

# Datenstrukturen und Algorithmen

## Vorlesung 4: Suchen (K5)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<http://moves.rwth-aachen.de/teaching/ss-15/dsa1/>

22. April 2015



## Übersicht

- 1 **Lineare Suche**
  - Average-Case Analyse von Linearer Suche
- 2 **Bilineare Suche**
  - Das Prominentensuche Problem
  - Das Boxenstopp Problem
- 3 **Binäre Suche**
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche

## Übersicht

- 1 **Lineare Suche**
  - Average-Case Analyse von Linearer Suche
- 2 **Bilineare Suche**
  - Das Prominentensuche Problem
  - Das Boxenstopp Problem
- 3 **Binäre Suche**
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche

## Formale Definition (I)

### Einige hilfreiche Begriffe

$D_n$  = Menge aller Eingaben der Länge  $n$

$t(I)$  = für Eingabe  $I$  benötigte Anzahl elementarer Operationen

$Pr(I)$  = Wahrscheinlichkeit, dass Eingabe  $I$  auftritt

### Woher kennen wir:

$t(I)$ ? – Durch Analyse des fraglichen Algorithmus.

$Pr(I)$ ? – Erfahrung, Vermutung (z. B. „alle Eingaben treten mit gleicher Wahrscheinlichkeit auf“).

## Formale Definition (II)

### Average-Case Laufzeit

Die **Average-Case** Laufzeit von  $A$  ist die von  $A$  **durchschnittlich** benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge  $n$ :

$$A(n) = \sum_{I \in D_n} \Pr(I) \cdot t(I)$$

## Lineare Suche

### Rechenproblem

**Eingabe:** Array  $E$  mit  $n > 0$  Einträgen, sowie das gesuchte Element  $K$ .

**Ausgabe:** Ist  $K$  in  $E$  enthalten?

```

1 bool linSearch(int E[], int n, int K) {
2   for (int index = 0; index < n; index++) {
3     if (E[index] == K) {
4       return true; // oder: return index;
5     }
6   }
7   return false; // nicht gefunden
8 }
```

## Intermezzo: Erwartungswerte

Betrachte einen einarmigen Banditen. Er hat 3 Räder und jedes Rad ist beschriftet mit Herz- und Karo-Symbolen.

Jedes Rad wird unabhängig von allen anderen Rädern angestoßen; jedes liefert Herz oder Karo, beides mit der Wahrscheinlichkeit  $\frac{1}{2}$ .

Man gewinnt den ganzen Jackpot wenn alle Räder ein Herz-Symbol zeigen.

Man gewinnt die Hälfte des Jackpots wenn alle Räder ein Karo-Symbol zeigen.

Sonst gewinnt man nichts.

Frage: Wieviel Prozent des Jackpots gewinnt man im Schnitt?

Antwort:  $\frac{1}{8} \times 1 + \frac{1}{8} \times \frac{1}{2} + \frac{6}{8} \times 0 = \frac{3}{16}$ .

## Lineare Suche – Analyse

### Elementare Operation

Vergleich einer ganzen Zahl  $K$  mit Element  $E[\text{index}]$ .

### Menge aller Eingaben

$D_n$  ist die Menge aller Permutationen von  $n$  ganzen Zahlen, die ursprünglich aus einer Menge  $N > n$  ganzer Zahlen ausgewählt wurden.

### Zeitkomplexität

- ▶  $W(n) = n$ , da  $n$  Vergleiche notwendig sind, falls  $K$  nicht in  $E$  vorkommt (oder wenn  $K == E[n]$ ).
- ▶  $B(n) = 1$ , da ein Vergleich ausreicht, wenn  $K$  gleich  $E[1]$  ist.
- ▶  $A(n) \approx \frac{1}{2}n$ , da im Schnitt  $K$  mit etwa der Hälfte des Arrays  $E$  verglichen werden muss? – **Nein**.

## Lineare Suche – Average-Case-Analyse (I)

### Zwei Szenarien

1.  $K$  kommt nicht in  $E$  vor.
2.  $K$  kommt in  $E$  vor.

### Zwei Definitionen

1. Sei  $A_{K \notin E}(n)$  die Average-Case-Laufzeit für den Fall " $K$  nicht in  $E$ ".
2. Sei  $A_{K \in E}(n)$  die Average-Case-Laufzeit für den Fall " $K$  in  $E$ ".

$$A(n) = \Pr\{K \text{ in } E\} \cdot A_{K \in E}(n) + \Pr\{K \text{ nicht in } E\} \cdot A_{K \notin E}(n)$$

## Herleitung

$$\begin{aligned}
 A(n) &= \Pr\{K \text{ in } E\} \cdot A_{K \in E}(n) + \Pr\{K \text{ nicht in } E\} \cdot A_{K \notin E}(n) \\
 &\quad \left| A_{K \in E}(n) = \frac{n+1}{2} \right. \\
 &= \Pr\{K \text{ in } E\} \cdot \frac{n+1}{2} + \Pr\{K \text{ nicht in } E\} \cdot A_{K \notin E}(n) \\
 &\quad \left| \Pr\{\text{nicht } B\} = 1 - \Pr\{B\} \right. \\
 &= \Pr\{K \text{ in } E\} \cdot \frac{n+1}{2} + (1 - \Pr\{K \text{ in } E\}) \cdot A_{K \notin E}(n) \\
 &\quad \left| A_{K \notin E}(n) = n \right. \\
 &= \Pr\{K \text{ in } E\} \cdot \frac{n+1}{2} + (1 - \Pr\{K \text{ in } E\}) \cdot n \\
 &= n \cdot \left(1 - \frac{1}{2} \Pr\{K \text{ in } E\}\right) + \frac{1}{2} \Pr\{K \text{ in } E\}
 \end{aligned}$$

## Der Fall " $K$ in $E$ "

- ▶ Angenommen alle Elemente in  $E$  sind **unterschiedlich**.
- ▶ Damit ist die Wahrscheinlichkeit für  $K == E[i]$  gleich  $\frac{1}{n}$ .
- ▶ Die Anzahl benötigter Vergleiche im Fall  $K == E[i]$  ist  $i + 1$ .
- ▶ Damit ergibt sich:

$$\begin{aligned}
 A_{K \in E}(n) &= \sum_{i=0}^{n-1} \Pr\{K == E[i] | K \text{ in } E\} \cdot t(K == E[i]) \\
 &= \sum_{i=0}^{n-1} \left(\frac{1}{n}\right) \cdot (i + 1) \\
 &= \left(\frac{1}{n}\right) \cdot \sum_{i=0}^{n-1} (i + 1) \\
 &= \left(\frac{1}{n}\right) \cdot \frac{n(n+1)}{2} \\
 &= \frac{n+1}{2}.
 \end{aligned}$$

## Lineare Suche – Average-Case-Analyse

### Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

$$A(n) = n \cdot \left(1 - \frac{1}{2} \Pr\{K \text{ in } E\}\right) + \frac{1}{2} \Pr\{K \text{ in } E\}$$

### Beispiel

Wenn  $\Pr\{K \text{ in } E\}$

- = 1, dann ist  $A(n) = \frac{n+1}{2}$ , d. h. etwa 50% von  $E$  ist überprüft.
- = 0, dann ist  $A(n) = n = W(n)$ , d. h.  $E$  wird komplett überprüft.
- =  $\frac{1}{2}$ , dann ist  $A(n) = \frac{3 \cdot n}{4} + \frac{1}{4}$ , d. h. etwa 75% von  $E$  wird überprüft.

## Übersicht

### 1 Lineare Suche

- Average-Case Analyse von Linearer Suche

### 2 Bilineare Suche

- Das Prominentensuche Problem
- Das Boxenstopp Problem

### 3 Binäre Suche

- Was ist binäre Suche?
- Worst-Case Analyse von Binärer Suche

## Bilineare Suche – Analyse

### Worst-Case Zeitkomplexität

Im schlimmsten Fall, wird die Schleife  $\lceil \frac{n}{2} \rceil$  mal durchlaufen.

Pro Schleife finden zwei Vergleiche  $E[i] == K$  statt.

Also  $W(n) = 2 \lceil \frac{n}{2} \rceil$ .

### Best-Case Zeitkomplexität

$B(n) = 2$ , da zwei Vergleiche reichen, wenn  $E[1] == K$  oder  $E[n] == K$ .

### Average-Case Zeitkomplexität

Ähnlich wie für die lineare Suche.

## Bilineare Suche

Statt eine Reihe in einer Richtung zu durchsuchen, kann man dies auch in beide Richtungen "zeitgleich".

Dies führt zur **bilineare** Suche.

```

1 bool bilinSearch(int E[], int n, int K) {
2   int left = 0, right = n - 1;
3   while (left <= right) {
4     if (E[left] == K || E[right] == K) {
5       return true;
6     }
7     left = left + 1;
8     right = right - 1;
9   }
10  return false; // nicht gefunden
11 }
```

## Bilineare Suche

Vereinfachung wenn gegeben ist, dass  $K$  in  $E$  vorkommt, und wenn verzichtet wird auf Terminierung der Suche sobald  $K$  gefunden ist.

Weiterhin soll der Ausgabe  $i$  sein, sodaß  $E[i] == K$  gilt.

```

1 int bilinSearch(int E[], int n, int K) {
2   int left = 0, right = n - 1;
3   while (left != right) {
4     if (E[left] != K || E[right] == K) { left = left + 1; }
5     if (E[right] != K || E[left] == K) { right = right - 1; }
6   }
7   }
8   return left
9 }
```

**Hausaufgabe:** bestimmen Sie für dieses Programm  $W(n)$  und  $A(n)$ .

## Deutsche Prominenten



## Das Prominentensuche Problem

### Was ist ein Prominenter?

Ein **Prominenter** (celebrity) ist jemand den alle kennen, der jedoch selber keinen kennt.

### Beispiel (Das Prominentensuche Problem)

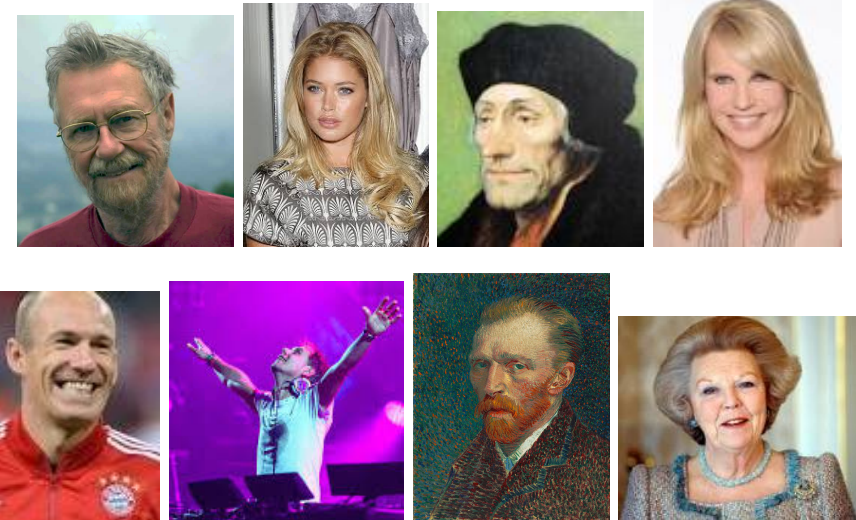
- Eingabe:**
- $n \in \mathbb{N}$  Personen nummeriert  $0, \dots, n-1$
  - Mindestens eine Person ist ein Prominenter
  - $n \times n$  boolean Matrix  $K$ , so dass für  $0 \leq i, j < n$ :

$$K[i, j] = \begin{cases} 1 & \text{falls Person } i \text{ kennt Person } j \\ 0 & \text{sonst} \end{cases}$$

**Ausgabe:** Sei  $k \in \{0, \dots, n-1\}$ , so dass Person  $k$  Prominenter ist, d.h.:

$$\underbrace{\forall 0 \leq i < n. i \neq k \Rightarrow K[i, k]}_{\text{alle kennen Person } k} \text{ und } \underbrace{\forall 0 \leq i < n. i \neq k \Rightarrow \neg K[k, i]}_{\text{Person } k \text{ kennt niemandem}}$$

## Niederländische Prominenten



## Beispiel: Wer ist ein Prominenter?

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

## Beispiel: Wer ist ein Prominenter?

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Es ist einfach, einen Prominenten mit  $W(n) \in O(n^2)$  zu bestimmen.

Geht es auch mit  $W(n) \in O(n)$ ?

## Das Prominentensuche Problem: Lineare Suche

Idee: starte eine Suche am  $K[0,0]$  und suche bis eine 1 gefunden wird in Zeile 0, Spalte  $m$ .

Dann gilt:  $\forall 0 \leq k < m. k$  ist kein Prominenter.

Die Suche geht dann weiter in Zeile  $m$ , Spalte  $m$ , usw.

```

1 int CelebritySearch(bool K[], int n) {
2   int row = 0; column = 0; // Reihe- und Spalte-index
3   while (row != n && column != n) {
4     if (row != column) {
5       if (!K[row,column]) { column = column + 1; }
6       if (K[row,column]) { row = column; }
7     } else { column = column + 1; } // row == column
8   }
9   return row
10 }
```

## Das Prominentensuche Problem: Bilineare Suche

### Einige Eigenschaften

Für alle  $0 < i \neq j \leq n$  gilt:

1.  $K[i,j] \implies i$  ist kein Prominenter
2.  $\neg K[j,i] \implies i$  ist kein Prominenter

Aus dieser Eigenschaft folgt direkt folgende bilineare Suche im Array  $K$ :

```

1 int CelebritySearch(bool K[], int n) {
2   int row = 0, column = n - 1; // Reihe- und Spalte-index
3   while (row != column) {
4     if (K[row,column]) { row = row + 1; } // Property 1
5     if (!K[row,column]) { column = column - 1; } // Property 2
6   }
7   return row
8 }
```

## Das Prominentensuche Problem

### Zeitkomplexität

Es gilt  $A(n), B(n), W(n) \in O(n)$ .

Der Algorithmus kann leicht angepasst werden, damit er terminiert sobald ein Prominenter gefunden wurde.

Dies ändert die asymptotische Zeitkomplexität  $W(n)$  jedoch nicht.

### Hausaufgabe:

Bestimmen Sie die Zeitkomplexität der linearen Suche für dieses Problem.

## Das Boxenstopp Problem



## Das Boxenstopp Problem

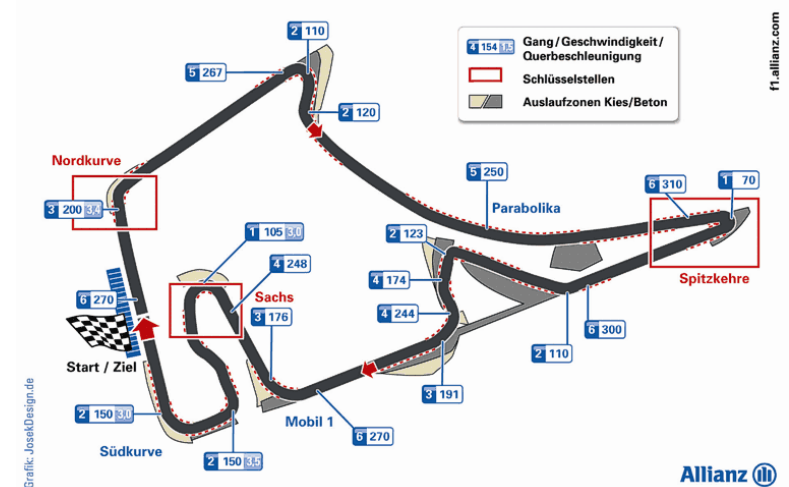
### Beispiel (Das Boxenstopp Problem)

- Eingabe:**
1.  $n \in \mathbb{N}$  Boxenstopps auf den Hockenheimring, rechts herum nummeriert 0 durch  $n-1$ .
  2. Am Boxenstopp  $i$  stehen uns  $T(i)$  Liter Benzin zur Verfügung
  3. Um von Boxenstopp  $i$  nach  $(i+1) \bmod n$  zu fahren braucht man  $V(i)$  Liter Benzin
  4. Gegeben:  $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} V(i)$

**Ausgabe:** Bestimme  $k \in \{0, \dots, n-1\}$ , so dass Sebastian Vettel mit einem leeren Tank **eine komplette Runde** fahren kann.

Erwünschte Worst Case Zeitkomplexität:  $O(n)$ .

## Der Hockenheimring



## Das Boxenstopp Problem: Beispiel

## Das Boxenstopp Problem

### Differenzmatrix

Sei  $D$  eine  $n \times n$  Integermatrix, so dass  $D[i, j]$  die Differenz ist zwischen der Anzahl der Liter Benzin die zur Verfügung stehen und die man braucht um von Boxenstopp  $i$  (rechts herum) nach Boxenstopp  $j$  zu fahren.

$$D[i, j] = \sum_{m=i}^{j-1} T(m) - V(m).$$

### Starting Boxenstopp

Boxenstopp  $0 \leq k < n$  ist **starting Boxenstopp** gdw.  
 $\forall 0 \leq i < n. D[k, i] \geq 0$

## Das Boxenstopp Problem: Bilineare Suche

Mittels einer Hilfsvariable  $d$  bekommen wir jetzt folgenden Algorithmus:

```

1 int PitstopSearch(int V[], int T[], int n) {
2   int left = 0, right = n - 1;
3   int d = V[n-1] - T[n-1]; // d = D[0, n-1]
4   while (left != right) { // Invariant: d = D[left, right]
5     if (d <= 0) { left = left + 1; d = d + V[left] - T[left]; }
6     if (d >= 0) { right = right - 1;
7       d = d + V[right-1] - T[right-1];
8     }
9   }
10  return left
11 }
```

Es ist leicht festzustellen, dass  $W(n) \in O(n)$ , da die Schleife genau  $n$  Mal durchlaufen wird.

## Das Boxenstopp Problem

### Starting Boxenstopp

Boxenstopp  $0 \leq k < n$  ist **starting Boxenstopp** gdw.  
 $\forall 0 \leq i < n. D[k, i] \geq 0$

### Einige Eigenschaften

1. Für alle  $0 \leq i, j < n$  gilt:  $D[i, i] = 0$  und  $D[i, j] + D[j, i] = 0$
2. Für alle  $0 \leq i, j, m < n$  gilt:  $D[i, m] = D[i, j] + D[j, m]$
3.  $k$  ist starting Boxenstopp gdw.  $D[0, k]$  minimal ist
4.  $D[i, j] > 0 \implies j$  ist kein starting Boxenstopp

## Übersicht

- 1 Lineare Suche
  - Average-Case Analyse von Linearer Suche
- 2 Bilineare Suche
  - Das Prominentensuche Problem
  - Das Boxenstopp Problem
- 3 Binäre Suche
  - Was ist binäre Suche?
  - Worst-Case Analyse von Binärer Suche



## Binäre Suche

### Suchen in einem sortierten Array

Eingabe: *Sortiertes* Array  $E$  mit  $n$  Einträgen, und das gesuchte Element  $K$ .

Ausgabe: Ist  $K$  in  $E$  enthalten?

### Idee

Da  $E$  sortiert ist, können wir das gesuchte Element  $K$  schneller suchen.

Liegt  $K$  nicht in der Mitte von  $E$ , dann:

1. suche in der linken Hälfte von  $E$ , falls  $K < E[\text{mid}]$
2. suche in der rechten Hälfte von  $E$ , falls  $K > E[\text{mid}]$

### Fazit:

Wir **halbieren** den Suchraum in jedem Durchlauf.

## Binäre Suche – Beispiel

## Binäre Suche

```

1 bool binSearch(int E[], int n, int K) {
2   int left = 0, right = n - 1;
3   while (left <= right) {
4     int mid = floor((left + right) / 2); // runde ab
5     if (E[mid] == K) { return true; }
6     if (E[mid] > K) { right = mid - 1; }
7     if (E[mid] < K) { left = mid + 1; }
8   }
9   return false;
10 }
```

## Binäre Suche – Analyse

### Lemma

Sei  $r \in \mathbb{R}$  und  $n \in \mathbb{N}$ . Dann gilt:

1.  $\lfloor r + n \rfloor = \lfloor r \rfloor + n$
2.  $\lceil r + n \rceil = \lceil r \rceil + n$
3.  $\lfloor -r \rfloor = -\lceil r \rceil$

## Binäre Suche – Analyse

Abkürzungen:  $m = \text{mid}$ ,  $r = \text{right}$ ,  $l = \text{left}$

### Größe des undurchsuchten Arrays

Im nächsten Durchlauf ist die Größe des Arrays  $m - l$  oder  $r - m$ .

Hierbei ist  $m = \lfloor (l + r)/2 \rfloor$ .

Die neue Größe ist also:

$$\blacktriangleright m - l = \lfloor (l + r)/2 \rfloor - l = \lfloor (r - l)/2 \rfloor = \lfloor (n - 1)/2 \rfloor$$

oder

$$\blacktriangleright r - m = r - \lfloor (l + r)/2 \rfloor = \lceil (r - l)/2 \rceil = \lceil (n - 1)/2 \rceil$$

Im schlimmsten Fall ist die neue Größe des Arrays also:

$$\lceil (n - 1)/2 \rceil$$

## Lösen der Rekursionsgleichung

Betrachte den Spezialfall  $n = 2^k - 1$ .

Da die maximal Größe des Arrays  $\lceil (n-1)/2 \rceil$  ist, leiten wir her:

$$\left\lceil \frac{(2^k - 1) - 1}{2} \right\rceil = \left\lceil \frac{2^k - 2}{2} \right\rceil = \lceil 2^{k-1} - 1 \rceil = 2^{k-1} - 1.$$

Daher gilt für  $k > 0$  nach der Definition  $S(n) = 1 + S(\lceil (n-1)/2 \rceil)$ , daß:

$$S(2^k - 1) = 1 + S(2^{k-1} - 1) \quad \text{und damit} \quad S(2^k - 1) = k + \underbrace{S(2^0 - 1)}_{=0} = k.$$

## Rekursionsgleichung für Binäre Suche

Sei  $S(n)$  die maximale Anzahl der Schleifendurchläufe bei einer erfolglosen Suche.

Wir erhalten die **Rekursionsgleichung**:

$$S(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 + S(\lceil (n-1)/2 \rceil) & \text{falls } n > 0 \end{cases}$$

Die erste Werten sind:

$n$	0	1	2	3	4	5	6	7	8
$S(n)$	0	1	2	2	3	3	3	3	4

Wir suchen eine geschlossene Formel für  $S(n)$

## Binäre Suche – Analyse

$n$	0	1	2	3	4	5	6	7	8
$S(n)$	0	1	2	2	3	3	3	3	4

Vermutung:  $S(2^k) = 1 + S(2^{k-1})$ .

$S(n)$  steigt monoton, also  $S(n) = k$  falls  $2^{k-1} \leq n < 2^k$ .

Oder: falls  $k - 1 \leq \log n < k$ .

Dann ist  $S(n) = \lfloor \log n \rfloor + 1$ .

## Binäre Suche – Analyse

Wir vermuten  $S(n) = \lfloor \log n \rfloor + 1$  für  $n > 0$

Induktion über  $n$ :

Basis:  $S(1) = 1 = \lfloor \log 1 \rfloor + 1$

Induktionsschritt: Sei  $n > 1$ . Dann:

$$S(n) = 1 + S(\lceil (n-1)/2 \rceil) = 1 + \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1$$

Man kann zeigen (Hausaufgabe):  $\lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1 = \lfloor \log n \rfloor$ .

Damit:  $S(n) = \lfloor \log n \rfloor + 1$  für  $n > 0$ .

## Vergleich der Suchalgorithmen

Algorithmus	Zeitkomplexität	Vorteil	Nachteil
Lineare Suche	$O(n)$	einfach	langsam
Bilineare Suche	$O(n)$	einfach / elegant	langsam
Binäre Suche	$O(\log n)$	schnell	sortiertes Array ( $O(n \cdot \log n)$ Initialisierungsaufwand)

## Binäre Suche – Analyse

### Theorem

Die Worst Case Zeitkomplexität der binären Suche ist  $W(n) = \lfloor \log n \rfloor + 1$ .

## Nächste Vorlesung

### Nächste Vorlesung

Freitag 24. April, 10:15 (Großer Hörsaal, Audimax). Bis dann!