Datenstrukturen und Algorithmen

Vorlesung 4: Suchen (K5)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

https://moves.rwth-aachen.de/teaching/ss-18/dsal/

27. April 2018

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

Formale Definition (I)

Einige hilfreiche Begriffe

D_n = Menge aller Eingaben der Länge n

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

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

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

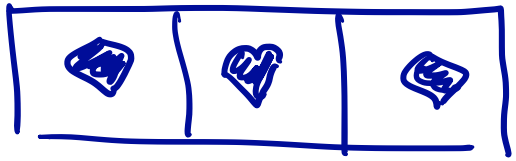
Die Average-Case Laufzeit von A ist die von A durchschnittlich benötigte Anzahl elementarer Operationen auf einer *beliebigen* Eingabe der Länge n:

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

Erwartungswerte

Intermezzo: Erwartungswerte

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



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 $\frac{1}{2}$.

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.

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.

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.

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?

Pr(♡♡♡) · Gewinn (1)
+ Pr(◇◇◇) · Gewinn (1/2) + Pr(was anders) · Gewinn (=0)

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

Rechenproblem

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

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

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 }
```

Lineare Suche – Analyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[index]`.

Lineare Suche – Analyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[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.

Lineare Suche – Analyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[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]`).

Lineare Suche – Analyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[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.

Lineare Suche – Analyse

Elementare Operation

Vergleich einer ganzen Zahl `K` mit Element `E[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.

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`".

Lineare Suche – Average-Case-Analyse (I)

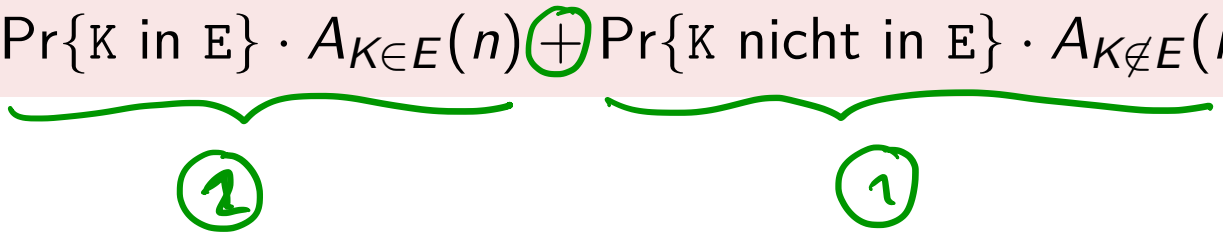
Zwei Szenarien

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`".

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



Der Fall "K in E"

Der Fall "`K` in `E`"

- Angenommen alle Elemente in `E` sind unterschiedlich.

Der Fall "`K` in `E`"

- Angenommen alle Elemente in `E` sind unterschiedlich.
- Damit ist die Wahrscheinlichkeit für `K == E[i]` gleich $\frac{1}{n}$.

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 $i + 1$.

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:

$$A_{K \in E}(n) = \sum_{i=0}^{n-1} \Pr\{\texttt{K == E[i]} \mid \texttt{K in E}\} \cdot t(\texttt{K == E[i]})$$

$\frac{1}{n}$ $\quad$ $i+1$

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:

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

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:

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

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:

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

$\sum_{i=0}^{n} i$

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:

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

Herleitung

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

Herleitung

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

$$\left| \; A_{K \in E}(n) = \frac{n+1}{2} \right.$$

$$= \Pr\{\texttt{K in E}\} \cdot \frac{n+1}{2} + \Pr\{\texttt{K nicht in E}\} \cdot A_{K \notin E}(n)$$

$1 - \Pr\{\texttt{K in E}\}$

Herleitung

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

Herleitung

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

Herleitung

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

Lineare Suche – Average-Case-Analyse

Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

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

Lineare Suche – Average-Case-Analyse

Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

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

Beispiel

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

Lineare Suche – Average-Case-Analyse

Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

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

Beispiel

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

$= 1$, dann ist $A(n) = \frac{n+1}{2}$, d. h. etwa 50% von E ist überprüft.

Lineare Suche – Average-Case-Analyse

Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

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

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.

Lineare Suche – Average-Case-Analyse

Endergebnis

Die Average-Case-Zeitkomplexität der linearen Suche ist:

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

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

Bilineare Suche

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

Bilineare Suche

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

Dies führt zur bilineare 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 – 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 $W(n) = 2 \lceil \frac{n}{2} \rceil$.

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`.

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

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

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.

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 }
```

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

Deutsche Prominenten

Deutsche Prominenten



Deutsche Prominenten




Deutsche Prominenten





Deutsche Prominenten






Deutsche Prominenten





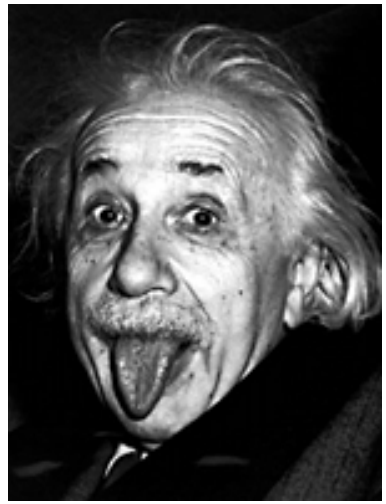


Deutsche Prominenten







Niederländische Prominenten

Das Prominentensuche Problem

Was ist ein Prominenter?

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

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: 1. $n \in \mathbb{N}$ Personen nummeriert $0, \ldots, n-1$

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:
1. $n \in \mathbb{N}$ Personen nummeriert $0, \ldots, n-1$
2. Mindestens eine Person ist ein Prominenter

Das Prominentensuche Problem

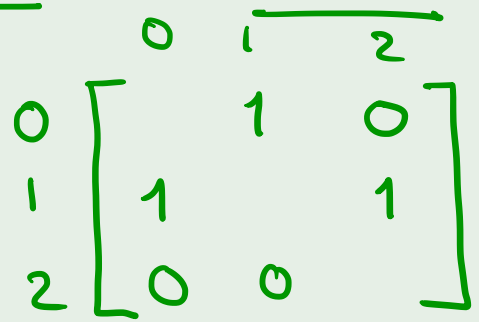
Was ist ein Prominenter?

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

Beispiel (Das Prominentensuche Problem)

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

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



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:
1. $n \in \mathbb{N}$ Personen nummeriert $0, \ldots, n-1$
2. Mindestens eine Person ist ein Prominenter
3. $n \times n$ boolean Matrix K, so dass für $0 \leqslant i, j < n$:

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

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

Das Prominentensuche Problem

Was ist ein Prominenter?

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

Beispiel (Das Prominentensuche Problem)

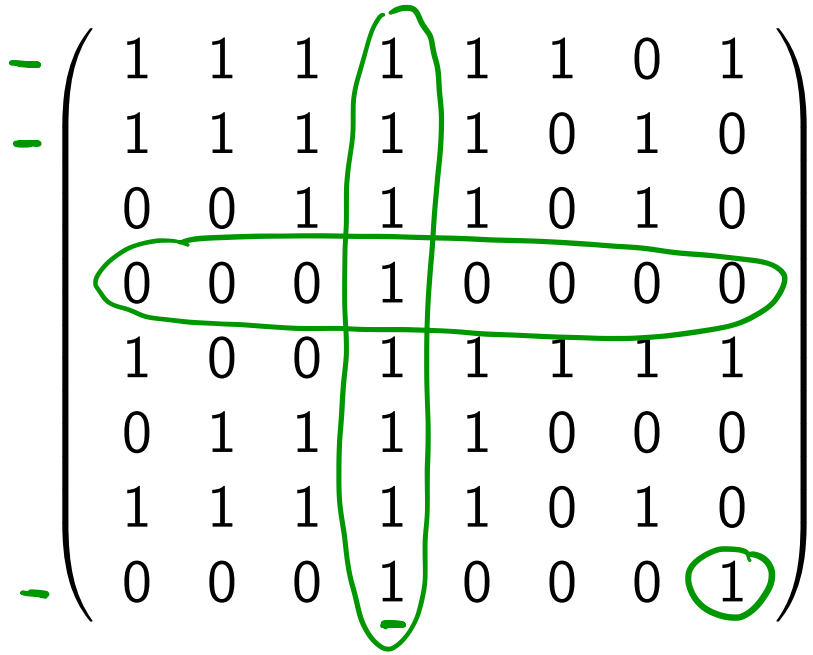
Eingabe:
1. $n \in \mathbb{N}$ Personen nummeriert $0, \ldots, n-1$
2. Mindestens eine Person ist ein Prominenter
3. $n \times n$ boolean Matrix K, so dass für $0 \leqslant i, j < n$:

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

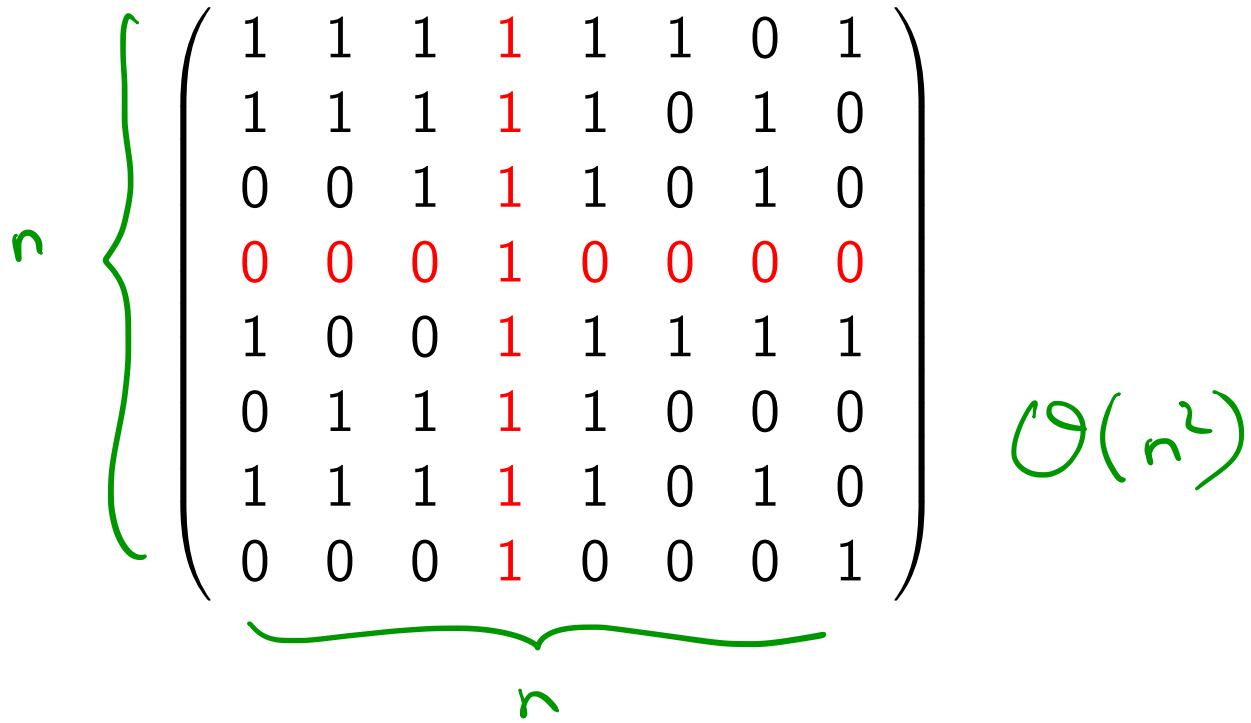
Ausgabe: Sei $k \in \{0, \ldots, n-1\}$, so dass Person k Prominenter ist, d.h.:

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

Beispiel: Wer ist ein Prominenter?



Beispiel: Wer ist ein Prominenter?



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.

Beispiel: Wer ist ein Prominenter?

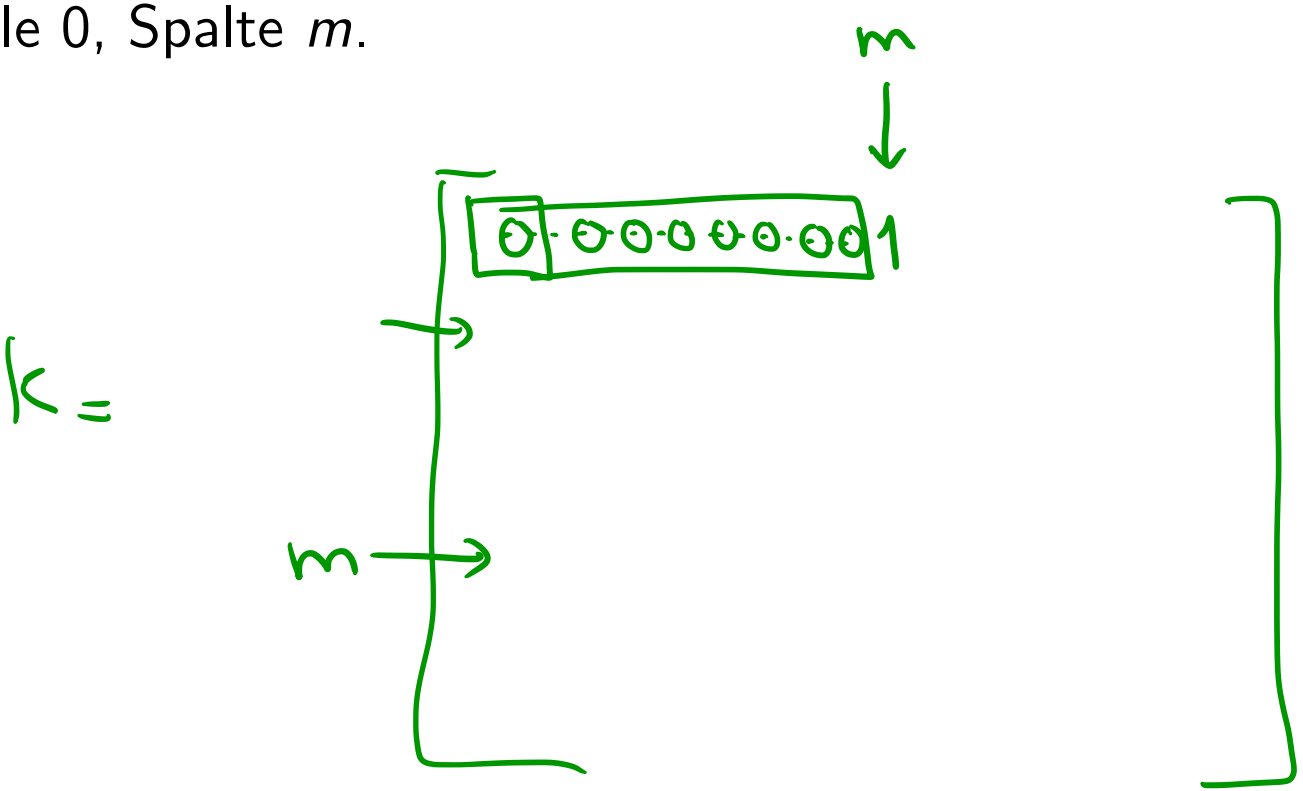
$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

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.



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: $\forall 0 \leqslant k < m.\ k$ ist kein Prominenter.

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: $\forall 0 \leqslant k < m.\, k$ ist kein Prominenter.

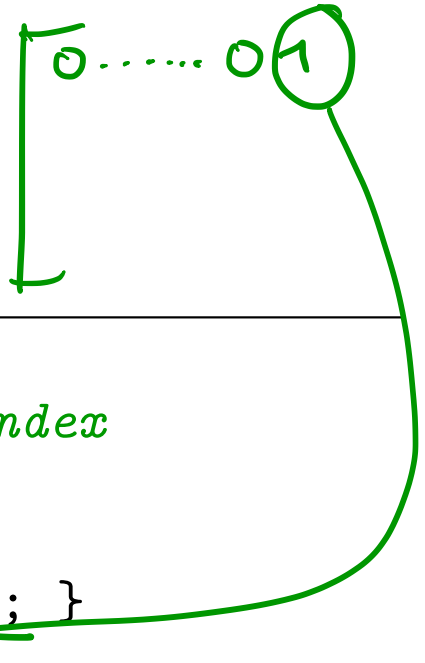
Die Suche geht dann weiter in Zeile m, Spalte m, usw.

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: $\forall 0 \leqslant k < m.\, k$ 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 $0 < i \neq j \leqslant n$ gilt:

1. $K[i, j] \Longrightarrow i$ ist kein Prominenter

≤ 1

Das Prominentensuche Problem: Bilineare Suche

Einige Eigenschaften

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

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

Das Prominentensuche Problem: Bilineare Suche

Einige Eigenschaften

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

1. $K[i,j] \Longrightarrow i$ ist kein Prominenter
2. $\neg K[j,i] \Longrightarrow i$ 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 $A(n), B(n), W(n) \in O(n)$.

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.

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.

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.

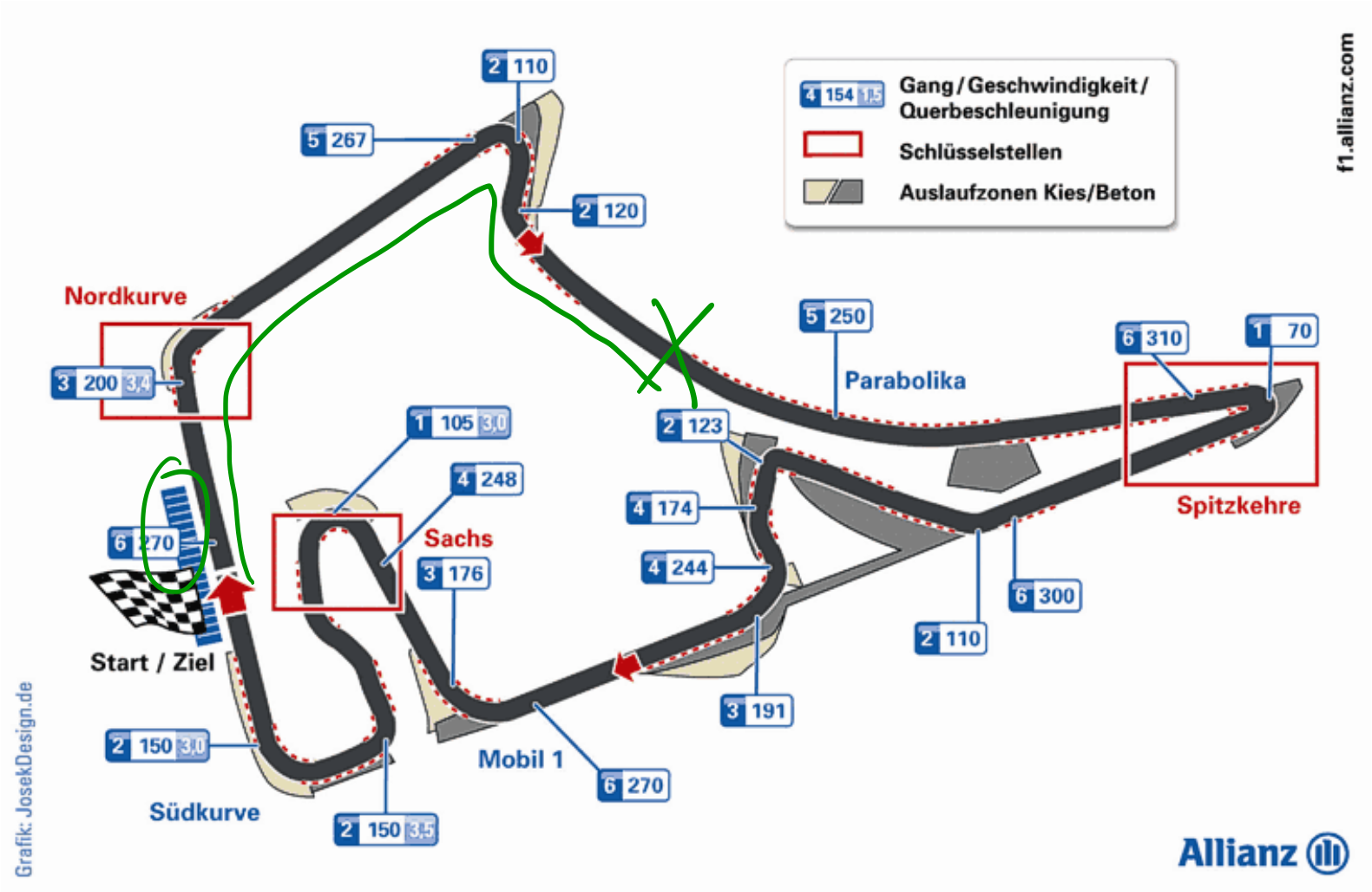
Dies ändert die asymptotische Zeitkomplexität $W(n)$ jedoch nicht.

Hausaufgabe:

Bestimmen Sie die Zeitkomplexität der linearen Suche für dieses Problem.

Das Boxenstopp Problem

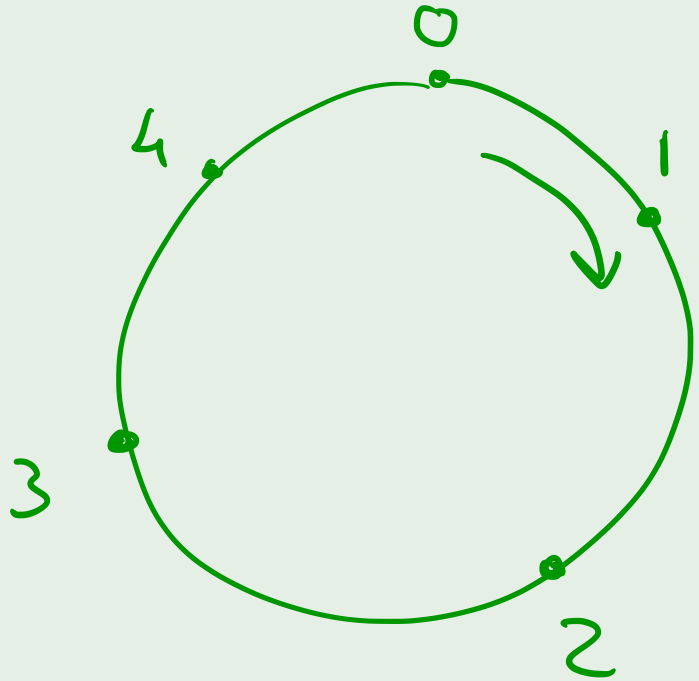
Der Hockenheimring



Das Boxenstopp Problem

Beispiel (Das Boxenstopp Problem)

Eingabe: 1. $n \in \mathbb{N}$ Boxenstopps auf den Hockenheimring, rechts herum nummeriert 0 durch $n-1$.



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

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 $(i+1)$ **mod** n zu fahren braucht man $V(i)$ Liter Benzin

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 $(i+1)$ **mod** n zu fahren braucht man $V(i)$ Liter Benzin
4. Gegeben: $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} V(i)$

$T(0) + T(1) + \ldots + T(n-1) = V(0) + \ldots + V(n-1)$

alles Tankvolume

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 $(i+1)$ **mod** 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 $k \in \{0, \ldots, n-1\}$, so dass Sebastian Vettel mit einem leeren Tank eine komplette Runde fahren kann.

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 $(i+1)$ **mod** n zu fahren braucht man $V(i)$ Liter Benzin
4. Gegeben: $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} V(i)$

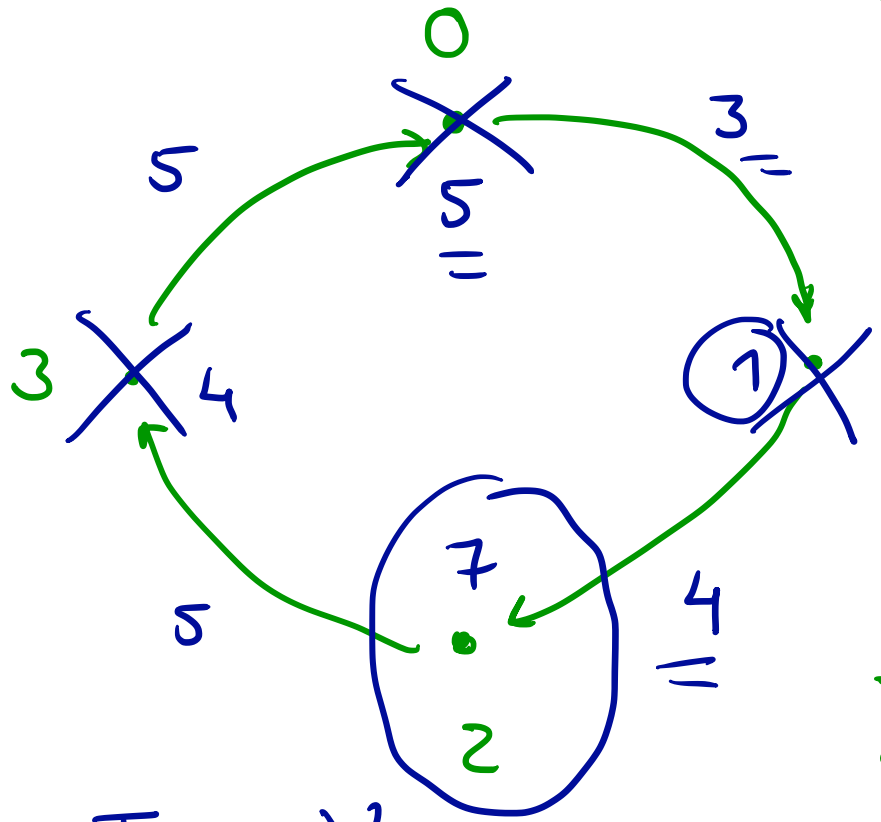
Ausgabe: Bestimme $k \in \{0, \ldots, n-1\}$, so dass Sebastian Vettel mit einem leeren Tank eine komplette Runde fahren kann.

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

Das Boxenstopp Problem: Beispiel

$D =$

	0	1	2	3
0	0	2	-1	1
	-2	0	-3	-1
	1	3	0	2
	-1	1	-2	0



i	T(i)	V(i)
0	5	3
1	1	4
2	7	5
3	4	5
Σ	17	17

start @ 0 :

T	V
~~2~~	3
3	

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.

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.

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

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 k ist starting Boxenstopp gdw. $\forall 0 \leqslant i < n. D[k, i] \geqslant 0$

Das Boxenstopp Problem

Differenzmatrix

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

Starting Boxenstopp

Boxenstopp k ist starting Boxenstopp gdw. $\forall 0 \leqslant i < n.\, D[k,i] \geqslant 0$

Einige Eigenschaften

Das Boxenstopp Problem

Differenzmatrix

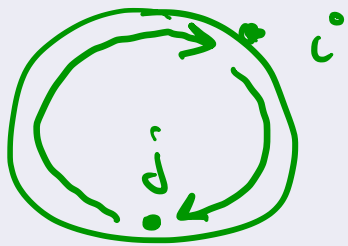
$$D[i,j] = \sum_{m=i}^{j-1} T(m) - V(m).$$

Starting Boxenstopp

Boxenstopp k ist starting Boxenstopp gdw. $\forall 0 \leqslant i < n.\, D[k,i] \geqslant 0$

Einige Eigenschaften

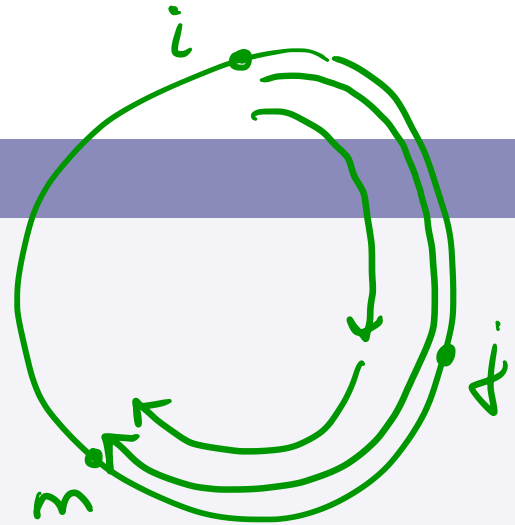
1. Für alle $0 \leqslant i,j < n$ gilt: $D[i,i] = 0$ und $D[i,j] + D[j,i] = 0$



Das Boxenstopp Problem

Differenzmatrix

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



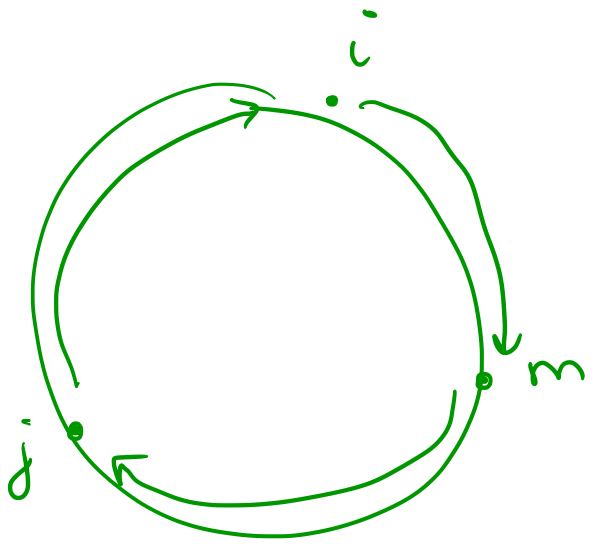
Starting Boxenstopp

Boxenstopp k ist starting Boxenstopp gdw. $\forall 0 \leqslant i < n.\, D[k,i] \geqslant 0$

Einige Eigenschaften

1. Für alle $0 \leqslant i,j < n$ gilt: $D[i,i] = 0$ und $D[i,j] + D[j,i] = 0$
2. Für alle $0 \leqslant i,j,m < n$ gilt: $D[i,m] = D[i,j] + D[j,m]$

2. Szenario



$j \notin \{i, \dots, m-1\}$

$D[i,m] = D[i,m] + 0$

$= D[i,m] + \underbrace{D[i,j] + D[j,i]}_{=0}$

$= D[i,j] + D[i,m] + D[j,i] + 0$

$= \underbrace{D[i,j]}_{①} + \underbrace{D[i,m] + D[j,i]}_{③} + \underbrace{D[m,j] + D[j,m]}_{②}$

$= 0$

$= D[i,j] + D[j,m]$

Das Boxenstopp Problem

Differenzmatrix

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

Starting Boxenstopp

Boxenstopp k ist starting Boxenstopp gdw. $\forall 0 \leqslant i < n.\, D[k,i] \geqslant 0$

Einige Eigenschaften

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

k ist starting Bottleneck

gdw. (Definition)

$\forall\, 0 \le i < n.\quad D[k,i] \ge 0$

$\Leftrightarrow$ (Eigenschaft 2. $D[i,m] = D[i,j] + D[j,m]$)

($i \to k$, $m \to i$, $j \to 0$)

$\forall\, 0 \le i < n.\quad D[k,0] + D[0,i] \ge 0$

$\Leftarrow$ ($D[i,j] = -D[j,i]$)

$\forall\, 0 \le i < n.\quad D[0,i] \ge \underbrace{D[k,0]}_{\text{minimal}}$

Also: $D[k,0]$ ist minimal

Das Boxenstopp Problem

Differenzmatrix

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

Starting Boxenstopp

Boxenstopp k ist starting Boxenstopp gdw. $\forall 0 \leqslant i < n.\, D[k,i] \geqslant 0$

Einige Eigenschaften

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

Invariante: $d = D[left, right]$

Initialisierung: $left = 0$

$right = n-1$

$$d = D[0, n-1] = \sum_{l=0}^{n-2} T(l) - V(l)$$

$$= \sum_{l=0}^{n-1} T(l) - V(l) - \left(T(n-1) - V(n-1)\right)$$

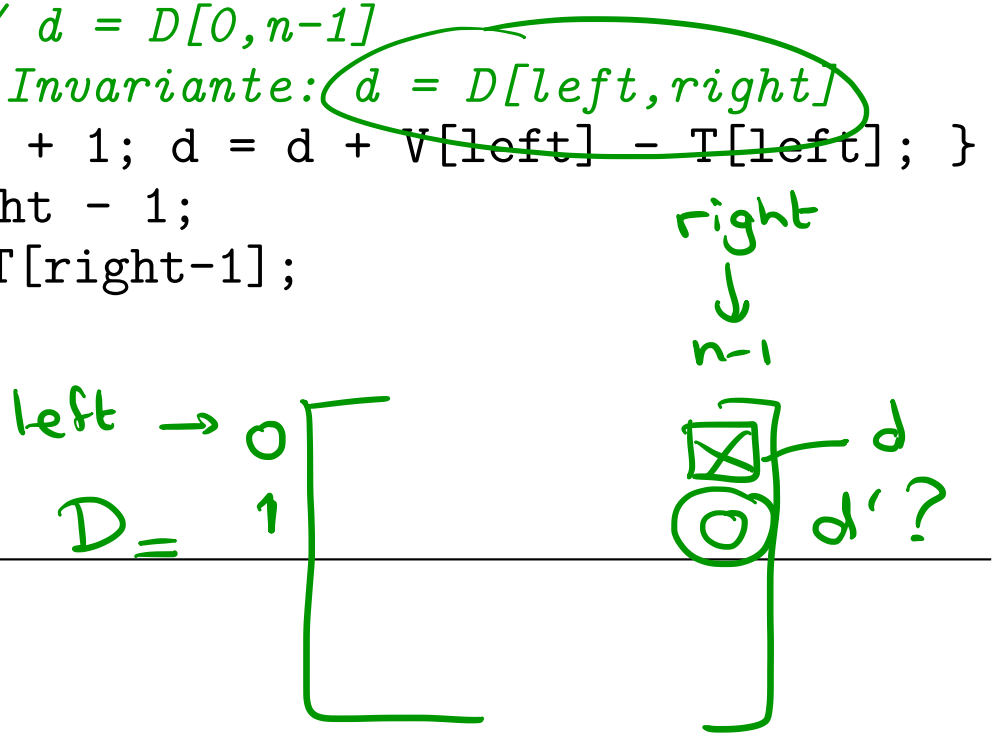
$$= 0$$

$$= V(n-1) - T(n-1)$$

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) { // Invariante: 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 }
```

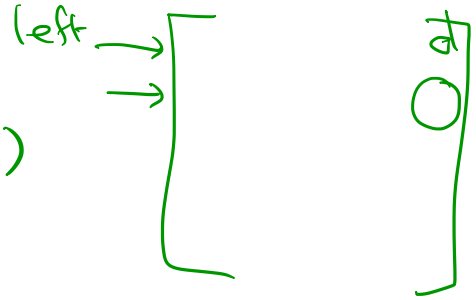


ls. left = left + 1 $\qquad$ d = D[left, right]

$d' = D[left+1, right]$

$= \underbrace{D[left, right]}_{= d} - \big(\underbrace{T(left) - V(left)}_{} \big)$

$$D[left+1, right] = \sum_{i=left+1}^{right-1} T(i) - V(i)$$



$$= \underbrace{\sum_{i=left}^{right-1} T(i) - V(i)}_{= d} - \underbrace{(T(left) - V(left))}_{\boxed{V(left) - T(left)}}$$

Bem: das Programm benutzt nicht der Matrix D. Damit spart man:

- $O(n^2)$ Initialisierungsaufwand
- $O(n^2)$ Speicherplatz

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) { // Invariante: 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.

Übersicht

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?

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.

schneller als $O(n)$

Binäre Suche

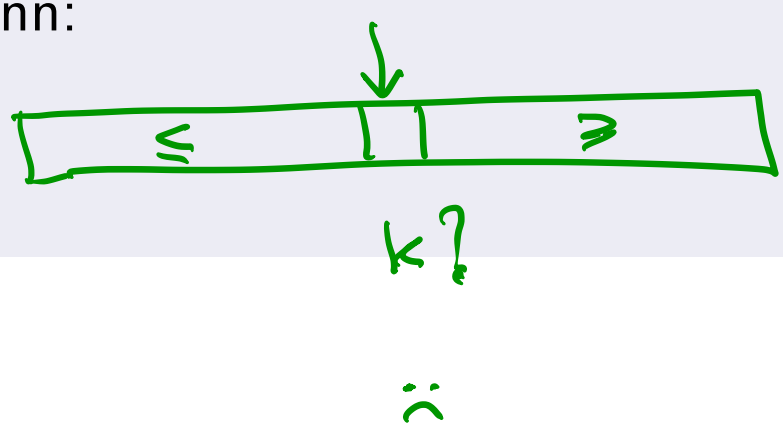
Suchen in einem sortierten Array

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:



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 $\texttt{K} < \texttt{E[mid]}$

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 $K < E[mid]$
2. suche in der rechten Hälfte von `E`, falls $K > E[mid]$

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

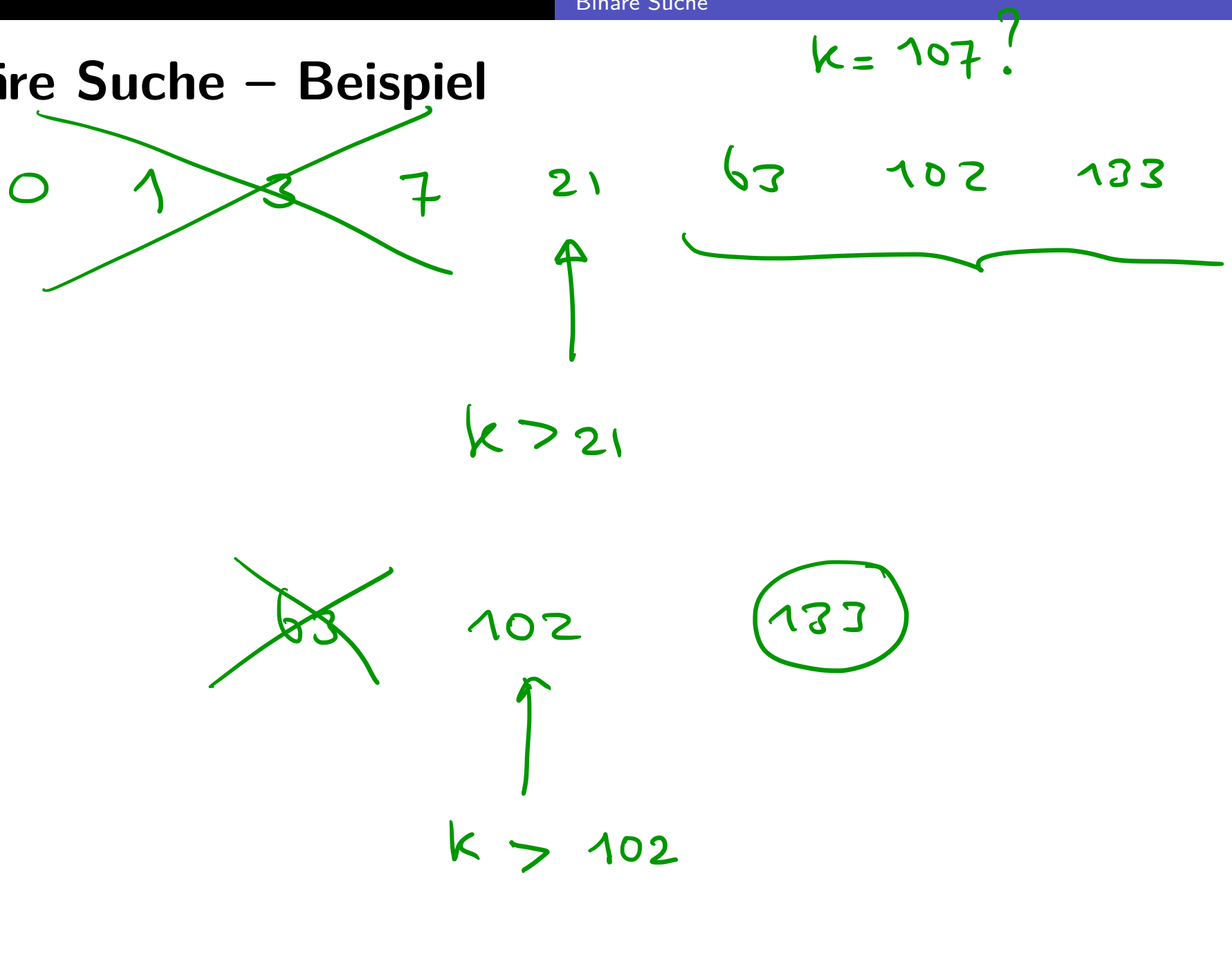
Da `E` sortiert ist, können wir das gesuchte Element `K` schneller suchen.
Liegt `K` nicht in der Mitte von `E`, dann:

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

Fazit:

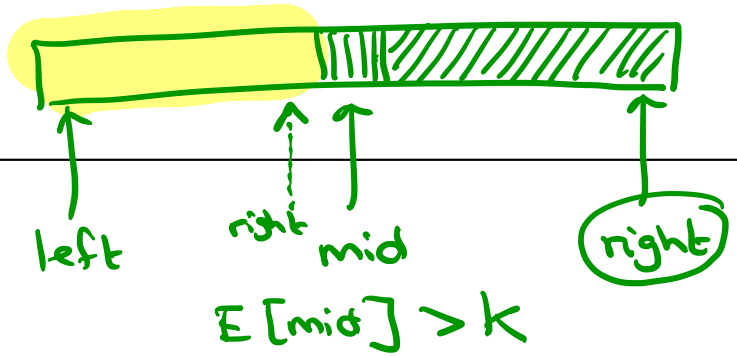
Wir halbieren den Suchraum in jedem Durchlauf.

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:

Binäre Suche – Analyse

Lemma

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

1. $\lfloor r + n \rfloor = \lfloor r \rfloor + n$

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. $\lceil r + n \rceil = \lceil r \rceil + n$

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

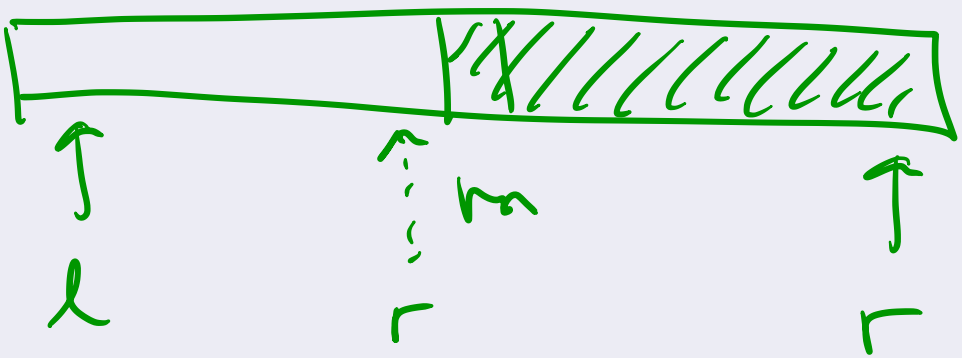
Abkürzungen: $m = \texttt{mid}$, $r = \texttt{right}$, $l = \texttt{left}$

Binäre Suche – Analyse

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

Größe des undurchsuchten Arrays

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



Binäre Suche – Analyse

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

Größe des undurchsuchten Arrays

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

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

Binäre Suche – Analyse

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

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:

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

Binäre Suche – Analyse

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

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:

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

Binäre Suche – Analyse

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

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:

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

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

Binäre Suche – Analyse

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

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:

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

  oder

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

Im schlimmsten Fall ist die neue Größe des Arrays also:

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

Rekursionsgleichung für Binäre Suche

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

Rekursionsgleichung für Binäre Suche

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

Wir erhalten die Rekursionsgleichung:

Rekursionsgleichung für Binäre Suche

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

Wir erhalten die Rekursionsgleichung:

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

vorherige Folie

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

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 $S(n)$

Lösen der Rekursionsgleichung

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

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:

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:

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

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:

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

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:

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

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:

$$S(\underbrace{2^k - 1}_{n}) = 1 + S(\underbrace{2^{k-1} - 1}_{\lceil (n-1)/2 \rceil})$$

$$= 2^{k-1}$$

$$2^{k-1} - 1$$

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:

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

$$S(2^k - 1) = k$$

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:

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

Binäre Suche – Analyse

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

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

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 $2^{k-1} \leqslant n < 2^k$.

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 $2^{k-1} \leqslant n < 2^k$.

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

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 $2^{k-1} \leqslant n < 2^k$.

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

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

Binäre Suche – Analyse

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

Binäre Suche – Analyse

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

Induktion über n:

Binäre Suche – Analyse

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

Induktion über n:

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

$1 \overset{\text{def}}{=} S(1)$

$\lfloor \log 1 \rfloor + 1 = 0 + 1$

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:

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:

$$S(n) = \underbrace{1 + S(\lceil (n-1)/2 \rceil)}_{\text{def } S(n)} =$$

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:

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

ind. hyp

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

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

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

Binäre Suche – Analyse

Theorem

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

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)

Nächste Vorlesung

Nächste Vorlesung

Montag 30. April, ~~09~~:30 (Hörsaal H01). Bis dann!