# 3. Suchen

Lineare Suche, Binäre Suche [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 $k$.



## Suche in Array

Gegeben

- Array $A$ mit $n$ Elementen $(A[1], \ldots, A[n])$.
- 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 $A[n]$.

- *Bestenfalls* $1$ Vergleich.
- *Schlimmstenfalls* $n$ Vergleiche.
- Annahme: Jede Anordnung der $n$ Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche für die erfolgreiche Suche:

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

## Suche im sortierten Array

Gegeben

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

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

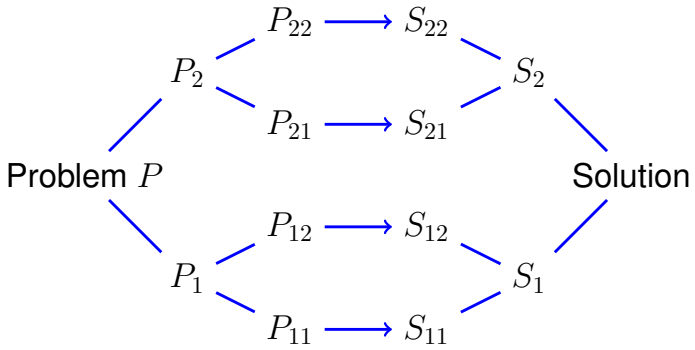
| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |



## 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 and Conquer!

Suche $b = 23$.

| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 | $b < 28$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 | $b > 20$ |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 | $b > 22$ |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 | $b < 24$ |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 | erfolglos |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |



## 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 (schlechtester 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: [2]

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

[2] Versuche eine geschlossene Form zu finden, indem die Rekurrenz, ausgehend von $T(n)$, wiederholt eingesetzt wird.



## Analyse (schlechtester 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 ($n/2 \to n$)

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



## 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*;

# 4. Sortieren

Einfache Sortierverfahren, Quicksort, Mergesort



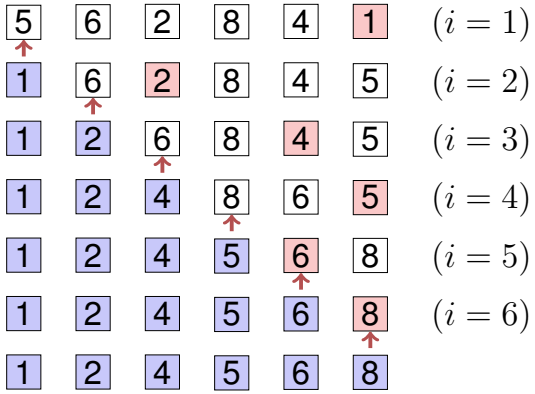
## Problemstellung

**Eingabe:** Ein Array $A = (A[1], ..., A[n])$ der Länge $n$.

**Ausgabe:** Eine Permutation $A'$ von $A$, die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.



## Sortieren durch Auswahl




- Auswahl des kleinsten Elementes durch Suche im unsortierten Teil $A[i..n]$ des Arrays.
- Tausche kleinstes Element an das erste Element des unsortierten Teiles.
- Unsortierter Teil wird ein Element kleiner ($i \to i + 1$). Wiederhole bis alles sortiert. ($i = n$)



## Algorithmus: Sortieren durch Auswahl

**Input**: Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
**Output**: Sortiertes Array $A$

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

## Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$



# 4.1 Mergesort

[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],



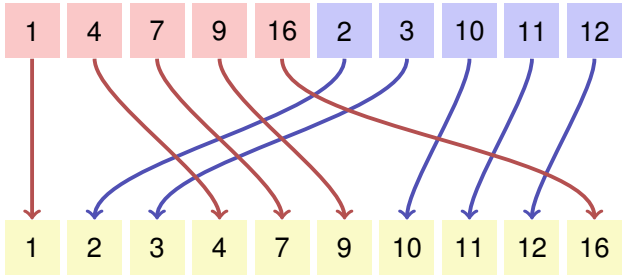
## Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays $A$ bereits sortiert.
- Folgerung: Minimum von $A$ kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Füge die beiden vorsortierten Hälften von $A$ zusammen in $\mathcal{O}(n)$.



## Merge



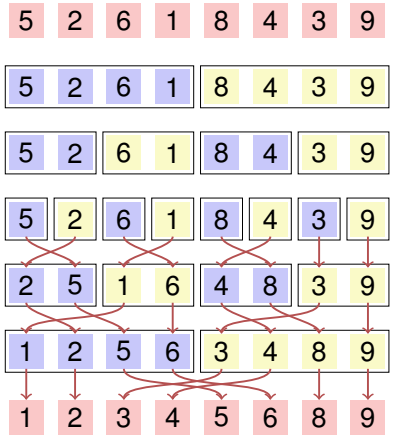
## Algorithmus Merge($A, l, m, r$)

**Input**: Array $A$ der Länge $n$, Indizes $1 \leq l \leq m \leq r \leq n$.
$A[l, \ldots, m]$, $A[m+1, \ldots, r]$ sortiert
**Output**: $A[l, \ldots, r]$ sortiert

```
1 B ← new Array(r − l + 1)
2 i ← l; j ← m + 1; k ← 1
3 while i ≤ m and j ≤ r do
4     if A[i] ≤ A[j] then B[k] ← A[i]; i ← i + 1
5     else B[k] ← A[j]; j ← j + 1
6     k ← k + 1;
7 while i ≤ m do B[k] ← A[i]; i ← i + 1; k ← k + 1
8 while j ≤ r do B[k] ← A[j]; j ← j + 1; k ← k + 1
9 for k ← l to r do A[k] ← B[k − l + 1]
```



## Mergesort






## Algorithmus (Rekursives 2-Wege) Mergesort($A, l, r$)

**Input**: Array $A$ der Länge $n$. $1 \leq l \leq r \leq n$
**Output**: Array $A[l, \ldots, r]$ sortiert.

```
if l < r then
    m ← ⌊(l + r)/2⌋          // Mittlere Position
    Mergesort(A, l, m)       // Sortiere vordere Hälfte
    Mergesort(A, m + 1, r)   // Sortiere hintere Hälfte
    Merge(A, l, m, r)        // Verschmelzen der Teilfolgen
```



## Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + \Theta(n) \in \Theta(n \log n)$$

## Herleitung für $n = 2^k$

Sei $n = 2^k$, $k > 0$. Rekurrenz

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

Teleskopieren

$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn \\ &= 2(2(T(n/8) + cn/4) + cn/2) + cn = ... \\ &= 2(2(...(2(2T(n/2^k) + cn/2^{k-1})...) + cn/2^2) + cn/2^1) + cn \\ &= 2^k T(1) + \underbrace{2^{k-1}cn/2^{k-1} + 2^{k-2}cn/2^{k-2} + ... + 2^{k-k}cn/2^{k-k}}_{k\text{Terme}} \\ &= nd + cnk = nd + cn\log_2 n \in \Theta(n \log n). \end{aligned}$$



# 4.2 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]



## Quicksort

**?** Was ist der Nachteil von Mergesort?

**!** Benötigt zusätzlich $\Theta(n)$ Speicherplatz für das Verschmelzen.

**?** Wie könnte man das Verschmelzen einsparen?

**!** Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

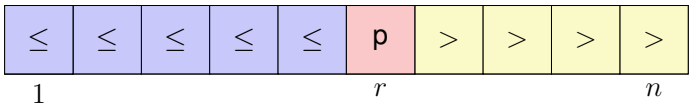
**?** Wie?

**!** Pivotieren und Aufteilen!



## Pivotieren

1. Wähle ein (beliebiges) Element $p$ als Pivotelement
2. Teile $A$ in zwei Teile auf: einen Teil $L$ der Elemente mit $A[i] \leq p$ und einen Teil $R$ der Elemente mit $A[i] > p$.
3. Quicksort: Rekursion auf Teilen L und R



## 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 ≤ r do
    while A[l] < p do
        l ← l + 1
    while A[r] > p do
        r ← r − 1
    swap(A[l], A[r])
    if A[l] = A[r] then
        l ← l + 1
return l-1
```



## Algorithmus Quicksort($A[l, \ldots, r]$

**Input**: Array $A$ der Länge $n$. $1 \leq l \leq r \leq n$.
**Output**: Array $A$, sortiert zwischen $l$ und $r$.

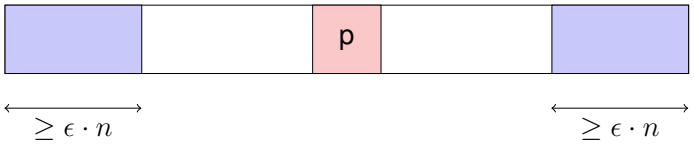
```
if l < r then
    Wähle Pivot p ∈ A[l, ..., r]
    k ← Partition(A[l, ..., r], p)
    Quicksort(A[l, ..., k − 1])
    Quicksort(A[k + 1, ..., r])
```



## Wahl des Pivots

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

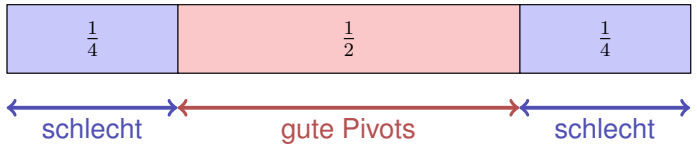

Ein guter Pivot hat linear viele Elemente auf beiden Seiten.






## Wahl des Pivots?

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



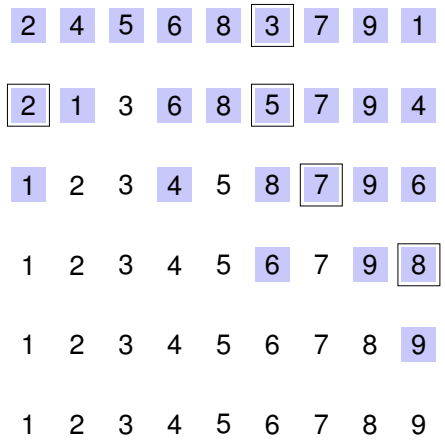

Wahrscheinlichkeit für guten Pivot nach einem Versuch: $\frac{1}{2} =: \rho$.

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

Erwartete Anzahl Versuche[3]: $1/\rho = 2$

[3] Erwartungswert der geometrischen Verteilung:

## Quicksort (willkürlicher Pivot)






## Analyse: Anzahl Vergleiche

*Schlechtester Fall.* Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n-1) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$



## Analyse (Randomisiertes Quicksort)

**Theorem**

*Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log n)$ Vergleiche.*

(ohne Beweis.)



## Praktische Anmerkungen

- Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: Median3$(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.