

4. Suchen

Lineare Suche, Binäre Suche, Interpolationssuche, Exponentielle Suche, Untere Schranken [Ottman/Widmayer, Kap. 3.2, Cormen et al, Kap. 2: Problems 2.1-3,2.2-3,2.3-5]

Das Suchproblem

Gegeben

- Menge von Datensätzen.

Beispiele

Telefonverzeichnis, Wörterbuch, Symboltabelle

- Jeder Datensatz hat einen Schlüssel k .
- Schlüssel sind vergleichbar: eindeutige Antwort auf Frage $k_1 \leq k_2$ für Schlüssel k_1, k_2 .

Aufgabe: finde Datensatz nach Schlüssel k .

Das Auswahlproblem

Gegeben

- Menge von Datensätzen mit vergleichbaren Schlüsseln k .

Gesucht: Datensatz, mit dem kleinsten, grössten, mittleren Schlüssel. Allgemein: finde Datensatz mit i -kleinstem Schlüssel.

Suche in Array

Gegeben

- Array A mit n Elementen ($A[1], \dots, A[n]$).
- Schlüssel b

Gesucht: Index k , $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

22	20	32	10	35	24	42	38	28	41
1	2	3	4	5	6	7	8	9	10

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche:

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche:

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche:

$$\frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}.$$

Suche in sortierten Array

Gegeben

- Sortiertes Array A mit n Elementen $(A[1], \dots, A[n])$ mit $A[1] \leq A[2] \leq \dots \leq A[n]$.
- Schlüssel b

Gesucht: Index k , $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

Divide and Conquer!

Suche $b = 23$.

Divide and Conquer!

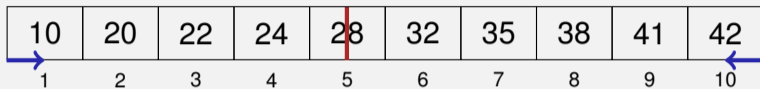
Suche $b = 23$.

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

Divide and Conquer!

Suche $b = 23$.

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

A horizontal array of 10 cells containing the values 10, 20, 22, 24, 28, 32, 35, 38, 41, and 42. Below each cell is its index from 1 to 10. A vertical red line is drawn through the cell containing 28. A blue arrow points from the left edge of the first cell (index 1) to the right edge of the cell containing 28. Another blue arrow points from the right edge of the cell containing 42 to the left edge of the cell containing 28.

$b < 28$

Divide and Conquer!

Suche $b = 23$.

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

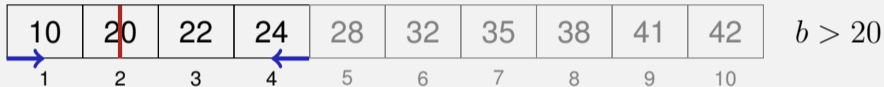
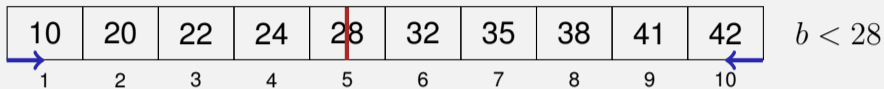
$b < 28$

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

$b > 20$

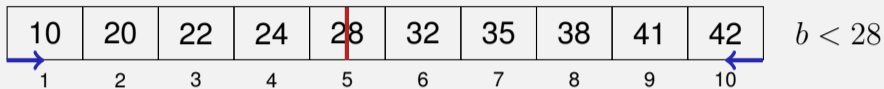
Divide and Conquer!

Suche $b = 23$.



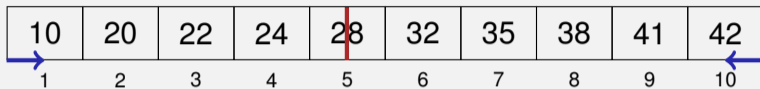
Divide and Conquer!

Suche $b = 23$.



Divide and Conquer!

Suche $b = 23$.



$b < 28$



$b > 20$



$b > 22$



$b < 24$



erfolglos

Binärer Suchalgorithmus $\text{BSearch}(A, b, l, r)$

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 $\text{BSearch}(A, b, l, m - 1)$

else // $b > A[m]$: Element liegt rechts

return $\text{BSearch}(A, b, m + 1, r)$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \end{aligned}$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c. \end{aligned}$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c. \end{aligned}$$

⇒ Annahme: $T(n) = d + c \log_2 n$

Analyse (Schlimmster Fall)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Vermutung : $T(n) = d + c \cdot \log_2 n$

Beweis durch Induktion:

- Induktionsanfang: $T(1) = d$.
- Hypothese: $T(n/2) = d + c \cdot \log_2 n/2$
- Schritt ($n/2 \rightarrow n$)

$$T(n) = T(n/2) + c = d + c \cdot (\log_2 n - 1) + c = d + c \log_2 n.$$

Resultat

Theorem

Der Algorithmus zur binären sortierten Suche benötigt $\Theta(\log n)$ Elementarschritte.

Iterativer binärer Suchalgorithmus

Input : Sortiertes Array A von n Schlüsseln. Schlüssel b .

Output : Index des gefundenen Elements. 0, wenn erfolglos.

$l \leftarrow 1; r \leftarrow n$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $A[m] = b$ **then**

return m

else if $A[m] < b$ **then**

$l \leftarrow m + 1$

else

$r \leftarrow m - 1$

return 0;

Korrektheit

Algorithmus bricht nur ab, falls A leer oder b gefunden.

Invariante: Falls b in A , dann im Bereich $A[l, \dots, r]$

Beweis durch Induktion

- Induktionsanfang: $b \in A[1, \dots, n]$ (oder nicht)
- Hypothese: Invariante gilt nach i Schritten
- Schritt:
 $b < A[m] \Rightarrow b \in A[l, \dots, m - 1]$
 $b > A[m] \Rightarrow b \in A[m + 1, \dots, r]$

Geht es noch besser?

Geht es noch besser?

Annahme: Gleichverteilung der *Werte* im Array.

Geht es noch besser?

Annahme: Gleichverteilung der *Werte* im Array.

Beispiel

Name "Becker" würde man im Telefonbuch vorne suchen.

"Wawrinka" wohl ziemlich weit hinten.

Geht es noch besser?

Annahme: Gleichverteilung der *Werte* im Array.

Beispiel

Name "Becker" würde man im Telefonbuch vorne suchen.

"Wawrinka" wohl ziemlich weit hinten.

Binäre Suche vergleicht immer zuerst mit der Mitte.

Binäre Suche setzt immer $m = \lfloor l + \frac{r-l}{2} \rfloor$.

Interpolationssuche

Erwartete relative Position von b im Suchintervall $[l, r]$

$$\rho = \frac{b - A[l]}{A[r] - A[l]} \in [0, 1].$$

Neue "Mitte": $l + \rho \cdot (r - l)$

Anzahl Vergleiche im Mittel $\mathcal{O}(\log \log n)$ (ohne Beweis).

Interpolationssuche

Erwartete relative Position von b im Suchintervall $[l, r]$

$$\rho = \frac{b - A[l]}{A[r] - A[l]} \in [0, 1].$$

Neue "Mitte": $l + \rho \cdot (r - l)$

Anzahl Vergleiche im Mittel $\mathcal{O}(\log \log n)$ (ohne Beweis).

❓ Ist Interpolationssuche also immer zu bevorzugen?

Interpolationssuche

Erwartete relative Position von b im Suchintervall $[l, r]$

$$\rho = \frac{b - A[l]}{A[r] - A[l]} \in [0, 1].$$

Neue "Mitte": $l + \rho \cdot (r - l)$

Anzahl Vergleiche im Mittel $\mathcal{O}(\log \log n)$ (ohne Beweis).

❓ Ist Interpolationssuche also immer zu bevorzugen?

❗ Nein: Anzahl Vergleiche im schlimmsten Fall $\Omega(n)$.

Exponentielle Suche

Annahme: Schlüssel b liegt eher vorne im Array A . n sehr gross.

Exponentielles Vorgehen:

- 1 Lege Suchbereich $l = 1, r = 1$ fest.
- 2 Verdopple r so lange, bis $r > n$ oder $A[r] > b$.
- 3 Setze $r \leftarrow \min(r, n)$.
- 4 Führe binäre Suche durch mit $l \leftarrow r/2, r$.

Analyse der Exponentiellen Suche

Sei m der gesuchte Index.

Anzahl Schritte für die Verdopplung von r : maximal $\log_2 m$.

Binäre Suche ist dann auch $\mathcal{O}(\log_2 m)$.

Schlechtester Fall insgesamt: Anzahl Schritte $\mathcal{O}(\log_2 n)$.

Analyse der Exponentiellen Suche

Sei m der gesuchte Index.

Anzahl Schritte für die Verdopplung von r : maximal $\log_2 m$.

Binäre Suche ist dann auch $\mathcal{O}(\log_2 m)$.

Schlechtester Fall insgesamt: Anzahl Schritte $\mathcal{O}(\log_2 n)$.

❓ Wann ist dieses Verfahren sinnvoll?

Analyse der Exponentiellen Suche

Sei m der gesuchte Index.

Anzahl Schritte für die Verdopplung von r : maximal $\log_2 m$.

Binäre Suche ist dann auch $\mathcal{O}(\log_2 m)$.

Schlechtester Fall insgesamt: Anzahl Schritte $\mathcal{O}(\log_2 n)$.

❓ Wann ist dieses Verfahren sinnvoll?

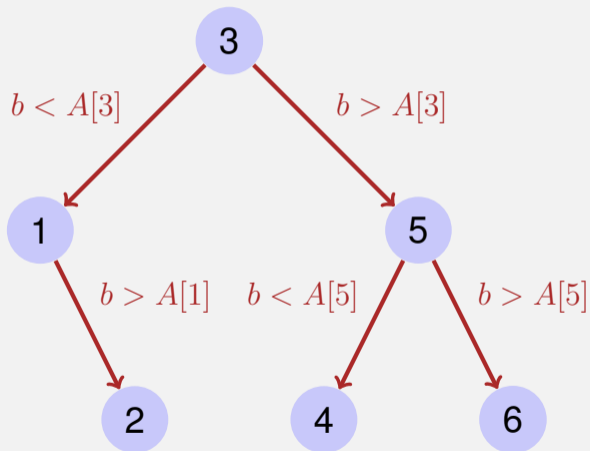
⚠️ Wenn $m \ll n$. Zum Beispiel bei positiven ganzzahligen paarweise verschiedenen Schlüsseln und $b \ll N$ (N : grösster Schlüsselwert).

Untere Schranke

Binäre und exponentielle Suche (im schlechtesten Fall): $\Theta(\log n)$
viele Vergleiche.

Gilt für *jeden* Suchalgorithmus in sortiertem Array (im schlechtesten Fall): Anzahl Vergleiche = $\Omega(\log n)$?

Entscheidungsbaum



- Für jede Eingabe $b = A[i]$ muss Algorithmus erfolgreich sein \Rightarrow Baum enthält mindestens n Knoten.
- Anzahl Vergleiche im schlechtesten Fall = Höhe des Baumes = maximale Anzahl Knoten von Wurzel zu Blatt.

Entscheidungsbaum

Binärer Baum der Höhe h hat höchstens
 $2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1 < 2^h$ Knoten.

Mindestens n Knoten im Entscheidungsbaum mit Höhe h .

$$n < 2^h \Rightarrow h > \log_2 n.$$

Anzahl Entscheidungen = $\Omega(\log n)$.

Theorem

Jeder Algorithmus zur Suche in sortierten Daten der Länge n benötigt im schlechtesten Fall $\Omega(\log n)$ Vergleichsschritte.

Untere Schranke für Suchen in unsortiertem Array

Theorem

Jeder Algorithmus zur Suche in unsortierten Daten der Länge n benötigt im schlechtesten Fall $\Omega(n)$ Vergleichsschritte.

❓ Korrekt?

”Beweis”: Um b in A zu finden, muss b mit jedem Element $A[i]$ ($1 \leq i \leq n$) verglichen werden.

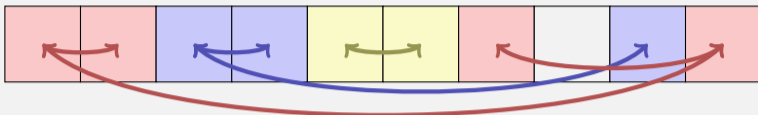
Versuch

❓ Korrekt?

”Beweis”: Um b in A zu finden, muss b mit jedem Element $A[i]$ ($1 \leq i \leq n$) verglichen werden.

❗ Falsch! Vergleiche zwischen Elementen von A möglich!

Besseres Argument



- i Vergleiche ohne b und e Vergleiche mit b
- Vergleiche erzeugen g Gruppen. Initial $g = n$.
- Verbinden zweier Gruppen benötigt mindestens einen Vergleich:
 $n - g \leq i$.
- Mindestens ein Element pro Gruppe muss mit b verglichen werden: $e \geq g$.
- Anzahl Vergleiche $i + e \geq n - g + g = n$.

5. Auswählen

Das Auswahlproblem, Randomisierte Berechnung des Medians,
Lineare Worst-Case Auswahl [Ottman/Widmayer, Kap. 3.1, Cormen
et al, Kap. 9]

Min und Max

② Separates Finden von Minimum und Maximum in $(A[1], \dots, A[n])$ benötigt insgesamt $2n$ Vergleiche. (Wie) geht es mit weniger als $2n$ Vergleichen für beide gemeinsam?

Min und Max

- ② Separates Finden von Minimum und Maximum in $(A[1], \dots, A[n])$ benötigt insgesamt $2n$ Vergleiche. (Wie) geht es mit weniger als $2n$ Vergleichen für beide gemeinsam?
- ① Es geht mit $\frac{3}{2}N$ Vergleichen: Vergleiche jeweils 2 Elemente und deren kleineres mit Min und grösseres mit Max.

Das Auswahlproblem

Eingabe

- Unsortiertes Array $A = (A_1, \dots, A_n)$ paarweise verschiedener Werte
- Zahl $1 \leq k \leq n$.

Ausgabe: $A[i]$ mit $|\{j : A[j] < A[i]\}| = k - 1$

Spezialfälle

$k = 1$: Minimum: Algorithmus mit n Vergleichsoperationen trivial.

$k = n$: Maximum: Algorithmus mit n Vergleichsoperationen trivial.

$k = \lfloor n/2 \rfloor$: Median.

Ansätze

Ansätze

- Wiederholt das Minimum entfernen / auslesen: $\mathcal{O}(k \cdot n)$.
Median: $\mathcal{O}(n^2)$

Ansätze

- Wiederholt das Minimum entfernen / auslesen: $\mathcal{O}(k \cdot n)$.
Median: $\mathcal{O}(n^2)$
- Sortieren (kommt bald): $\mathcal{O}(n \log n)$

Ansätze

- Wiederholt das Minimum entfernen / auslesen: $\mathcal{O}(k \cdot n)$.
Median: $\mathcal{O}(n^2)$
- Sortieren (kommt bald): $\mathcal{O}(n \log n)$
- Pivotieren $\mathcal{O}(n)$!

Pivotieren



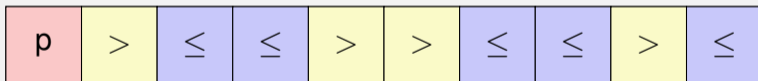
Pivotieren

- 1 Wähle ein Element p als Pivotelement



Pivotieren

- 1 Wähle ein Element p als Pivotelement
- 2 Teile A in zwei Teile auf, den Rang von p bestimmend.



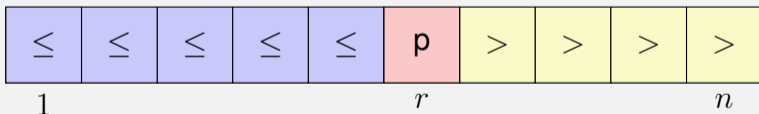
Pivotieren

- 1 Wähle ein Element p als Pivotelement
- 2 Teile A in zwei Teile auf, den Rang von p bestimmend.



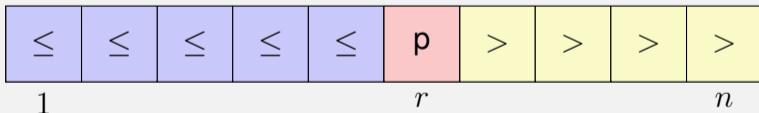
Pivotieren

- 1 Wähle ein Element p als Pivotelement
- 2 Teile A in zwei Teile auf, den Rang von p bestimmend.



Pivotieren

- 1 Wähle ein Element p als Pivotelement
- 2 Teile A in zwei Teile auf, den Rang von p bestimmend.
- 3 Rekursion auf dem relevanten Teil. Falls $k = r$, dann 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 $l-1$

Korrektheit: Invariante

Invariante I : $A_i \leq p \forall i \in [0, l), A_i > p \forall i \in (r, n], \exists k \in [l, r] : A_k = p$.

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**

└ $l \leftarrow l + 1$

I

I und $A[l] \geq p$

I und $A[r] \leq p$

I und $A[l] \leq p \leq A[r]$

I

return $l-1$

Korrektheit: Fortschritt

```
while  $l < r$  do  
  while  $A[l] < p$  do      Fortschritt wenn  $A[l] < p$   
     $l \leftarrow l + 1$   
  while  $A[r] > p$  do      Fortschritt wenn  $A[r] > p$   
     $r \leftarrow r - 1$   
  swap( $A[l], A[r]$ )          Fortschritt wenn  $A[l] > p$  oder  $A[r] < p$   
  if  $A[l] = A[r]$  then      Fortschritt wenn  $A[l] = A[r] = p$   
     $l \leftarrow l + 1$   
return  $l-1$ 
```

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$



Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$



Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$



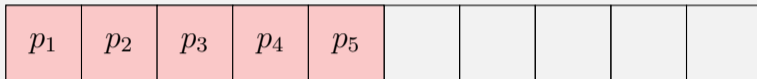
Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$



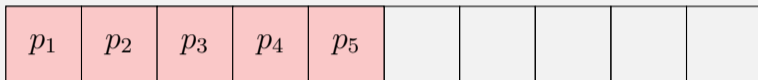
Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$

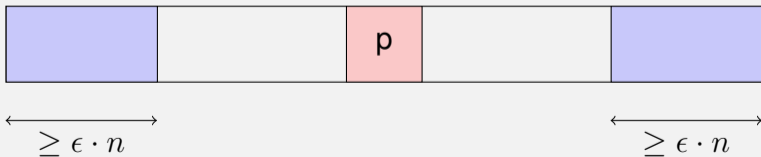


Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$



Ein guter Pivot hat linear viele Elemente auf beiden Seiten.



Analyse

Unterteilung mit Faktor q ($0 < q < 1$): zwei Gruppen mit $q \cdot n$ und $(1 - q) \cdot n$ Elementen (ohne Einschränkung $q \geq 1 - q$).

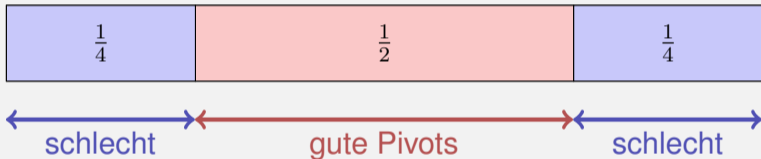
$$T(n) \leq T(q \cdot n) + c \cdot n$$

$$= c \cdot n + q \cdot c \cdot n + T(q^2 \cdot n) = \dots = c \cdot n \sum_{i=0}^{\log_q(n)-1} q^i + T(1)$$

$$\leq c \cdot n \underbrace{\sum_{i=0}^{\infty} q^i}_{\text{geom. Reihe}} = c \cdot n \cdot \frac{1}{1 - q} = \mathcal{O}(n)$$

Wie bekommen wir das hin?

Der Zufall hilft uns (Tony Hoare, 1961). Wähle in jedem Schritt einen zufälligen Pivot.



Wahrscheinlichkeit für guten Pivot nach einem Versuch: $\frac{1}{2} =: \rho$.

Wahrscheinlichkeit für guten Pivot nach k Versuchen: $(1 - \rho)^{k-1} \cdot \rho$.

Erwartungswert der geometrischen Verteilung: $1/\rho = 2$

[Erwartungswert der geometrischen Verteilung]

Zufallsvariable $X \in \mathbb{N}^+$ mit $\mathbb{P}(X = k) = (1 - p)^{k-1} \cdot p$.

Erwartungswert

$$\begin{aligned}\mathbb{E}(X) &= \sum_{k=1}^{\infty} k \cdot (1 - p)^{k-1} \cdot p = \sum_{k=1}^{\infty} k \cdot q^{k-1} \cdot (1 - q) \\ &= \sum_{k=1}^{\infty} k \cdot q^{k-1} - k \cdot q^k = \sum_{k=0}^{\infty} (k + 1) \cdot q^k - k \cdot q^k \\ &= \sum_{k=0}^{\infty} q^k = \frac{1}{1 - q} = \frac{1}{p}.\end{aligned}$$

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 $x \in A[l..r]$

$p \leftarrow l$

for $j = l$ **to** r **do**

if $A[j] \leq x$ **then** $p \leftarrow p + 1$

until $\frac{l+r}{4} \leq p \leq \frac{3(l+r)}{4}$

$m \leftarrow \text{Partition}(A[l..r], x)$

if $i < m$ **then**

 quickselect($A[l..m], i$)

else

 quickselect($A[m..r], i$)

Median der Mediane

Ziel: Finde einen Algorithmus, welcher im schlechtesten Fall nur linear viele Schritte benötigt.

Algorithmus Select (k -smallest)

- Fünfergruppen bilden.
- Median jeder Gruppe bilden (naiv).
- Select rekursiv auf den Gruppenmedianen.
- Partitioniere das Array um den gefundenen Median der Mediane.
Resultat: i
- Wenn $i = k$, Resultat. Sonst: Select rekursiv auf der richtigen Seite.

Median der Mediane



Median der Mediane



1 Fünfergruppen

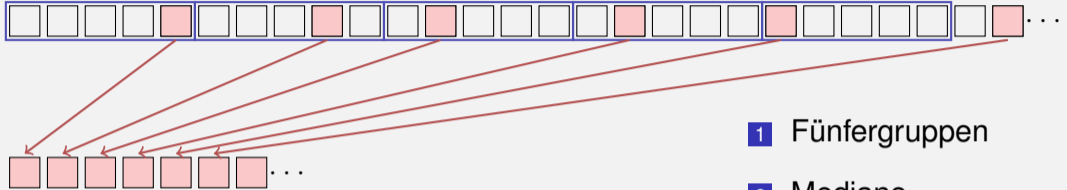
Median der Mediane



1 Fünfergruppen

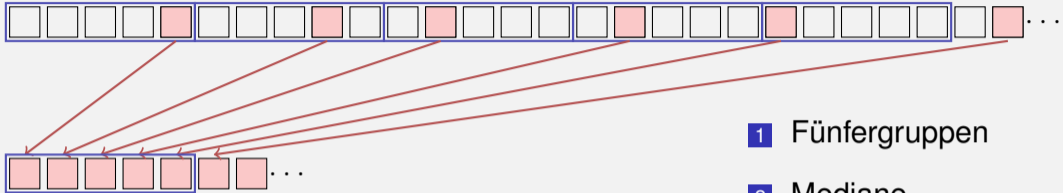
2 Mediane

Median der Mediane



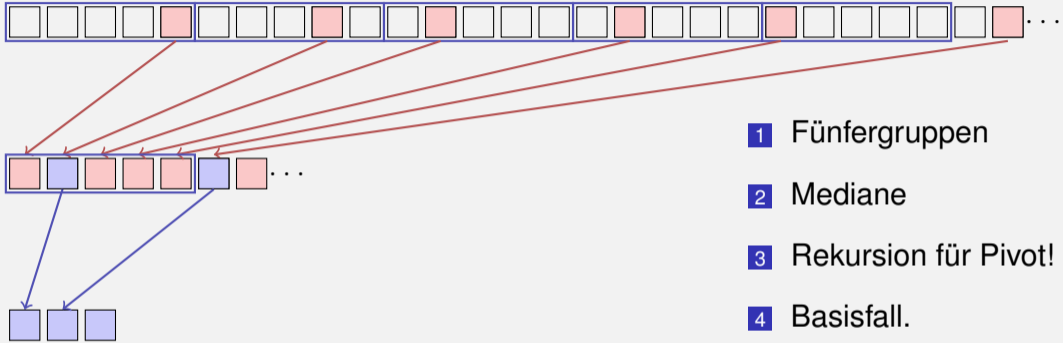
- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!

Median der Mediane

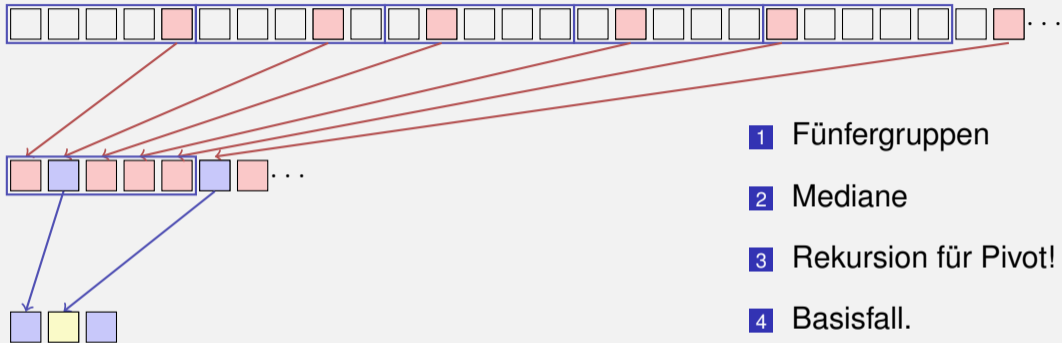


- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!

Median der Mediane

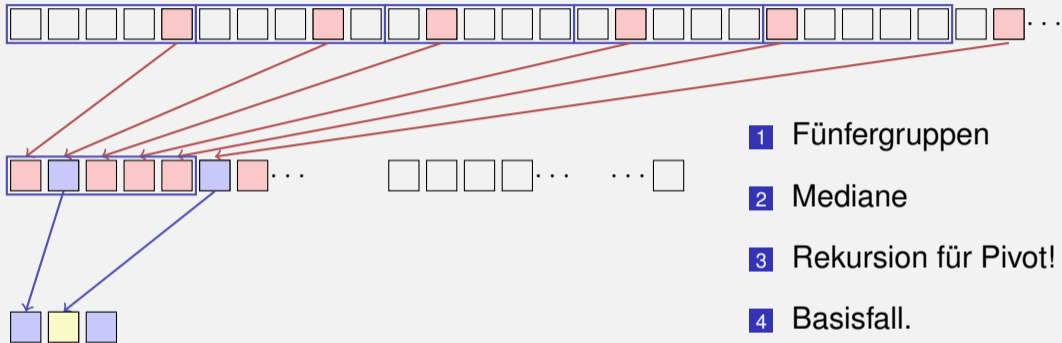


Median der Mediane



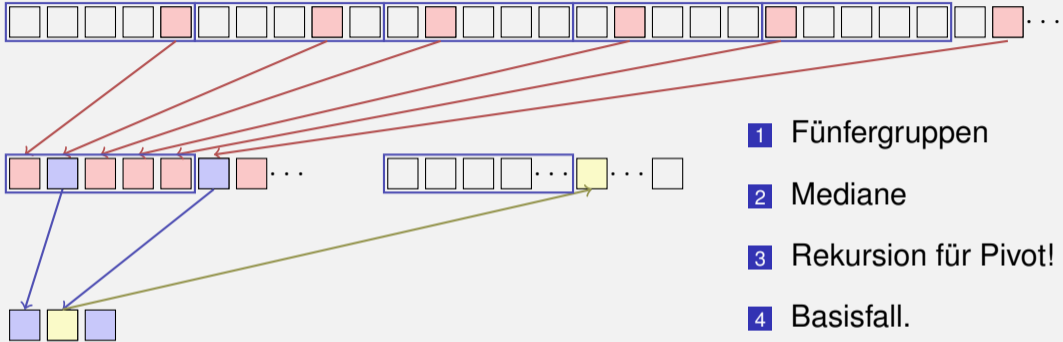
- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!
- 4 Basisfall.
- 5 Pivot (Level 1)

Median der Mediane

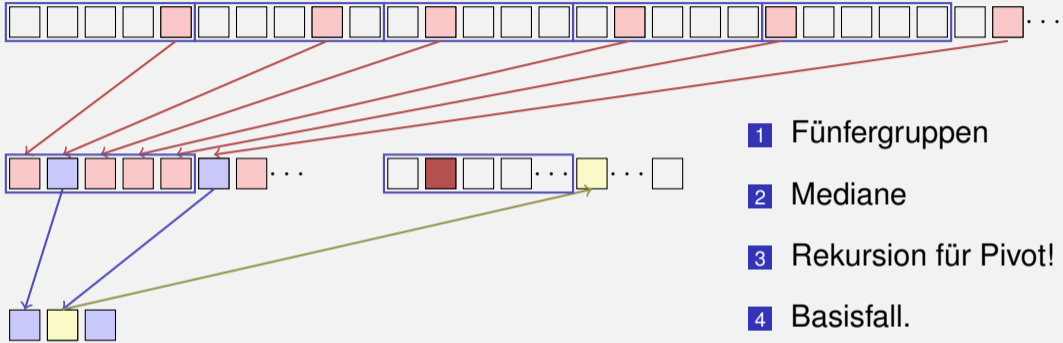


- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!
- 4 Basisfall.
- 5 Pivot (Level 1)

Median der Mediane

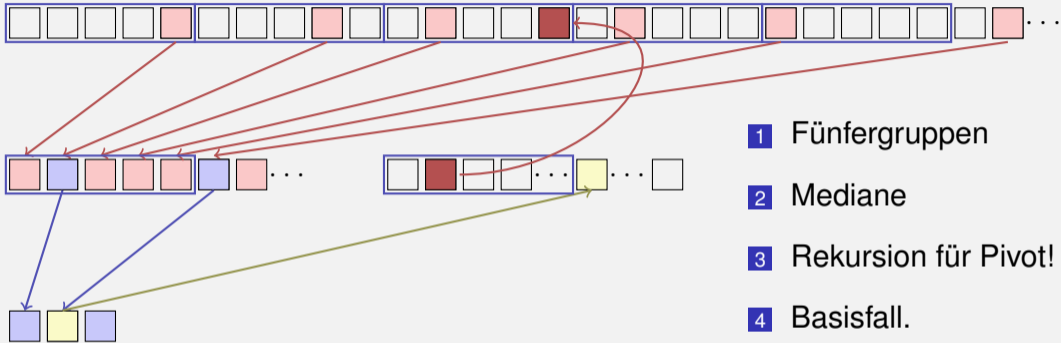


Median der Mediane



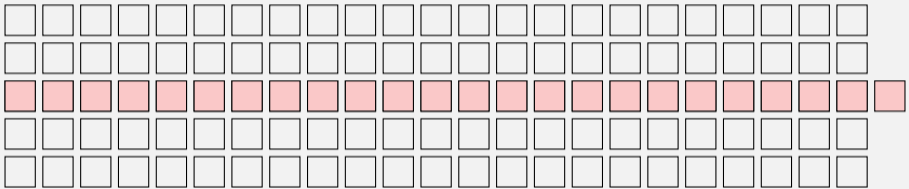
- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!
- 4 Basisfall.
- 5 Pivot (Level 1)
- 6 Partition (Level 1)
- 7 Median = Pivot Level 0

Median der Mediane

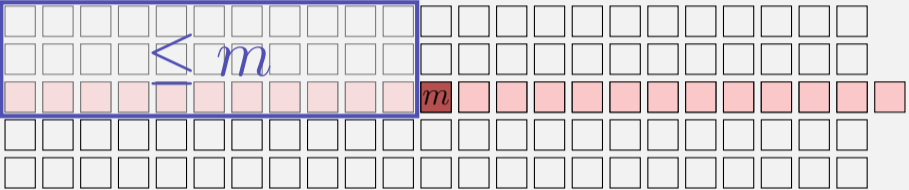


- 1 Fünfergruppen
- 2 Mediane
- 3 Rekursion für Pivot!
- 4 Basisfall.
- 5 Pivot (Level 1)
- 6 Partition (Level 1)
- 7 Median = Pivot Level 0
- 8 2. Rekursion startet

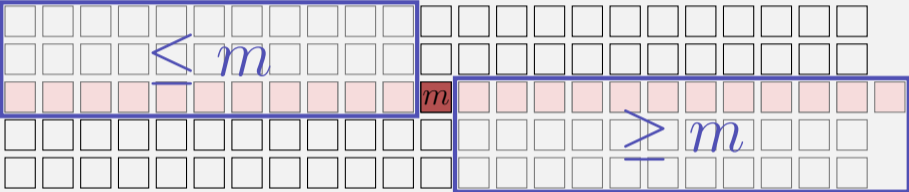
Was bringt das?



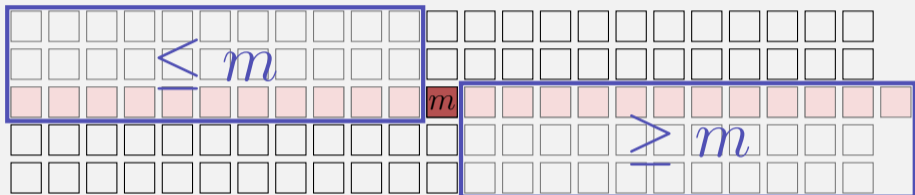
Was bringt das?



Was bringt das?

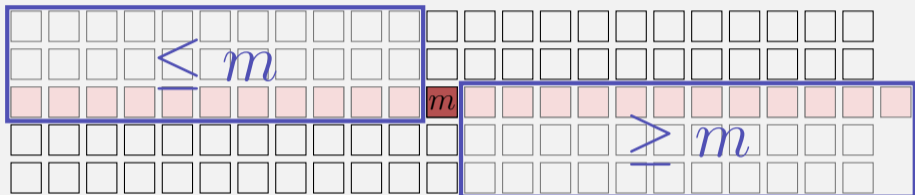


Was bringt das?



Anzahl Punkte links / rechts vom Median der Mediane (ohne
Mediangruppe und ohne Restgruppe) $\geq 3 \cdot (\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$

Was bringt das?



Anzahl Punkte links / rechts vom Median der Mediane (ohne Mediengruppe und ohne Restgruppe) $\geq 3 \cdot (\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$

Zweiter Aufruf mit maximal $\lceil \frac{7n}{10} + 6 \rceil$ Elementen.

Analyse

Rekursionsungleichung:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n.$$

mit einer Konstanten d .

Behauptung:

$$T(n) = \mathcal{O}(n).$$

Beweis

Induktionsanfang: Wähle c so gross, dass

$$T(n) \leq c \cdot n \text{ für alle } n \leq n_0.$$

Induktionsannahme:

$$T(i) \leq c \cdot i \text{ für alle } i < n.$$

Induktionsschritt:

$$\begin{aligned} T(n) &\leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n \\ &= c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n. \end{aligned}$$

Beweis

Induktionsschritt:

$$\begin{aligned} T(n) &\leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n \\ &\leq c \cdot \frac{n}{5} + c + c \cdot \frac{7n}{10} + 6c + c + d \cdot n = \frac{9}{10} \cdot c \cdot n + 8c + d \cdot n. \end{aligned}$$

Wähle $c \geq 80 \cdot d$ und $n_0 = 91$.

$$T(n) \leq \frac{72}{80} \cdot c \cdot n + 8c + \frac{1}{80} \cdot c \cdot n = c \cdot \underbrace{\left(\frac{73}{80}n + 8 \right)}_{\leq n \text{ für } n > n_0} \leq c \cdot n.$$

Theorem

Das i -te Element einer Folge von n Elementen kann in höchstens $\mathcal{O}(n)$ Schritten gefunden werden.

Überblick

- | | |
|-------------------------------------|--------------------------------------|
| 1. Wiederholt Minimum finden | $\mathcal{O}(n^2)$ |
| 2. Sortieren und $A[i]$ ausgeben | $\mathcal{O}(n \log n)$ |
| 3. Quickselect mit zufälligem Pivot | $\mathcal{O}(n)$ im Mittel |
| 4. Median of Medians (Blum) | $\mathcal{O}(n)$ im schlimmsten Fall |

