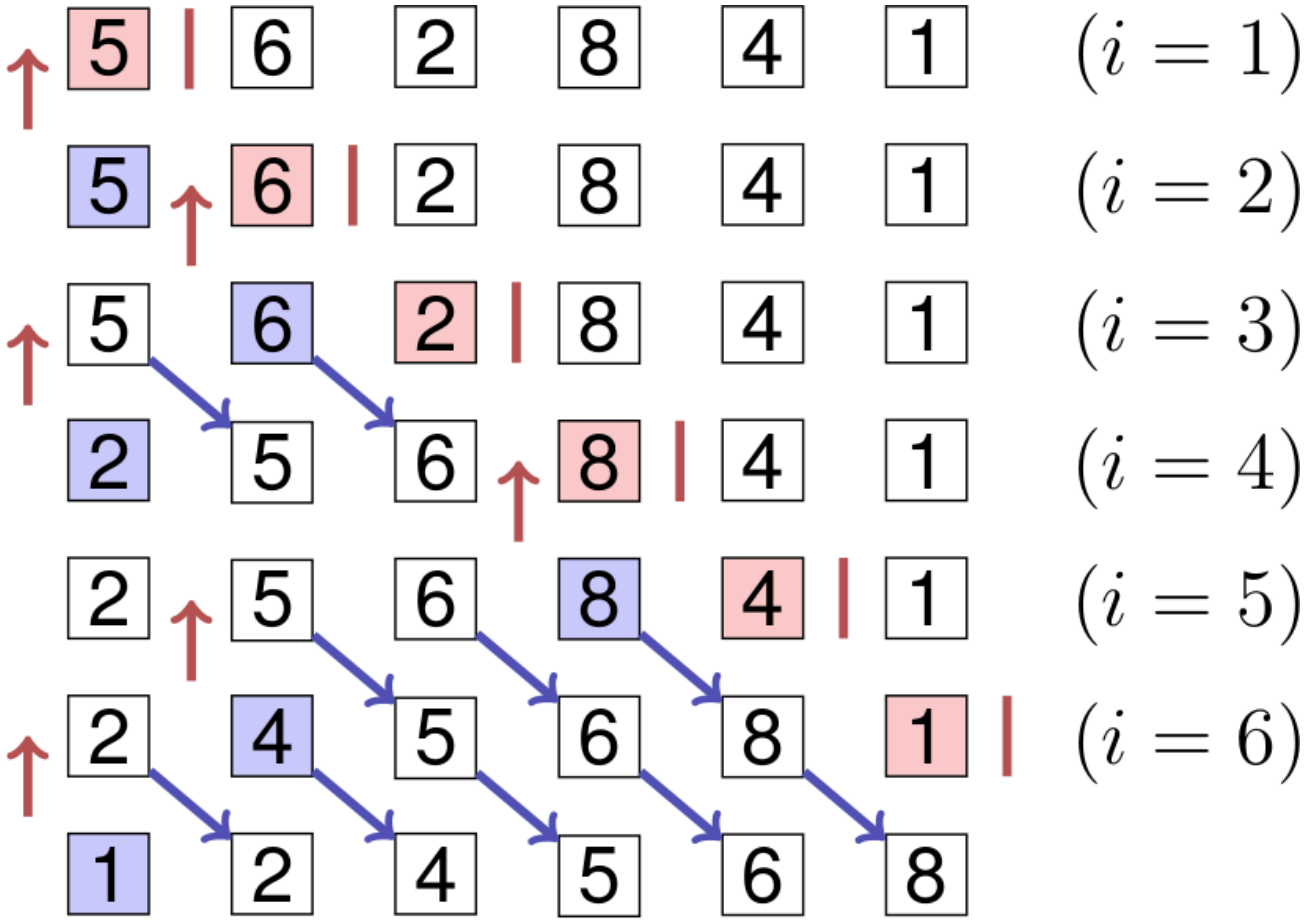
- **worst-case Anzahl Vertauschungen**: $\boldsymbol{\Theta(n)}$

Insertion Sort (Sortieren durch Einfügen)

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sortiertes Array A

```
for i ← 2 to n do
    x ← A[i]
    p ← BinarySearch(A[1...i − 1], x); // Kleinstes p ∈ [1, i] mit A[p] ≥ x
    for j ← i − 1 downto p do
        A[j + 1] ← A[j]
    A[p] ← x
```

Insertion Sort (Sortieren durch Einfügen)



Insertion Sort (Sortieren durch Einfügen)

- **worst-case Anzahl Vergleiche**: $O(n^2)$
- **best-case Anzahl Vergleiche**: $O(n)$

- **worst-case Anzahl Vertauschungen**: $O(n^2)$

Selection Sort vs. Insertion Sort

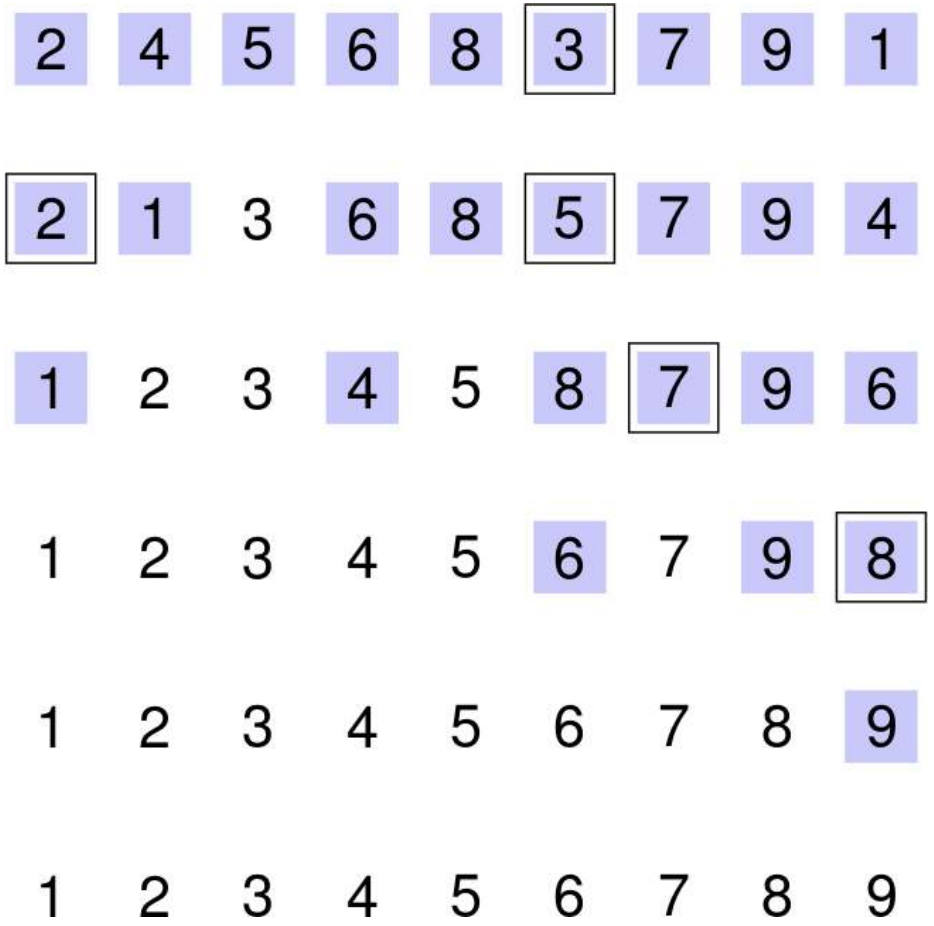
- **Vorteil** Insertion Sort: Suchbereich ist bereits sortiert, deshalb können wir eine binäre Suche anwenden
- **Nachteil** Insertion Sort: Im schlechtesten Fall haben wir viele Elementverschiebungen

Quicksort

Input : Array A der Länge n. $1 \leq l \leq r \leq n$.
Output : Array A, sortiert zwischen l und r.

if $l < r$ **then**
    Wähle Pivot $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition$(A[l, \ldots, r], p)$
    Quicksort$(A[l, \ldots, k-1])$
    Quicksort$(A[k+1, \ldots, r])$

Quicksort mit Random Pivot



Quicksort

- **worst-case Anzahl Vergleiche**: $\boldsymbol{\Theta(n^2)}$
  - Pivot = Minimum oder Maximum
- **best-case Anzahl Vergleiche**: $\boldsymbol{O(n \log(n))}$
  - Pivot = Median

- Bemerkung: Der randomisierte Quicksort (Pivot = Zufällig) benötigt im Mittel $O(n \log(n))$ Vergleiche