

## 3. Suchen

# 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  $k$ .

# 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".

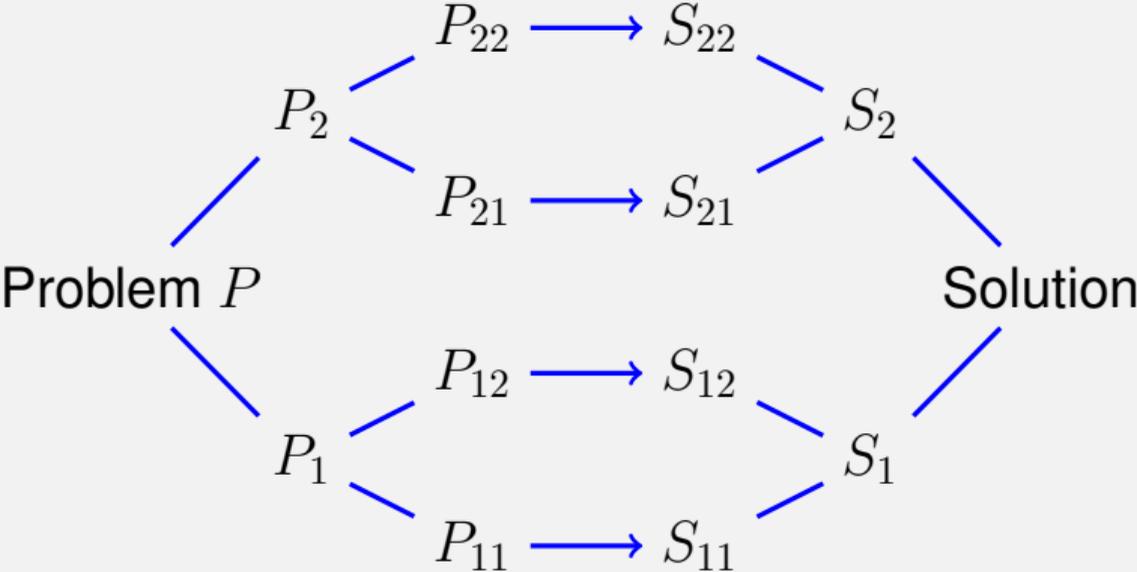
10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

# divide et impera

Teile und (be)herrsche (engl. divide and conquer)

Zerlege das Problem in Teilprobleme, deren Lösung zur vereinfachten Lösung des Gesamtproblems beitragen.

# divide et impera



# Divide and Conquer!

Suche  $b = 23$ .

# Divide and Conquer!

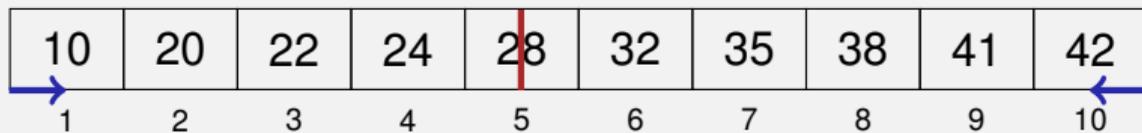
Suche  $b = 23$ .

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

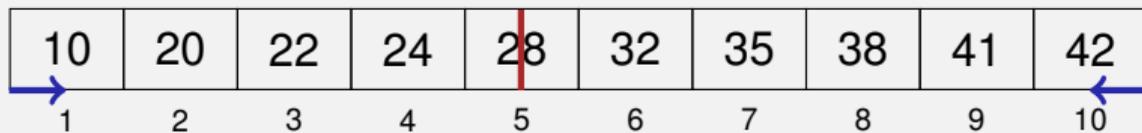
A horizontal array of 10 cells containing the values 10, 20, 22, 24, 28, 32, 35, 38, 41, and 42. Below each cell is its index from 1 to 10. A vertical red line is drawn through the cell containing 28. A blue arrow points from the left edge of the first cell (index 1) to the right edge of the cell containing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

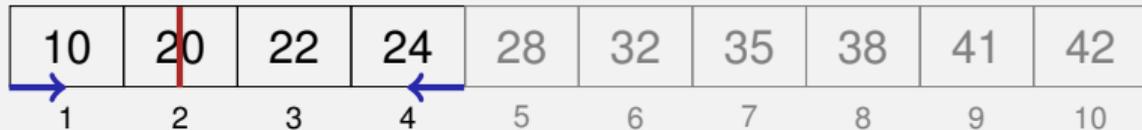
Suche  $b = 23$ .

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  $b = 23$ .

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

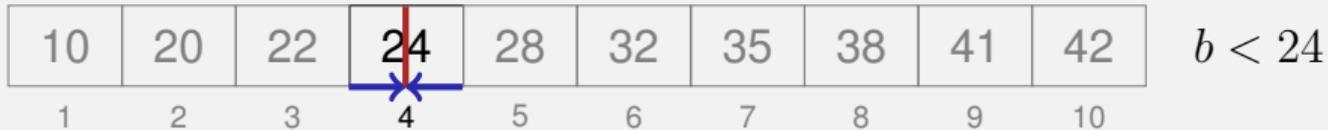
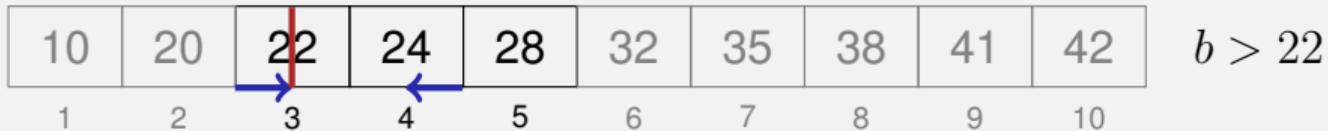
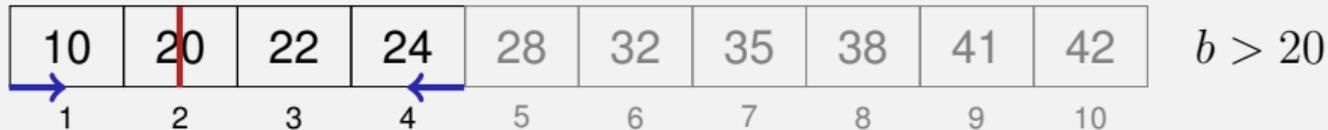
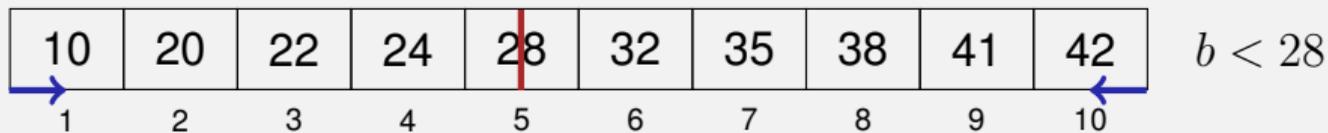
$b > 20$

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

$b > 22$

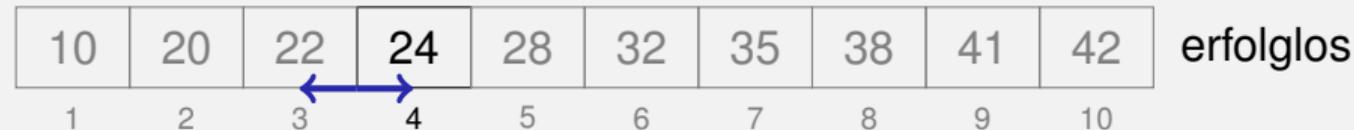
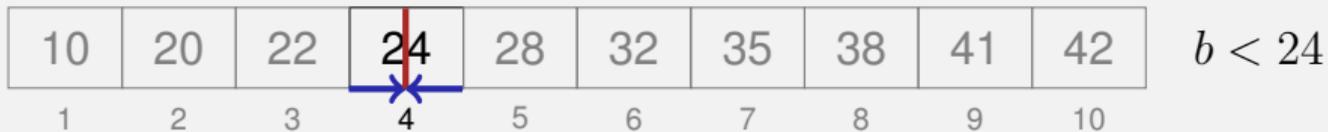
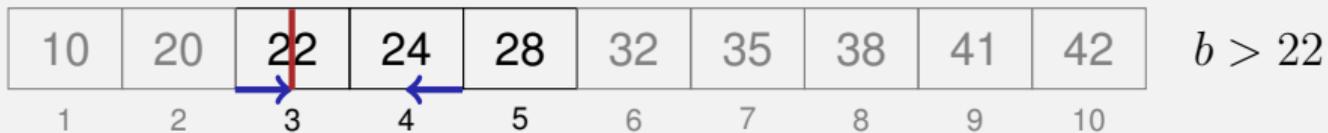
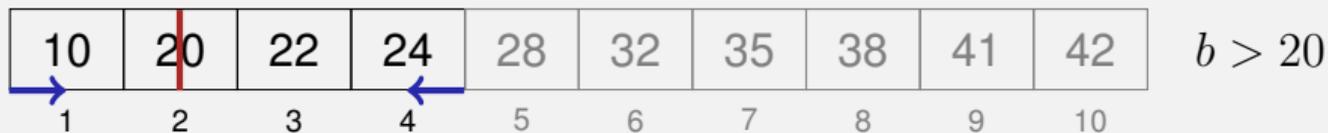
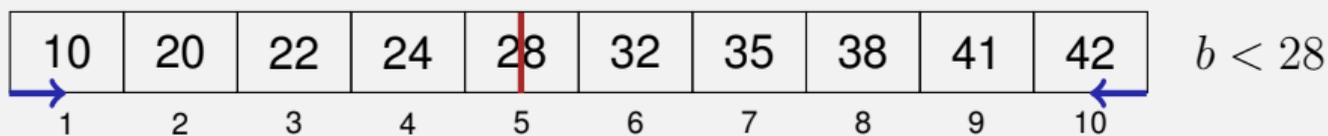
# Divide and Conquer!

Suche  $b = 23$ .



# Divide and Conquer!

Suche  $b = 23$ .



# Binärer Suchalgorithmus

## BSearch( $A[l..r]$ , $b$ )

**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** *NotFound*

**else if**  $b = A[m]$  **then** // gefunden

**return**  $m$

**else if**  $b < A[m]$  **then** // Element liegt links

**return** BSearch( $A[l..m - 1]$ ,  $b$ )

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

**return** BSearch( $A[m + 1..r]$ ,  $b$ )

# 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:

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

# 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:

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

⇒ 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:**

# 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:  $T(1) = d$ .

# 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:  $T(n/2) = d + c \cdot \log_2 n/2$

# 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** *NotFound*;

# Korrektheit

Algorithmus bricht nur ab, falls  $A[l..r]$  leer oder  $b$  gefunden.

**Invariante:** Falls  $b$  in  $A$ , dann im Bereich  $A[l..r]$

## Beweis durch Induktion

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

## 4. Auswählen

# 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.



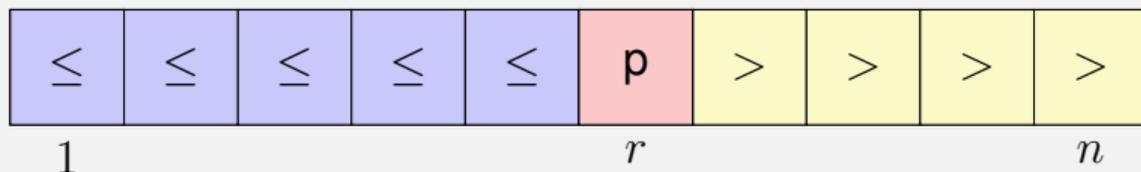
# Pivotieren

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



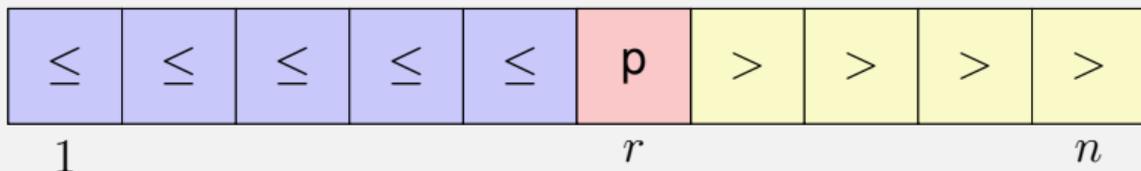
# Pivotieren

- 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 Pivot  $p$  im Intervall  $[l, r]$  mindestens einmal enthält.

**Output :** Array  $A$  partitioniert in  $[l..r]$  um  $p$ . Rückgabe der Position von  $p$ .

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

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

**while**  $A[l] < p$  **do**

$l \leftarrow l + 1$

**while**  $A[r] > p$  **do**

$r \leftarrow r - 1$

$\text{swap}(A[l], A[r])$

**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  $l \leq r$  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  $\Theta(n^2)$



# 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  $\Theta(n^2)$



# Wahl des Pivots

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



# Wahl des Pivots

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

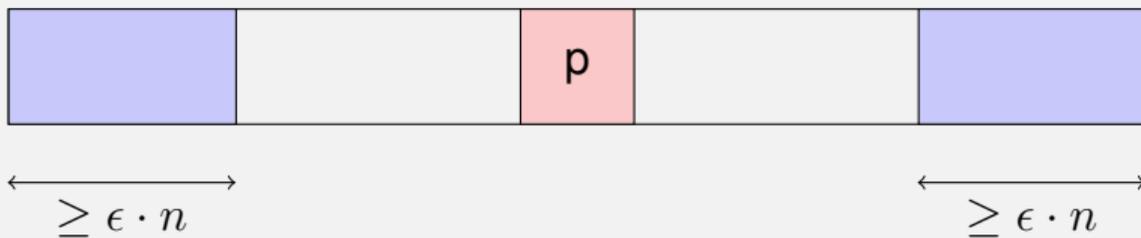


# Wahl des Pivots

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  $q \geq 1 - q$ ).

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

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

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

# 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:  $1/\rho = 2$

# Algorithmus Quickselect ( $A[l..r], k$ )

**Input :** Array  $A$  der Länge  $n$ . Indizes  $1 \leq l \leq k \leq r \leq n$ , so dass für alle

$x \in A[l..r] : |\{j|A[j] \leq x\}| \geq l$  und  $|\{j|A[j] \leq x\}| \leq r$ .

**Output :** Wert  $x \in A[l..r]$  mit  $|\{j|A[j] \leq x\}| \geq k$  und  $|\{j|x \leq A[j]\}| \geq n - k + 1$

**if**  $l=r$  **then**

  | return  $A[l]$ ;

$x \leftarrow \text{RandomPivot}(A[l..r])$

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

**if**  $k < m$  **then**

  | return QuickSelect( $A[l..m - 1], k$ )

**else if**  $k > m$  **then**

  | return QuickSelect( $A[m + 1..r], k$ )

**else**

  | **return**  $A[k]$

# Algorithmus RandomPivot ( $A[l..r]$ )

**Input** : Array  $A$  der Länge  $n$ . Indizes  $1 \leq l \leq i \leq r \leq n$

**Output** : Zufälliger “guter” Pivot  $x \in A[l..r]$

**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**  $\lfloor \frac{3l+r}{4} \rfloor \leq p \leq \lceil \frac{l+3r}{4} \rceil$

**return**  $x$

*Dieser Algorithmus ist nur von theoretischem Interesse und liefert im Erwartungswert nach 2 Durchläufen einen guten Pivot. Praktisch kann man im Algorithmus Quickselect direkt einen zufälligen Pivot uniformverteilt ziehen oder einen deterministischen Pivot wählen, z.B. den Median von drei Elementen.*