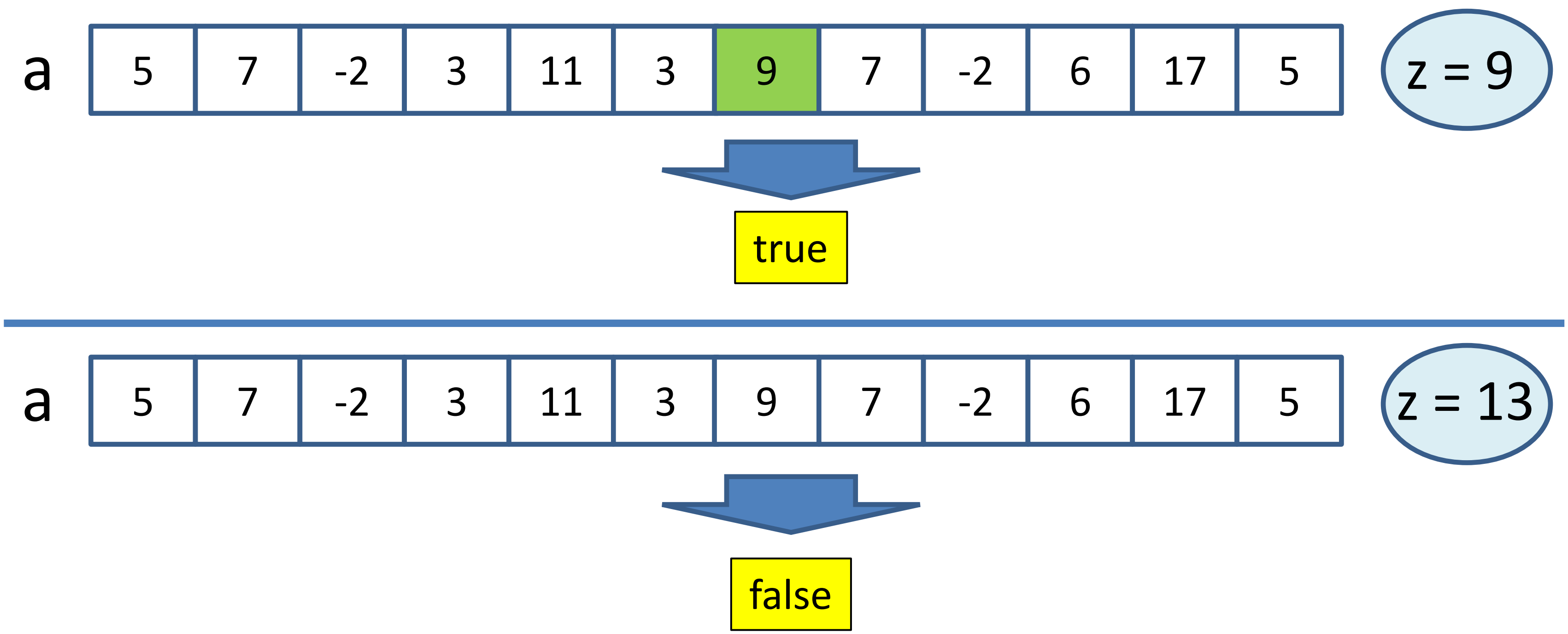
Informatik 1

Suchprobleme - Aufwandsanalyse

Suchen

Suchproblem (unsortierte Folgen)	
Gegeben:	Eine Folge a von n Werten, ein Wert z
Gesucht:	Ist z in der Folge a enthalten?



Suchen

- Algorithmus
  - Gehe jedes Elemente der Folge von Anfang bis Ende durch und vergleiche es mit z
  - Wenn z gefunden ist: mit true abbrechen
  - Ansonsten nach Schleife mit false abbrechen
- <u>Lineare, Sequentielle Suche</u>

```
public boolean istEnthalten1(int [] a, int z) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == z) {
      return true;
    }
  }

  return false;
}
```

1

Suchen

Suchproblem (sortierte Folgen)	
Gegeben:	Eine aufsteigend sortierte Folge a von n Werten, ein Wert z
Gesucht:	Ist z in der Folge a enthalten?

a	-2	-2	3	3	5	5	6	7	7	9	11	17

z = 9

true

a	-2	-2	3	3	5	5	6	7	7	9	11	17

z = 13

false

Suchen

- Algorithmus
  - istEnthalten1 löst das Problem (auch)
  - Sortierung ausnutzen

a													
a	-2	-2	3	3	5	5	6	7	7	9	11	17	z = 4

4 kann ab hier nicht enthalten sein

```java
public boolean istEnthalten2(int [] a, int z) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == z) {
      return true;
    } else if (a[i] > z) {
      return false;
    }
  }
  return false;
}
```

Suchen

- Welcher Algorithmus ist besser?

  1 oder 2?
- Zeit messen zum Vergleichen?
  - Hardware wird immer schneller
  - Compiler könnte besseren Code erzeugen
  - Nur endliche viele Eingaben können gemessen werden
- Messen produziert keine dauerhaften Ergebnisse!
- Ungenaue Alternative zu Messen?

  Welche Temperatur in Grad Celsius hat die Luft außerhalb des Gebäudes? (kein Thermometer vorhanden)

Suchen

- Anzahl ausgeführter Befehle schätzen
- Schätzgrundlage
  - Berechenbarkeitsmodell nötig
- z.B. Java Byte Code
  - Abstrakte Maschinensprache
  - Alle Daten sind auf einem Stapelspeicher
  - **Jeder Befehl verbraucht konstant viel Zeit**

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84;
34:  iload_1
35:  invokevirtual   #85;
38:  iinc    1, 1
41:  goto    2
```

Aufwandsanalyse

Problembeschreibung	
Gegeben (Eingabe):	Menge E aller Eingaben e n := |e| Eingabekomplexität
Gesucht (Ausgabe):	…

Berechnungsmodell festlegen für Aufwandsab**schätzung**, z.B:

Anzahl ausgeführter Byte-Code-Befehle der JVM

A sei ein deterministisches Java Programm (Algorithmus), der das Problem löst

Definitionen		**Beschreibung**
$T(e)$	Anzahl Byte-Code-Befehle, die A für Eingabe e ausführt T ist wohldefiniert, aber uns nicht bekannt	
$T_{bc}(\mathrm{n})$	$\min\{\, T(e) \mid \forall e \in E : n = \lvert e \rvert \}$	Zeitaufwand im besten Fall
$T_{wc}(\mathrm{n})$	$max\{\, T(e) \mid \forall e \in E : n = \lvert e \rvert \}$	Zeitaufwand im schlimmsten Fall
$T_{ac}(\mathrm{n})$	$\frac{\sum_{\lvert e \rvert = n} T(e)}{n}$	Durchschnittlicher Zeitaufwand

Aufwandsanalyse

- Eingabekomplexität n muss so gewählt werden, das mit der Problemkomplexität auch n wächst
  - n meist vorgegeben
- Die Anzahl ausgeführter Bytecodebefehle anhand der Java-Quelltexte für alle drei Fälle in Abhängigkeit von n abschätzen

Anweisung	Anzahl	Beispiel	Schätzung

Variablendeklaration
Ausdruck
	Variablen
	Operatoren
		int i = 5;
		int j = i + 5;
		i = j * 2;
			1
			2
			2

Felder	Erzeugte Feldelemente	char [] hi = {‘H‘, ‘i‘}; new int[n][5]	2 5*n

Kontrollanweisungen	Alle ausgeführten Befehle	If (a > 0) {
		a = a + 5;
		}
			3

Methodenaufruf Konstruktor	Ausdrücke der Parameter + alle ausgeführten Befehle	getFakultaet(n) (rekursiv)	2*n

Aufwandsanalyse

```
public boolean istEnthalten1(int [] a, int z) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == z) {
      return true;
    }
  }
  return false;
}
```

1

- Schlimmster Fall: z ist nicht in a enthalten

$$T_{wc}(\mathrm{n}) = 1 + 2\mathrm{n} + 2\mathrm{n} + 1 = 4\mathrm{n} + 2$$

`int i = 0`

`i < a.length`

`a[i] == z`

`return false`

Aufwandsanalyse

```
public boolean istEnthalten1(int [] a, int z) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == z) {
      return true;
    }
  }
  return false;
}
```

1

- Durchschnittlicher Fall: z ist in der Mitte

$$T_{ac}(\mathrm{n}) = 1 + \mathrm{n} + \mathrm{n} + 1 = 2\mathrm{n} + 2$$

`int i = 0`

`i < a.length`

`a[i] == z`

`return true`

Aufwandsanalyse

- Diese Schätzungen sind ungenau

  `i < a.length`

  vielleicht 3 Bytecodebefehle (oder 2 oder 4)

  1. Wert von i holen
  2. Wert von a.length holen
  3. beide Werte vergleichen

- Es sind nur konstant viele Bytecodebefehle für diesen Ausdruck nötig
- Konstante Faktoren sind irrelevant in den Schätzungen
- Funktionen im O-Kalkül vereinfach angeben reicht für Vergleich von Algorithmen

O-Kalkül (Landau-Notation)

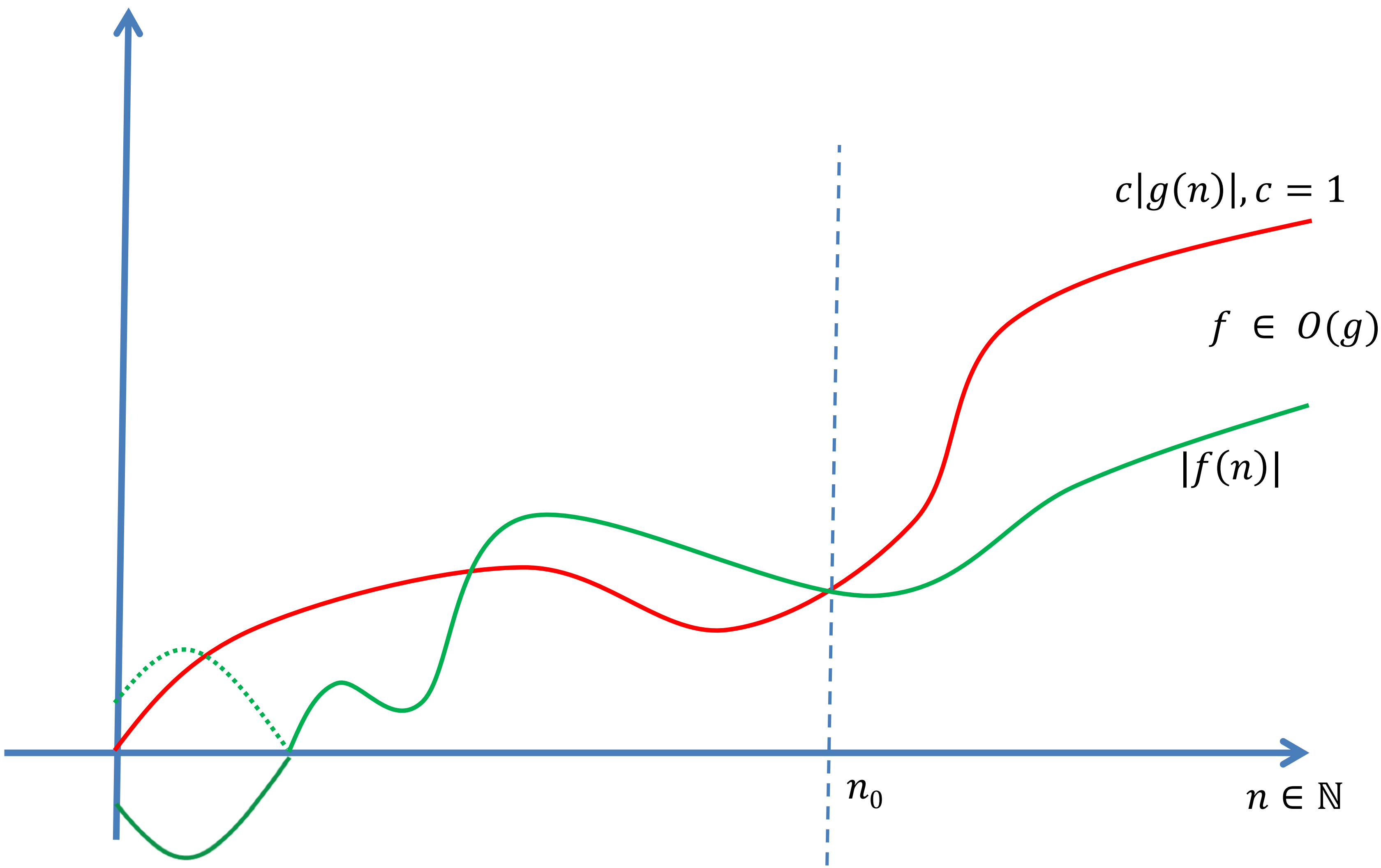
Notation $f, g: \mathbb{N} \to \mathbb{R}$	**Definition**	**Analogie Zahlen**
$f \in O(g)$ $f(n) = O(g(n))$	$O(g) := \{ f \mid \exists c > 0 \, \exists n_0 > 0 \; \forall n \geq n_0 : \lvert f(n) \rvert \leq c \lvert g(n) \rvert \}$ „f wächst höchstens so schnell wie g“	$a \leq b$ „$f \leq g$“
$f(n) = o(g(n))$	$f \in O(g)$ und $f \notin \Omega(g)$	$a < b$
$f(n) = \theta(g(n))$	$f \in O(g)$ und $f \in \Omega(g)$	$a = b$
$f(n) = \omega(g(n))$	$g \in o(f)$	$a > b$
$f(n) = \Omega(g(n))$	$g \in O(f)$ [D. Knuth]	$a \geq b$

Funktionen werden im O-Kalkül immer vereinfacht angegeben

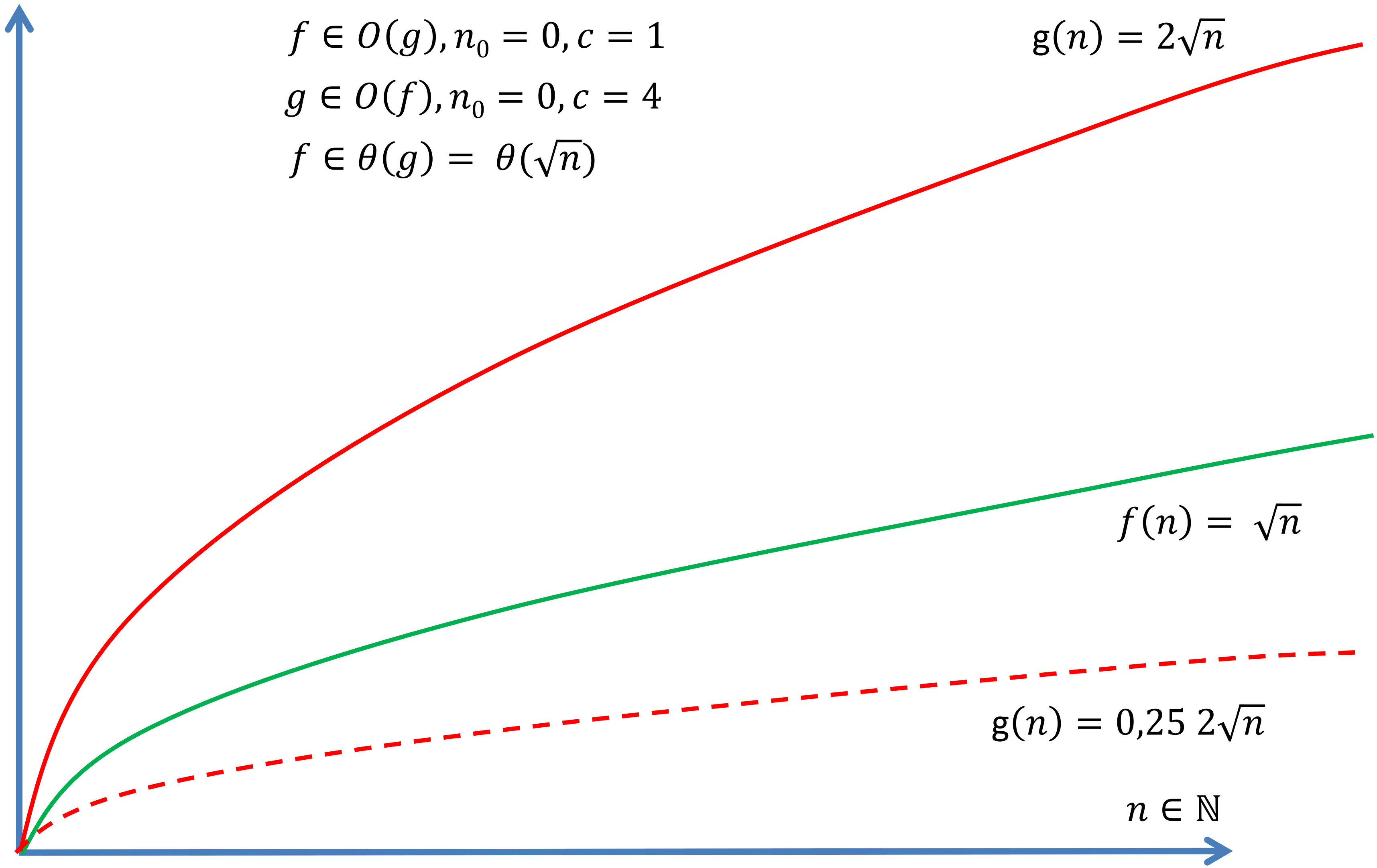
Rechenregeln zum Vereinfachen	**Beispiel**
$O(cf) = O(f)$ für $c \in \mathbb{R}$	$O(5n^2) = O(n^2)$
$O(f + g) = O(f)$ falls $g = O(f)$	$O(n^2 + n \log n) = O(n^2)$

Rechenregeln gelten für alle fünf Notationen

O-Kalkül (Landau-Notation)



O-Kalkül (Landau-Notation)



Aufwandsanalyse

- Anzahl Schritte Algorithmus 1 höchstens:
  - $T_{bc}(\mathrm{n})$ = O$(6) = O(1)$
  - $T_{wc}(\mathrm{n})$ = O $(4n + 2) =$ O (n)
  - $T_{ac}(\mathrm{n})$ = O$(2n + 2) = \mathrm{O}(\mathrm{n})$
- Anzahl Schritte Algorithmus 2 höchstens:
  - Es reicht aus, die Vergleiche i < a.length zu zählen
  - $T_{bc}(\mathrm{n}) = \mathrm{O}(1)$
  - $T_{wc}(\mathrm{n}) = \mathrm{O}(\mathrm{n})$
  - $T_{ac}(\mathrm{n}) = \mathrm{O}\left(\frac{\mathrm{n}}{2}\right) = \mathrm{O}(\mathrm{n})$
- Beide Algorithmen haben in allen drei Fällen den gleichen Zeitaufwand
- Konstanter Faktor 2. Algorithmus schlechter als vom 1.
- Praxis: Konstanter Faktor ist oft relevant

Aufwandsanalyse

- Abschätzung war nach oben
  - $T_{wc}(\mathrm{n}) \leq$ cn = O(n) „höchsten n Schritte“
- Abschätzung gilt hier nach unten (mindest Anzahl Schritte)
  - $T_{wc}(\mathrm{n}) \geq cn = \Omega(n)$ „mindestens n Schritte“
- Algorithmus 1 („genau n Schritte“)
  - $T_{bc}(\mathrm{n}) = \theta(1)$
  - $T_{wc}(\mathrm{n}) = \theta(n)$
  - $T_{ac}(\mathrm{n}) = \theta(\mathrm{n})$
- Algorithmus 2
  - $T_{bc}(\mathrm{n}) = \theta(1)$
  - $T_{wc}(\mathrm{n}) = \theta(\mathrm{n})$
  - $T_{ac}(\mathrm{n}) = \theta\left(\frac{\mathrm{n}}{2}\right) = \theta(\mathrm{n})$
- Angabe oft nur O(n) sprachlich mit „hat genau O(n) Schritte“

Aufwandsanalyse

- Nur Befehle abschätzen, die oft durchlaufen werden (O-Kalkül-Regel schon bei der Schätzung berücksichtigen)
  - Initialisierung Schleifenvariable irrelevant
  - Schleifenrumpf konstante Anzahl Anweisungen
  - Anzahl Abbruchbedingung prüfen reicht in diesem Beispiel (oder Anzahl Vergleiche in if-Anweisung)

$$T_{wc}(\mathrm{n}) = n\ c = \theta(n)$$

```
public boolean istEnthalten1(int [] a, int z) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == z) {
      return true;
    }
  }
  return false;
}
```

Aufwandsanalyse

- Kann in einem aufsteigend sortiertem Feld schneller als O(n) gesucht werden?
- Wie können systematisch mit einem Vergleich viele Werte von der Suche ausgeschlossen werden?

a	-2	-2	3	3	5	5	6	7	7	9	11	17

z = 13

Binärsuche

- z mit Wert in der Mitte m vergleichen

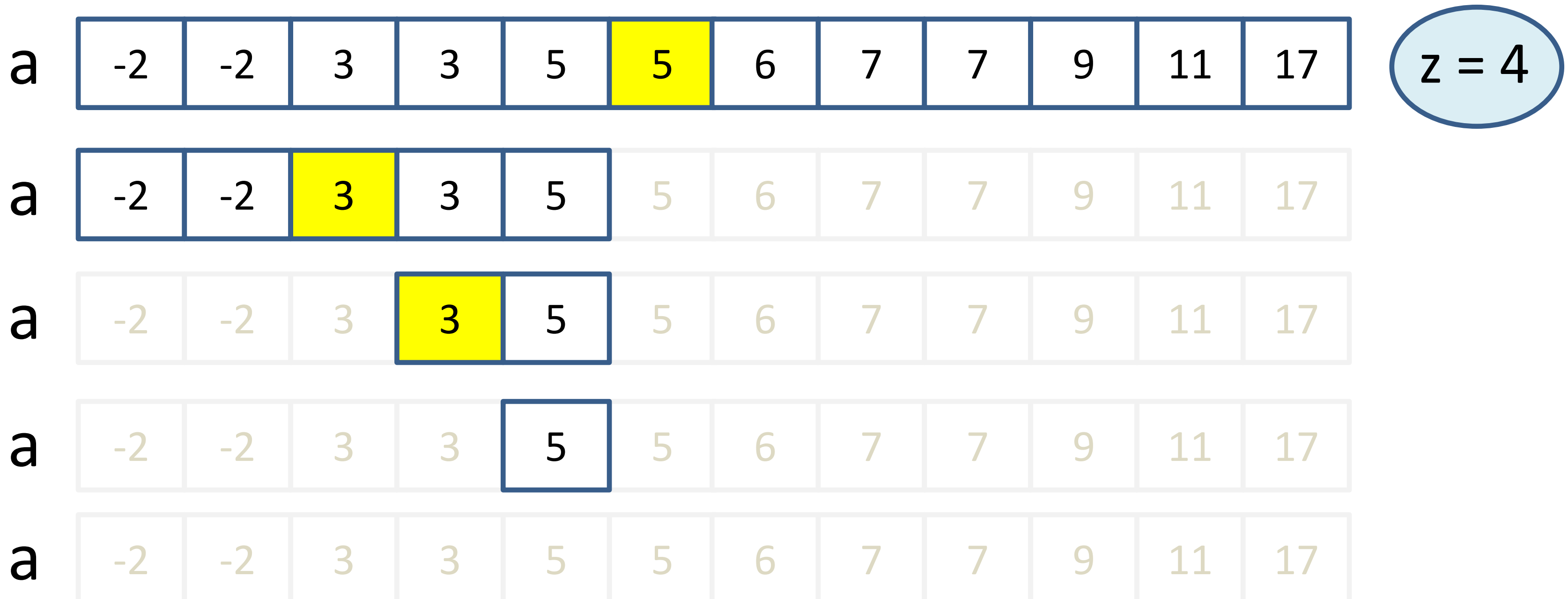
						m							
a	-2	-2	3	3	5	5	6	7	7	9	11	17	z = 6

- Drei Fälle
  1. a[m] = z: Fertig
  2. a[m] > z: in der linken Hälfte (rekursiv) suchen
  3. a[m] < z: in der reichten Hälfte (rekursiv) suchen

						m			m				
a	-2	-2	3	3	5	5	6	7	7	9	11	17	z = 6

						m	m		m				
a	-2	-2	3	3	5	5	6	7	7	9	11	17	z = 6

Binärsuche



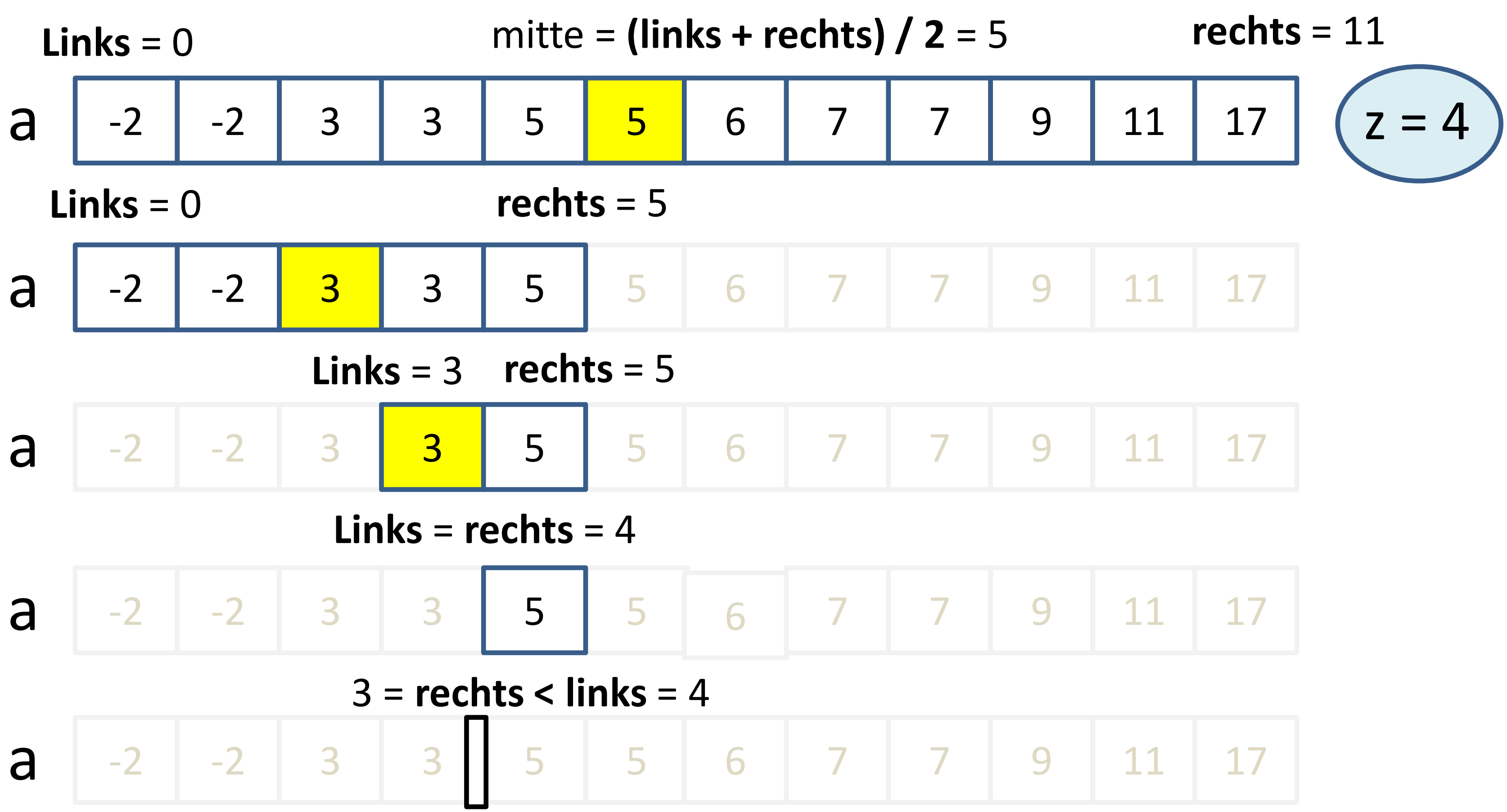
- Rekursionsabbruch

  Wenn z gefunden wurde

  oder der zu durchsuchende Bereich leer geworden ist

Binärsuche

- Bereich, in dem gesucht wird, markieren



Binärsuche

```
private boolean binaerSuchen(int [] a, int zahl,
                             int links, int rechts) {
  if (rechts < links) {
    return false;
  }
  int mitte = (links + rechts) / 2;
  if (a[mitte] == zahl) {
    return true;
  } else if (a[mitte] < zahl) {
    return binaerSuchen(a, zahl, mitte + 1, rechts);
  } else {
    return binaerSuchen(a, zahl, links, mitte - 1);
  }
}
```

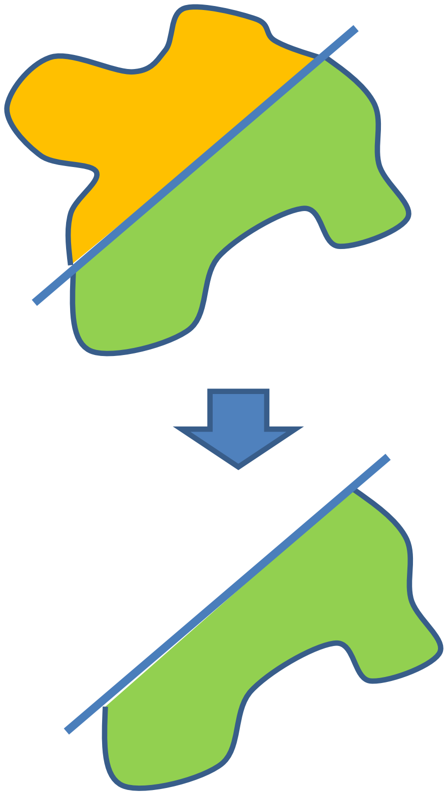
Lineare Rekursion oder verzweigende Rekursion?

Halbierungsverfahren

Rekursiver Algorithmus

Falls Problem nicht direkt zu lösen ist:

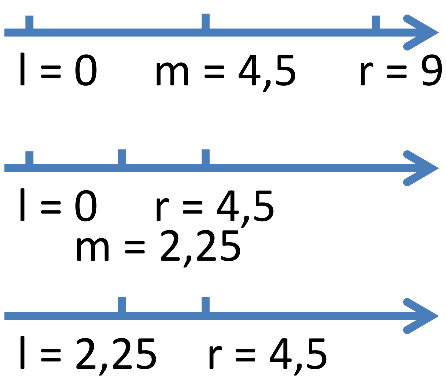
- Problem in zwei Teile unterteilen
- Rekursiv mit dem Teilproblem fortfahren, das die Lösung enthält



Beispiel: Verfahren von Heron

Näherung von $\sqrt{a}$ berechnen, $a \geq 0$

z.B. a = 9, Beginn l = 0, r = 9



$9 = a < 16 < m^2$

Links suchen

$9 = a > m^2$

Rechts suchen

Verfahren nähert sich Lösung Rasch an

Binärsuche

- Zeitaufwand?
  - Beste Fall O(1)
- Schlimmste Fall: Direktes Zählen wegen Rekursion nicht möglich.
- Konstant viele Befehle pro Aufruf (bis zum nächsten rekursiven Aufruf)
- **Aufrufbaum betrachten**: maximale Rekursionstiefe $\theta(log_2 n)$
  - Zeitaufwand $T_{wc}(\mathrm{n}) = \theta(log_2 n)$

Rekurrenzgleichungen

- Alternative Methode, für rekursive Algorithmen, den Zeitaufwand zu bestimmen
- $T_{wc}(\mathrm{n})$ existiert und gibt für **jedes** n die Anzahl ausgeführter Bytecodebefehle der Binärsuche an, **auch für n / 2**
- Rekursive Definition (nur noch T genannt)

$$T(\mathrm{n}) = \begin{cases} c > 0 & , n = 0 \\ T(\frac{n}{2} - 1) + \mathrm{c} & , \mathrm{n} > 0 \end{cases}$$

- Abbruch fast immer bei c, da nicht von n abhängig
- <u>Rekurrenzgleichung:</u> Nur rekursiver Teil angegeben

$$T(\mathrm{n}) = T(\frac{n}{2}) + \mathrm{c} \qquad , \ \mathrm{n} > 0$$

- Statt Gleichheit auch nur $T(\mathrm{n}) \leq$ für Abschätzung nach oben oder $T(\mathrm{n}) \geq$ für Abschätzung nach unten möglich

Rekurrenzgleichungen

Geschlossenen Ausdruck für die Funktion „raten“.

Geschlossener Ausdruck: Ausdruck mit endlich vielen elementaren mathematischen Operatoren +, -, *, : sowie einige Funktionen wie Potenz- und Exponentialfunktion und deren Umkehrfunktionen.

Beweis mit vollständiger Induktion führen.

Es gibt viele Möglichkeiten zur „raten“:

- Aufrufbaum betrachten
- Rekurrenz auf sich selbst anwenden, bis ein Schema deutlich wird
- Rekurrenz mit konkreten Werten schrittweise ausrechnen und dann Ergebnis verallgemeinern
- Summen versuchen, mit Integral abzuschätzen

Induktionsprinzip

1. Induktionshypothese aufstellen.
   Induktionsparameter beschreiben.

Aussage(n)

2. Induktionshypothese für kleines n, z.B. n = 1, direkt beweisen.

Aussage(1)

3. Induktionsschluss, z. B. von n auf n + 1

Beweis Aussage(n + 1) unter Annahme der Gültigkeit von Aussage(n)

Der Induktionsschluss ist eine Schablone für eine Beweis

Die Schablone liefert eine unendliche Kette von Einzelbeweisen

Beweis Aussage(1 + 1) unter Annahme der Gültigkeit von Aussage(1)

Aussage(2)

Beweis Aussage(2+ 1) unter Annahme der Gültigkeit von Aussage(2)

Aussage(3)

⋮

Rekurrenzgleichung

- Induktionsvermutung: $T(n) = (2 + log_2 n)$ c
- Induktionsanfang für n = 0 probieren

$$T(0) = c \neq (2 + log_2 0)\text{ c}$$

- Induktionsanfang für n = 1

$$T(1) = T(0) + c = c + c = 2c$$
$$= (2 + 0)\, c = (2 + log_2 1)\text{c}$$

- Induktionsschluss von n auf 2n

$$T(2n) = T(n) + c = (2 + log_2 n)c + c$$
$$= (2 + log_2 n)\, c + c\ log_2 2$$
$$= (2 + log_2 2 + log_2 n)c$$
$$= (2 + log_2 2\text{n})\text{c}$$

- Eigentlich noch zu zeigen: $T(n) \leq T(n+1)$

k-kleinste Element (1/6)

k-kleinste Element suchen	
Gegeben:	Folge a von n Werten, 1 <= k <= n
Gesucht :	k-kleinste Element der Folge

<u>k-kleinste Element</u>

k-te Wert in der aufsteigend sortierten Folge

Beispiel

a 8 5 7 3 5 2 9 6, k = 4

2 3 5 <u>5</u> 6 7 8 9, 5 ist 4-kleinste Element

Sonderfälle

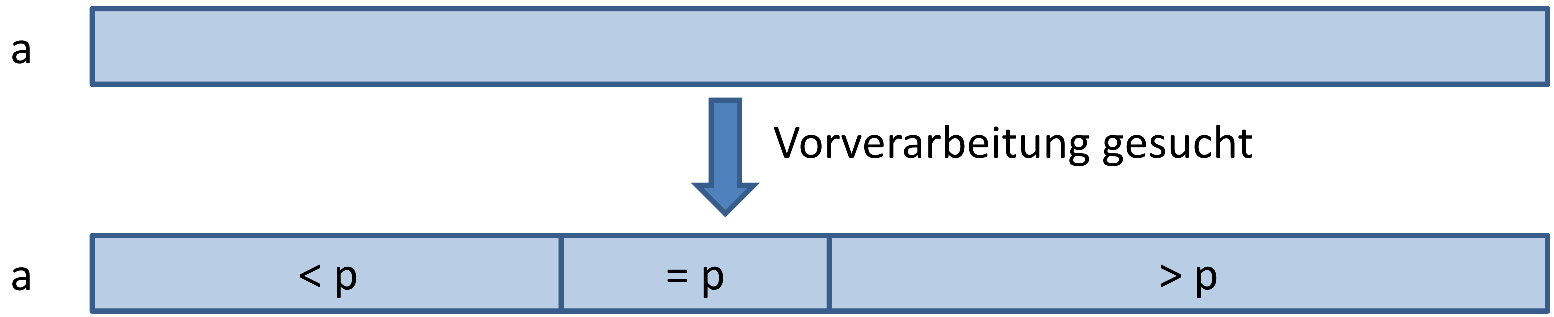
k = 1: Minimum

k = n: Maximum

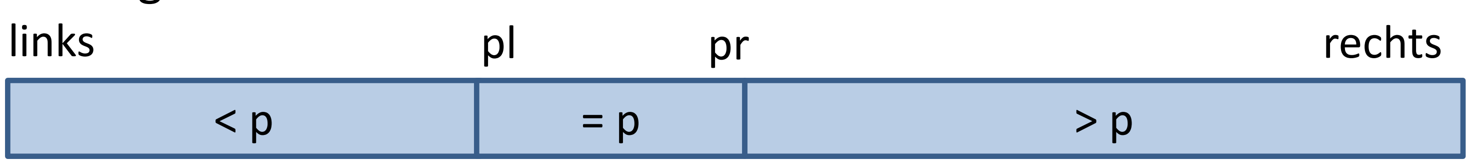
k = n/2: Mittlere Element, Median

k-kleinste Element (2/6)

- Halbierungsverfahren entwickeln

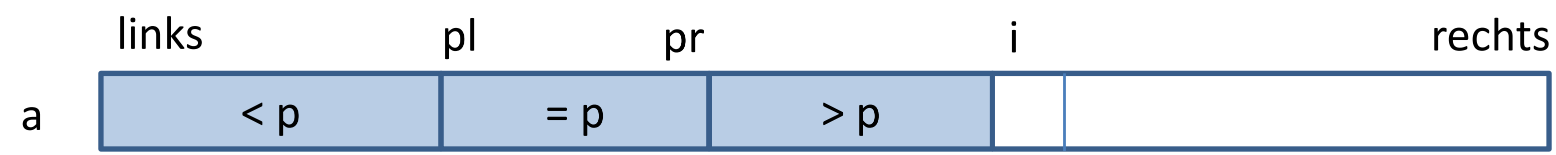


- p muss ein Wert aus der Folge sein
- Drei Fälle
  - k liegt im mittleren Bereich: p ist k-kleinste Element
  - k liegt im linken Bereich: rekursiv links suchen
  - k liegt im rechten Bereich: rekursiv rechts suchen
- Anfang und Ende der Bereiche müssen bekannt sein



k-kleinste Element (3/6)

- Name dieser Unterteilung: **Three-way-partitioning**
- Vorverarbeitung sollte max. O(n) Zeit verbrauchen
- Von links nach rechts probieren
- **Annahme**: a[0] bis a[i - 1] besitzt schon gewünschte Eigenschaft



- Drei Fälle (in jedem Fall wird danach i um Eins erhöht)
  - a[i] > p: nichts zu tun
  - a[i] = p: a[pr + 1] mit a[i] tauschen, pr = pr + 1
  - a[i] < p: a[pl - 1], a[pr - 1] und a[i] tauschen, pr = pr + 1, pl = pl + 1
- Beginn dieser Vorverarbeitung?
- p = a[links] wählen

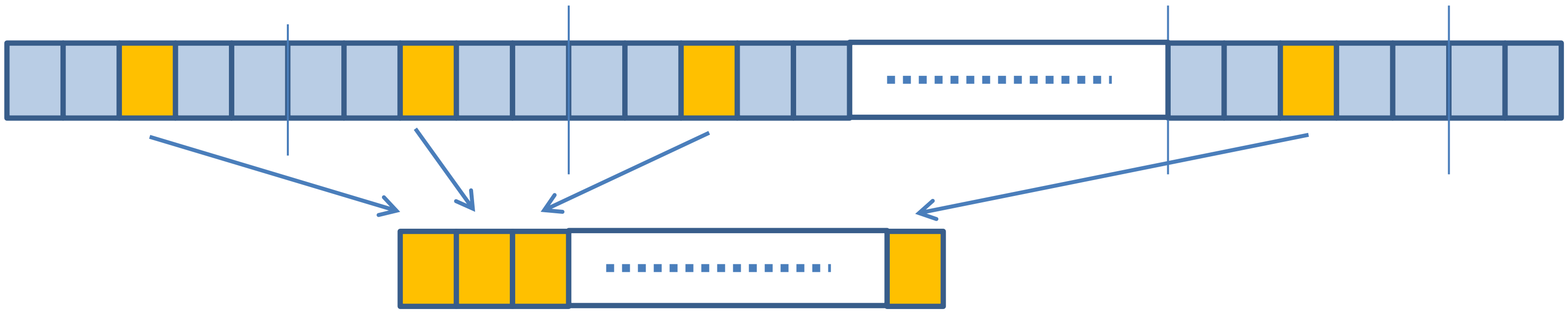
k-kleinste Element (4/6)

- Schlimmste Fall
  - p ist immer das kleinste Element, z.B. wenn a aufsteigend sortiert ist
  - $T(n) = T(n-1) + c\,n$
- Beste Fall
  - p ist immer der Median
  - $T(n) = T(n/2) + c\,n$
- Durchschnittliche Fall
  - In 50% aller Fälle ist p ein Wert, so dass jede Hälfte mindestens ¼ der Werte enthält
  - Ein Hälfte kann maximal ¾ der Werte enthalten
  - $T(n) = T(\frac{3}{4} n) + c\,n$

k-kleinste Element (5/6)

- Verbesserung
  - Einzige Möglichkeit: p besser wählen, z.B. so dass wie im durchschnittlichen Fall jede Hälfte mindestens ¼ der Werte enthält
  - Zusätzliche Vorverarbeitung nötig: alle fünf aufeinander folgende Werte sortieren
  - Insgesamt O(n) Schritte

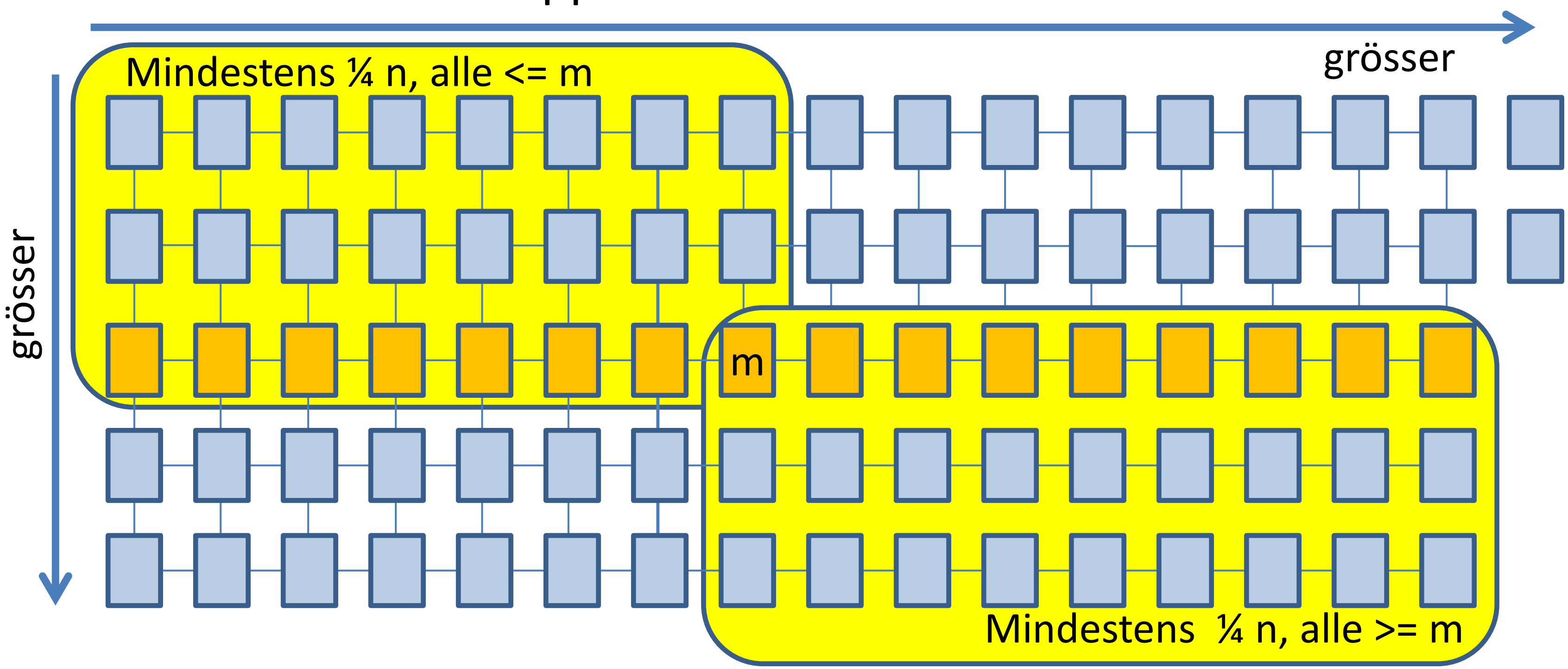
a



- Neue Folge b aus den 5er-Medianen bilden
- Median von b rekursiv bestimmen
- Rekursion des gesamten Algorithmus bei n > 5 abbrechen und mit Sortieren k-kleinsten Wert bestimmen

k-kleinste Element (6/6)

- Eigenschaft des Median m der 5er-Mediane?
- Werte der 5er-Gruppen nach Größe anordnen



- Mindestesten ¼ der Werte sind <= m oder >= m (gelb)
- T(n) = T(1/5 n) + T(¾ n) + c n im schlimmsten Fall (kleinste Hälfte hat ¼ n Werte)
- T(n) = O(n) , Beweis mit Aufrufbaum
- <u>Hasse-Diagramm</u>: Werte werden entlang Achsen geordnet dargestellt. Strich (oder Pfeil) zwischen Werten, deren Ordnungsrelation bekannt ist.

Untere Schranke eines Problem

- Verbesserte Algorithmus: **Median-der-Mediane-Algorithmus**.
- f(n) bzw. O(f(n)) ist eine <u>untere Schranke</u> für ein Problem, wenn jeder Algorithmus, der dieses Problem löst mindestens O(f(n)) Schritte im **schlimmsten Fall** benötigt.
- Kein kann Algorithmus schneller sein, als die untere Schranke.
- Ein Algorithmus, dessen Laufzeit im schlimmsten Fall die untere Schranke eines Problems erreicht, heißt <u>optimal.</u>
- Die untere Schranke für das k-kleinste Element aus einer Folge mit n Werten zu suchen ist O(n),
  - jeder Wert der Folge muss mindestens einmal betrachtet werden
  - der Median-der-Mediane-Algorithmus erreicht O(n) im schlimmsten Fall
- Der **Median-der-Mediane-Algorithmus** ist **optimal**.