Freie Universität Berlin

# Objektorientierte Programmierung

## Wiederholung

SoSe 2018

Oliver Wiese

Freie Universität Berlin

# Sortieralgorithmen

- Vergleichs-Algorithmen
  - quadratisch
    - Bubble-Sort $\mathbf{O}(n^2)$
    - Insert-Sort $\mathbf{O}(n^2)$
    - Selection-Sort $\mathbf{O}(n^2)$
  - n (log (n))
    - Quicksort $\mathbf{O}(\ n \cdot \log(n)\ )$ Durchschnittsfall
    - Mergesort $\mathbf{O}(\ n \cdot \log(n)\ )$
    - Heap-Sort $\mathbf{O}(\ n \cdot \log(n)\ )$
- Ohne Vergleiche
  - linear
    - Counting-Sort $\mathbf{O}(n)$
    - Radix-Sort $\mathbf{O}(n)$
    - Bucket-Sort $\mathbf{O}(n)$

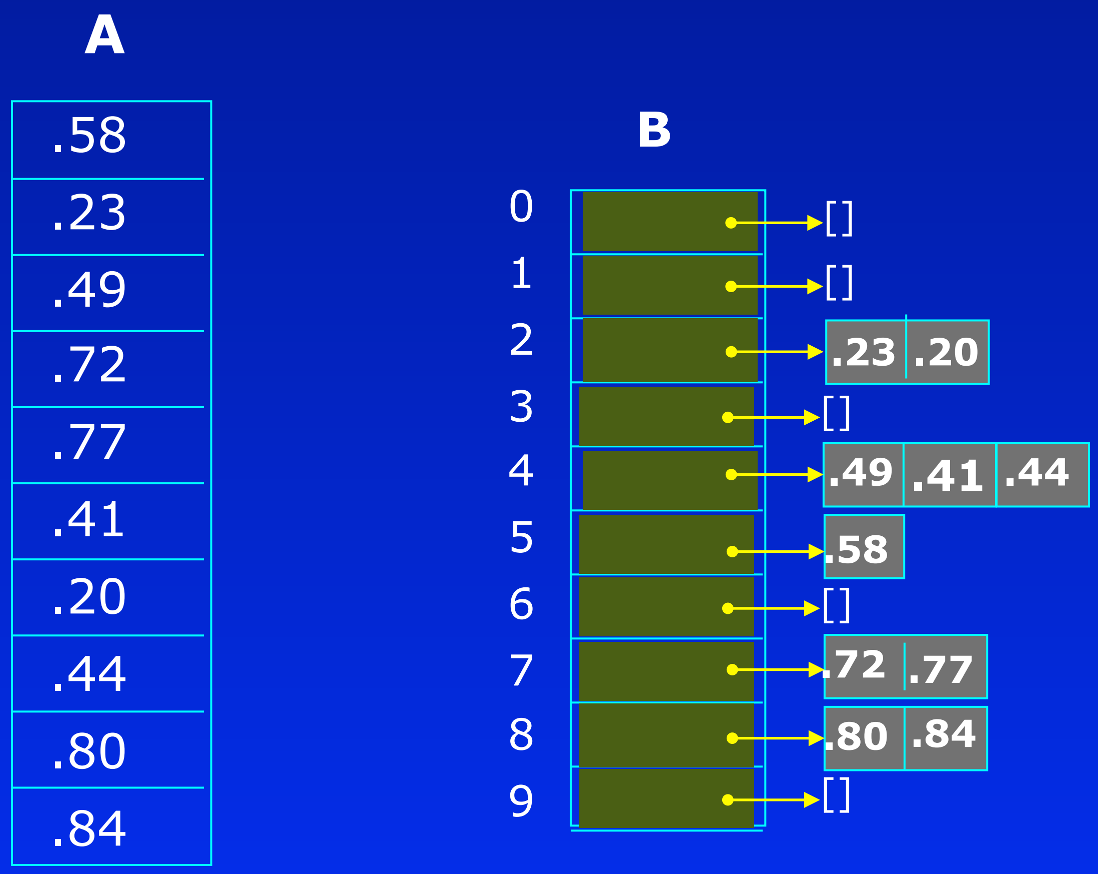
 

# Bucket-Sort

- Die zu sortierenden Daten müssen gleich verteilt über den Wertebereich [0,1) sein.
- **Not-In-Place**

  zusätzlicher Speicherplatz (**O(n)**) wird benötigt
- linearer Aufwand **O(n)**
- Grundidee ist den Wertebereich [0,1) in **m** kleinere Wertebereiche zu teilen und *Buckets* dafür zu definieren.
- Die Zahlen werden in den dazugehörigen *Buckets* verteilt und innerhalb diesen sortiert.
- Zum Schluss werden die Zahlen der Reihe nach aus den *Buckets* ausgegeben.
- Eine Hilfsarray von verketteten Listen wird für die *Buckets* verwendet.

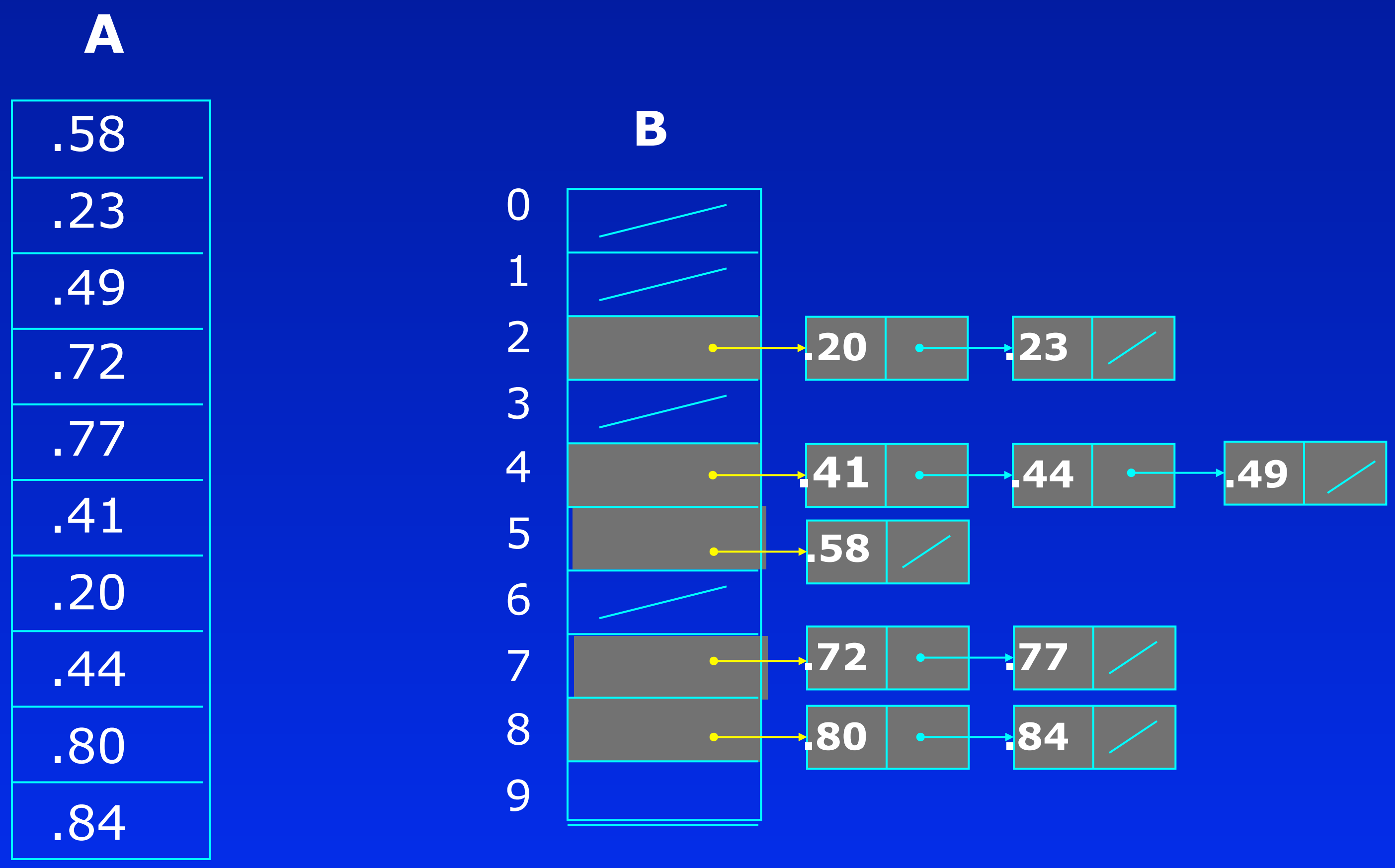
# Bucket-Sort


A
.58
.23
.49
.72
.77
.41
.20
.44
.80
.84
B
0 []
1 []
2 .23 .20
3 []
4 .49 .41 .44
5 .58
6 []
7 .72 .77
8 .80 .84
9 []

# Bucket-Sort

```
def bucketsort ( A ):
    n = len(A)
    B = [ [] for i in range(n) ]
    for i in range(n):
        B[math.floor((A[i])*10)].append(A[i])
    for i in range(0,n):
        insertsort(B[i])
    R = []
    for i in range(n):
        R = R+B[i]
    return R
```

# Bucket-Sort


A
.58
.23
.49
.72
.77
.41
.20
.44
.80
.84
B
0
1
2
3
4
5
6
7
8
9
.20
.23
.41
.44
.49
.58
.72
.77
.80
.84

Freie Universität Berlin

# Sortieralgorithmen

Vergleichs-Algorithmen

- quadratisch
  - Bubble-Sort $\mathbf{O}(n^2)$
  - Insert-Sort $\mathbf{O}(n^2)$
  - Selection-Sort $\mathbf{O}(n^2)$
- n (log (n))
  - Quicksort $\mathbf{O}(\, n \cdot \log(n)\, )$ Durchschnittsfall
  - Mergesort $\mathbf{O}(\, n \cdot \log(n)\, )$
  - Heap-Sort $\mathbf{O}(\, n \cdot \log(n)\, )$

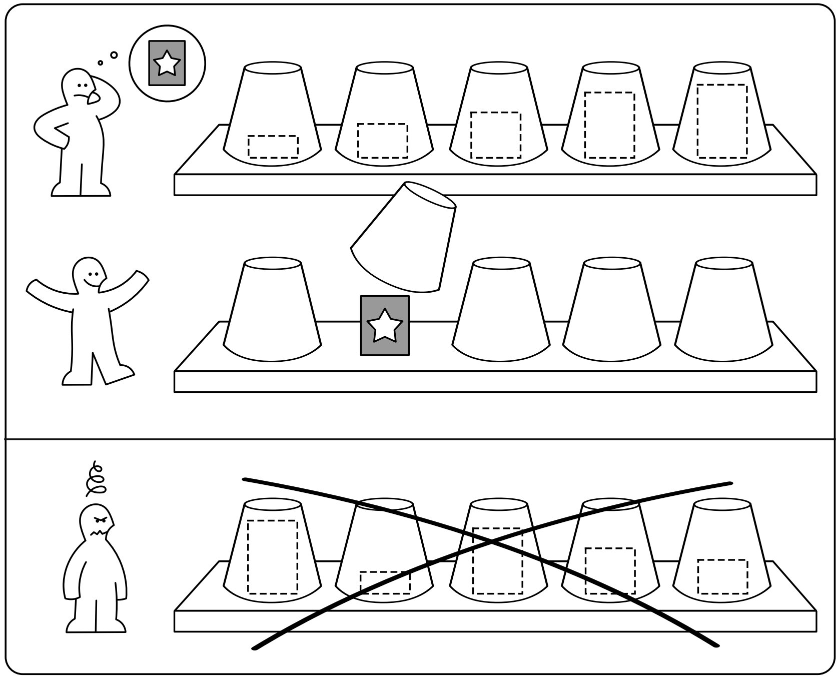
Ohne Vergleiche

- linear
  - Counting-Sort $\mathbf{O}(n)$
  - Radix-Sort $\mathbf{O}(n)$
  - Bucket-Sort $\mathbf{O}(n)$

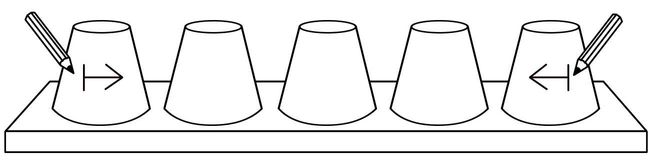
# BINÄRY SEARCH

idea-instructions.com/binary-search/

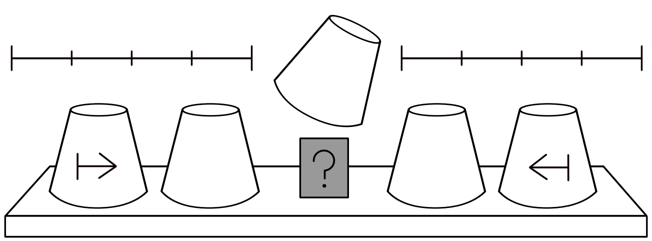

IDEA



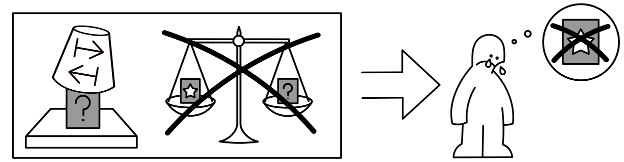
**1**



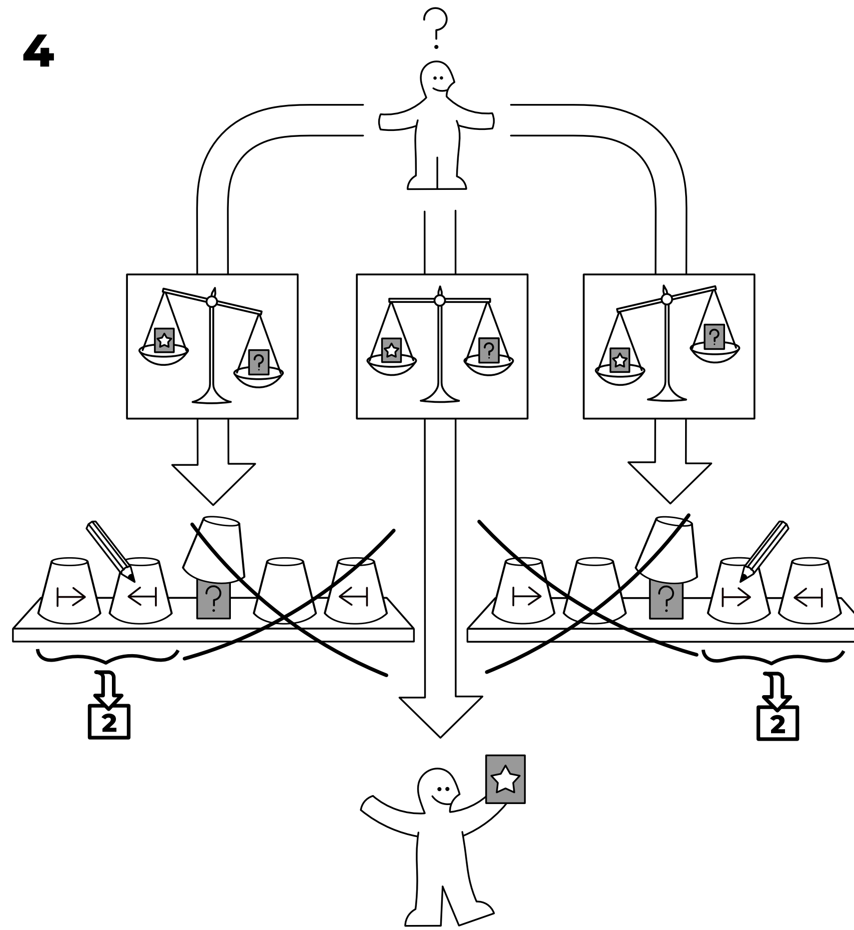
**2**



**3**



**4**



# Binäre Suche

**Rekursiv**

```
def bin_search(key,seq):
    if len(seq)>1:
        m = len(seq)//2
        if seq[m]==key:
            return True
        elif key<seq[m]:
            return bin_search(key, seq[0:m])
        else:
            return bin_search(key, seq[(m+1):])
    elif len(seq)==1:
        return seq[0]==key
    else:
        return False
```

Freie Universität Berlin

# Maximale Schrittanzahl mit Arrays

| | |
|---|---|
| $n = 128$ | $= 2^7$ |
| $64$ | $= 2^6$ |
| $32$ | $= 2^5$ |
| $16$ | $= 2^4$ |
| $8$ | $= 2^3$ |
| $4$ | $= 2^2$ |
| $2$ | $= 2^1$ |
| $1$ | $= 2^0$ |

$7 = \log_2 ( 128 )$

Im schlimmsten Fall

$\approx$ **$\log_2(n)$**

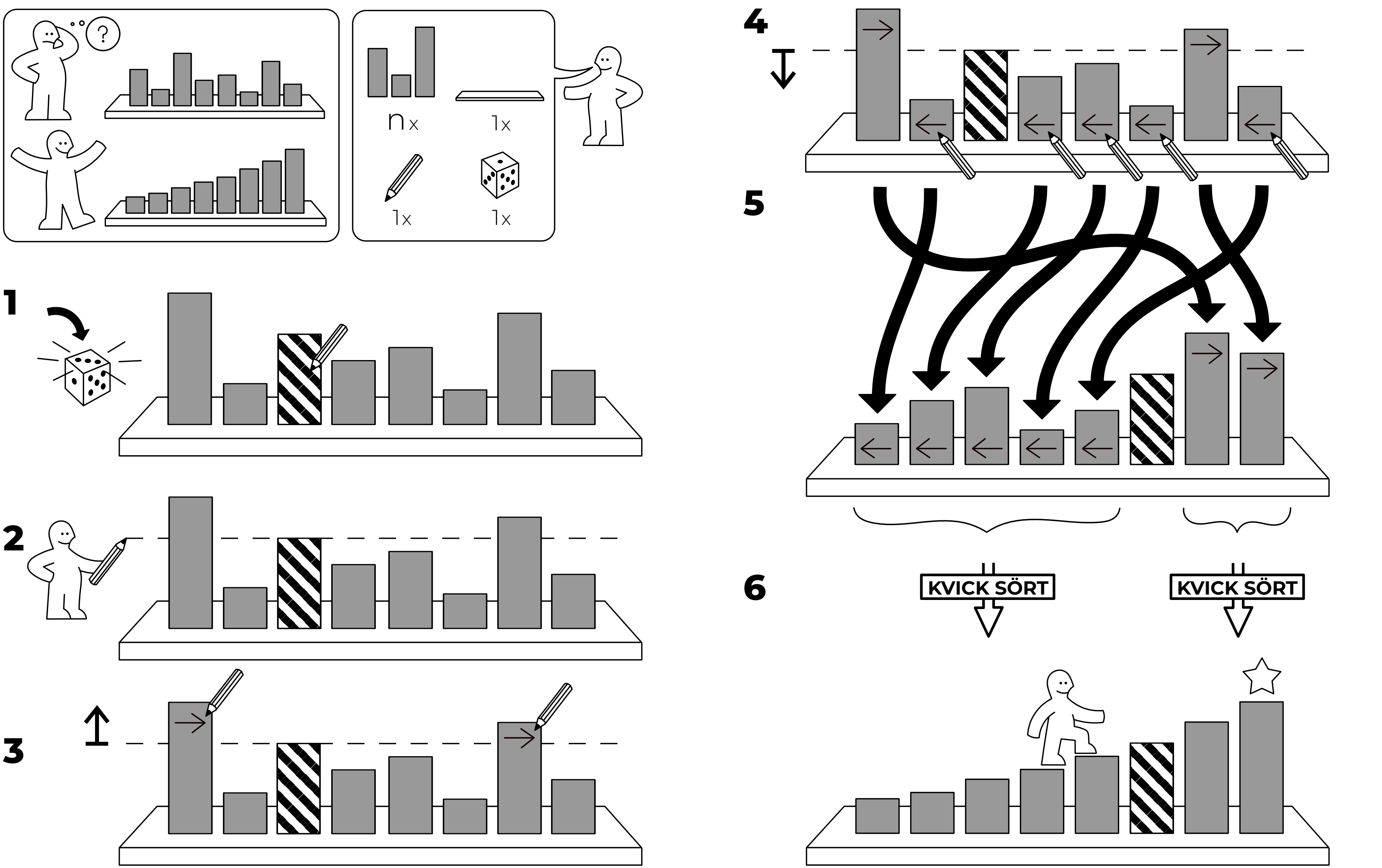
| Anzahl der Elemente | Lineare Suche | Binäre Suche |
|---|---|---|
| 3 | 3 | 2 |
| ... | ... | ... |
| 1,023 | 1,023 | 10 |
| ... | ... | ... |
| 1,073,741,824 | 1,073,741,824 | 30 |

# KVICK SÖRT

idea-instructions.com/quick-sort/


IDEA



# Quick-Sort-Algorithmus á la Haskell

```
def quick_sort(seq):
    if len(seq)>1:
        q1 = [s for s in seq[1:] if s<seq[0]]
        q2 = [s for s in seq[1:] if s>=seq[0]]
        return quick_sort(q1) + [seq[0]] + quick_sort(q2)
    else:
        return seq
```

Speicherverbrauch?   Komplexität der Verkettungsfunktion (+)?

Wenn die Zahlen sortiert sind, stürzt das Programm bereits bei kleinen Arrays ab.

Freie Universität Berlin

# Quicksort-Algorithmus

## imperativ!

Rekursive Implementierung

```
def quicksort (A, low, high ):
    if  low<high:
        m = partition(A, low, high )
        quicksort ( A,  low,  m-1 )
        quicksort ( A,  m+1,  high )
```

low high

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 5 | 7 | 3 | 4 | 9 | 1 | 6 | 5 | 9 | 2 | 0 | 4 | 8 | 3 | 6 |

Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

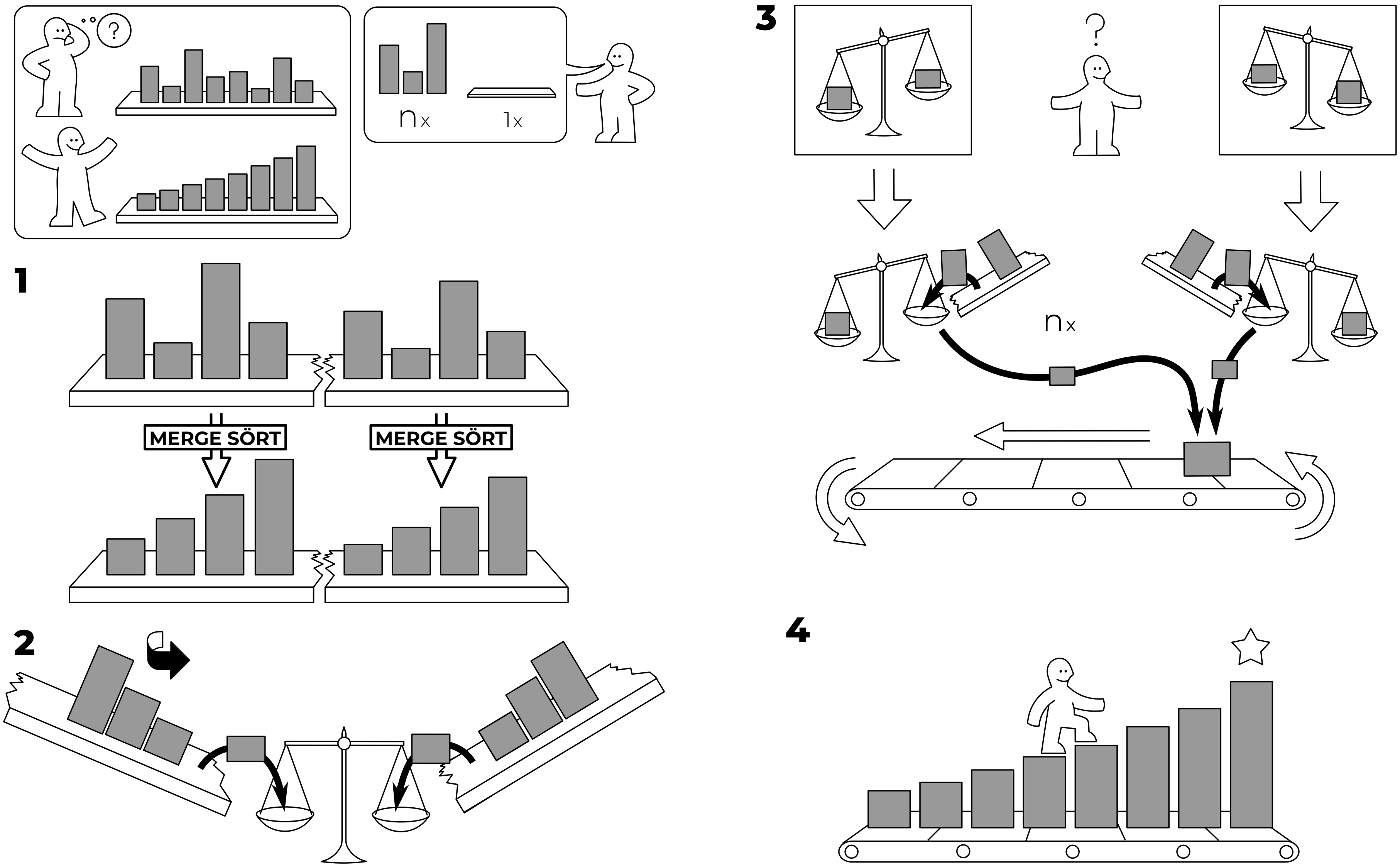
Freie Universität Berlin

# Quicksort-Algorithmus

- **1962** von **Hoare** entwickelt
- ist einer der beliebtesten Sortieralgorithmen
- **einfache** Implementierung
- **in-place**
- Komplexität
    - $\mathbf{O(n \cdot \log(n))}$ im Durchschnitt
    - $\mathbf{O(n^2)}$ im schlimmsten Fall
- **nicht stabil!**

# MERGE SÖRT

idea-instructions.com/merge-sort/

Freie Universität Berlin

# Merge-Sort-Algorithmus

```
def merge(low, high):
        res = []
        i, j = 0, 0
        while i<len(low) and j<len(high):
                if low[i] <= high[j]:
                        res.append(low[i])
                        i = i+1
                else:
                        res.append(high[j])
                        j = j+1
        res = res + low[i:]
        res = res + high[j:]
        return res
```

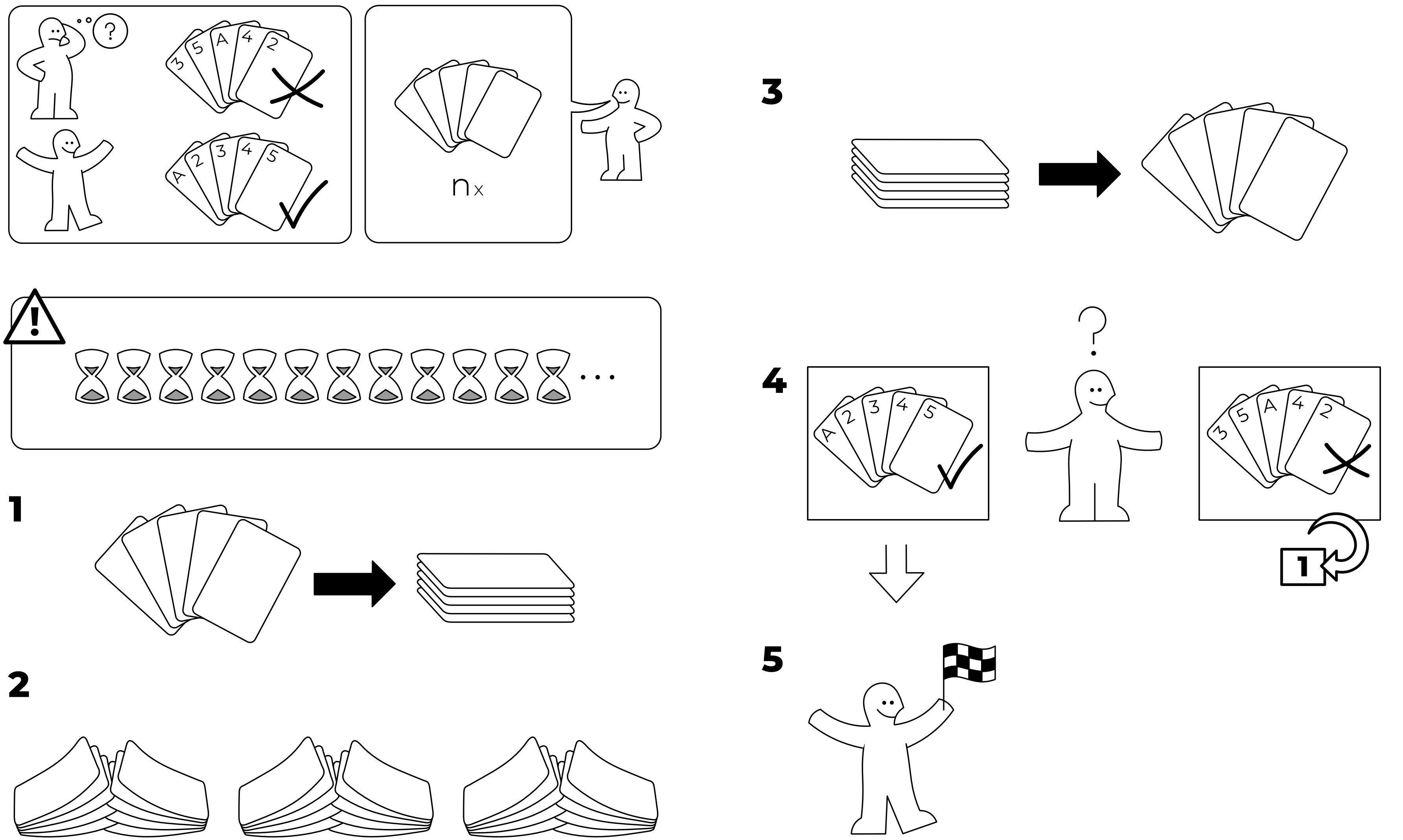
**Speicherverbrauch?**

Rekursiv

```
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A) // 2
        return merge( mergesort(A[:m]), mergesort(A[m:]) )
```

# BOGO SÖRT

idea-instructions.com/bogo-sort/

# Shellsort

Shellsort ist eines der am längsten (1959) bekannten Sortierverfahren.

Der Urheber ist  Donald .L. Shell.

Die Idee des Verfahrens ist es, die Daten als zweidimensionales Feld zu arrangieren und spaltenweise zu sortieren.

Nach dieser Grobsortierung werden die Daten als schmaleres zweidimensionales Feld wieder angeordnet und wiederum spaltenweise sortiert.

Das Ganze wiederholt sich, bis zum Schluss das Feld nur noch aus einer Spalte besteht.

Die Spalten werden alle parallel mit Hilfe des Insertsort-Algorithmus sortiert.

Sei

# Shellsort

9 0 2 2 6 3 7 | 1 9 0 2 6 3 7 | 4 8 5 6 3 7

die zu sortierende Datenfolge

# Shellsort

9 0 2 2 6 3 7

1 9 0 2 6 3 7

4 8 5 6 3 7

Freie Universität Berlin

# Shellsort

| | | | | | | |
|---|---|---|---|---|---|---|
| 9 | 0 | 2 | 2 | 6 | 3 | 7 |
| 1 | 9 | 0 | 2 | 6 | 3 | 7 |
| 4 | 8 | 5 | 6 | 3 | 7 | |

Die Spalten werden sortiert

# Shellsort

| | | | | | | |
|---|---|---|---|---|---|---|
| 9 | 0 | 2 | 2 | 6 | 3 | 7 |
| 1 | 9 | 0 | 2 | 6 | 3 | 7 |
| 4 | 8 | 5 | 6 | 3 | 7 | |

$\longrightarrow$

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 2 | 3 | 3 | 7 |
| 4 | 8 | 2 | 2 | 6 | 3 | 7 |
| 9 | 9 | 5 | 6 | 6 | 7 | |

Sortiert

1 0 0 | 2 3 3 | 7 4 8 | 2 2 6 | 3 7 9 | 9 5 6 | 6 7

# Shellsort

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 3 | 3 |
| 7 | 4 | 8 |
| 2 | 2 | 6 |
| 3 | 7 | 9 |
| 9 | 5 | 6 |
| 6 | 7 | |

# Shellsort

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 3 | 3 |
| 7 | 4 | 8 |
| 2 | 2 | 6 |
| 3 | 7 | 9 |
| 9 | 5 | 6 |
| 6 | 7 | |

Die Spalten werden sortiert

# Shellsort

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 3 | 3 |
| 7 | 4 | 8 |
| 2 | 2 | 6 |
| 3 | 7 | 9 |
| 9 | 5 | 6 |
| 6 | 7 | |

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 2 | 3 |
| 2 | 3 | 6 |
| 3 | 4 | 6 |
| 6 | 5 | 8 |
| 7 | 7 | 9 |
| 9 | 7 | |

## Shellsort

| | |
|---|---|
| 1 | 0 |
| 0 | 0 |
| 0 | 1 |
| 2 | 2 |
| 2 | 2 |
| 3 | 2 |
| 2 | 3 |
| 3 | 3 |
| 6 | 3 |
| 3 | 4 |
| 4 | 5 |
| 6 | 6 |
| 6 | 6 |
| 5 | 6 |
| 8 | 7 |
| 7 | 7 |
| 7 | 7 |
| 9 | 8 |
| 6 | 9 |
| 7 | 9 |

# Shellsort

```
magic = [1391376, 463792, 198768, 86961, 33936, 13776,
         4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1]

def shellsort (A):
        SIZE = len(A)
        for k in range(len(magic)):
                h = magic[k]
                for i in range(h, SIZE):
                        j = i
                        temp = A[j]
                        while j>=h and A[j-h]>temp:
                                A[j] = A[j-h]
                                j = j-h
                        A[j] = temp
```

Aus Erfahrung entwickelte Folge für die Pseudo-Segmentierung

Hier wird das Prinzip des Insertionsort-Algorithmus verwendet

# Shellsort

Wenn die Feldbreiten geschickt gewählt werden, reichen jedes mal wenige Sortierschritte aus, um die Daten spaltenweise zu sortieren.

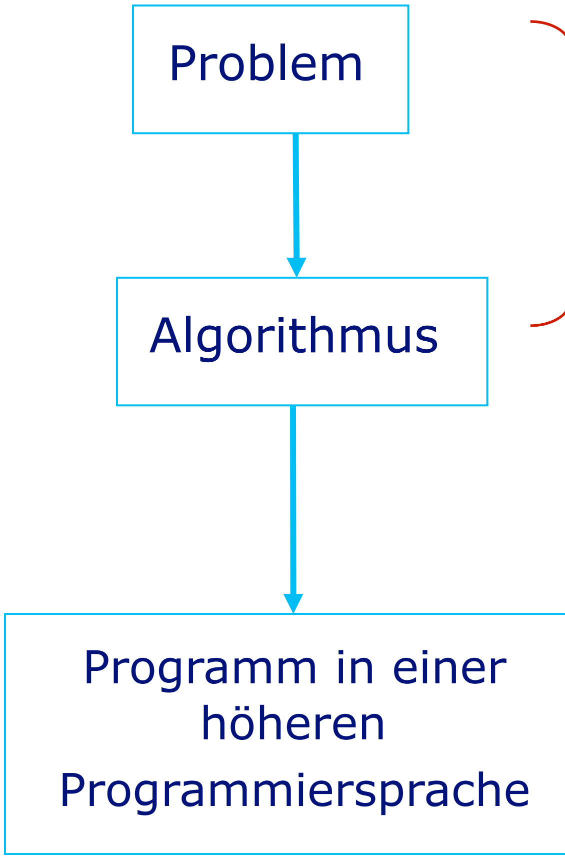
Es gibt noch kein mathematisches Modell, um für beliebige Datenmengen zu entscheiden, welche die optimale Segmentierungssequenz ist.

Eigenschaften:

- **nicht stabil**
- **die Komplexität hängt von der Segmentierung ab**

Mersenne-Zahlen 1, 3, 15,…,$2^k$-1 $O(n^{1,5})$

# Korrekte und effiziente Lösung von Problemen




Wesentlicher Teil der Lösung eines Problems.

Sehr **kreative Phase**

Programmierung oder Codierung des Algorithmus in einer bestimmten Programmiersprache.

**Einfache Phase**

Freie Universität Berlin

# Analyse von Algorithmen

| | |
|---|---|
| **Rechenzeit** | Anzahl der durchgeführten Elementaroperationen in Abhängigkeit von der Eingabegröße. |
| **Speicherplatz** | Maximaler Speicherverbrauch während der Ausführung des Algorithmus in Abhängigkeit von der Komplexität der Eingabe. |
| **Bandbreite** | Wie groß ist die erforderliche Datenübertragung. |

Freie Universität Berlin

# Analyse von Algorithmen

Zeitanalyse

Charakterisierung unserer Daten
(**Eingabegröße**)

Bestimmung der abstrakten Operationen
(**Berechnungsschritte** in unserem Algorithmus)

Eigentliche mathematische Analyse, um eine Funktion in Abhängigkeit der Eingabegröße zu finden.
**Komplexitätsanalyse**

# Eingabedaten

- Eingabedaten und Eingabegröße charakterisieren
- Betrachtung vom **schlimmsten Fall**

**Beispiel:** Die Anzahl der Objekte, die wir sortieren wollen

Die Anzahl der Listen, die wir verarbeiten wollen

u.s.w.

# Die zu messenden Operationen

Bestimmung der abstrakten Operationen:

- mehrere kleinere Operationen zusammenfassen
- einzeln in konstanter Zeit

Bestimmen wesentlich den gesamten Zeitaufwand des Algorithmus (durch ihr häufiges Vorkommen)

# Die zu messenden Operationen

**Beispiel:** Bei Sortieralgorithmen messen wir Vergleiche

Bei anderen Algorithmen in imperativen Sprachen:

- Speicherzugriffe
- Anzahl der Multiplikationen
- Anzahl der Bitoperationen
- Anzahl der Schleifen-Durchgänge
- Anzahl der Funktionsaufrufe
- u.s.w.

In Funktionalen Programmiersprachen

- Anzahl der Reduktionen

# O-Notation

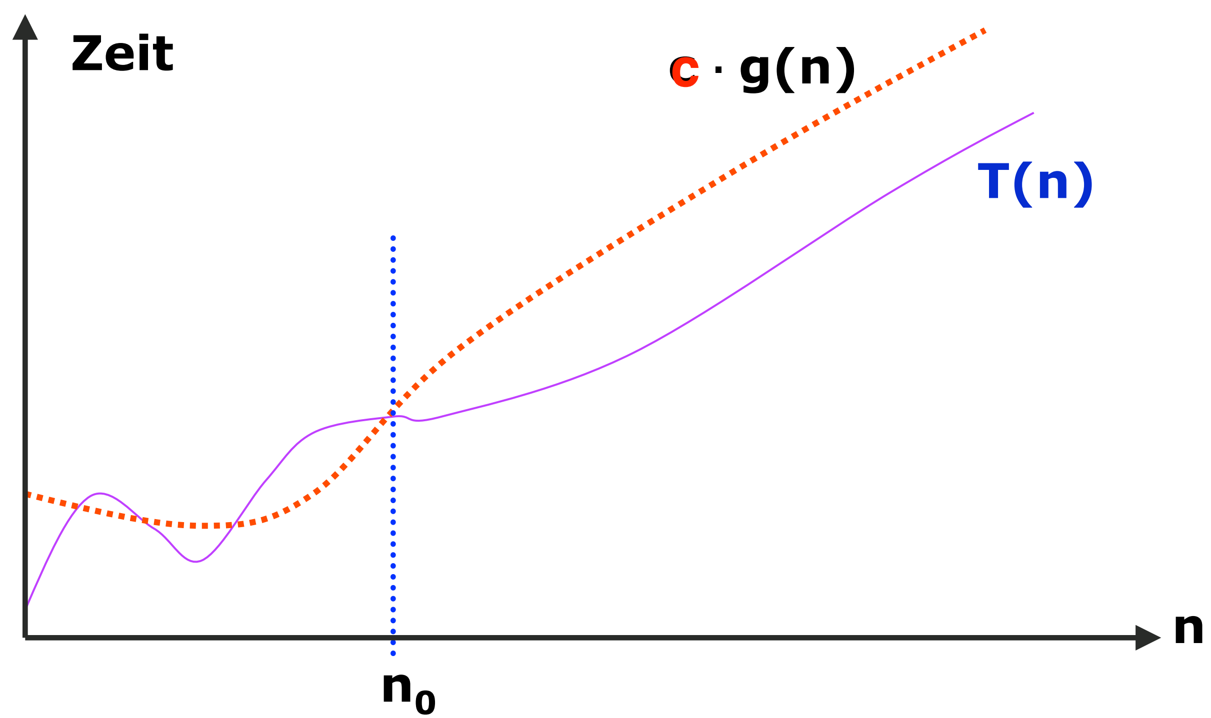
Für die Effizienzanalyse von Algorithmen wird eine spezielle mathematische Notation verwendet, die als **O-Notation** bezeichnet wird.

Die **O-Notation** erlaubt es, Algorithmen auf einer höheren Abstraktionsebene miteinander zu vergleichen.

Algorithmen können mit Hilfe der **O-Notation** unabhängig von Implementierungsdetails, wie Programmiersprache, Compiler und Hardware-Eigenschaften, verglichen werden.

## Definition:

Die Funktion **$T(n) = O(g(n))$**, wenn es positive Konstanten **$c$** und **$n_0$** gibt, so dass **$T(n) \leq c \cdot g(n)$** für alle **$n \geq n_0$**



# Entwurfsprinzipien

- Divide & Conquer
- Dynamisches Programmieren
- Gierige Algorithmen (greedy)

# Teile und Herrsche

**"Divide und Conquer"**

Viele Probleme lassen sich nicht mit trivialen Schleifen lösen und haben gleichzeitig den Vorteil, dass eine rekursive Lösung keine überflüssigen Berechnungen verursacht.

Solche Probleme lassen sich in Teilprobleme zerlegen, deren Lösung keine überlappenden Berechnungen beinhalten.

**Lösungsschema:**

**Divide:**

Teile ein Problem in zwei oder mehrere kleinere ähnliche Teilprobleme, die (rekursiv) isoliert behandelt werden können.

**Conquer:**

Löse die Teilprobleme auf dieselbe Art (rekursiv).

**Merge:**

Füge die Teillösung zur Gesamtlösung zusammen.

# Dynamisches Programmieren

Rekursive Lösungen von einigen Problemen haben exponentielle Laufzeit, weil Teilprobleme wiederholt benötigt und berechnet werden.

Solche Probleme lassen sich in Teilprobleme zerlegen, deren Lösung überlappenden Berechnungen beinhalten und überlappende Berechnungen werden zwischengespeichert.

**Lösungsschema:**

**Aufteilung in Unterprobleme:**

Teile ein Problem in zwei oder mehrere kleinere Unterprobleme

Löse Unterprobleme

**Wiederverwendung von Zwischenergebnisse:**

Merke  Zwischenergebnisse oder Lösungen von Unterproblemen

# Gierige Algorithmen

**"Greedy algorithms"**

Bei manchen Problemen wird nach einer optimalen/besten/größten Ergebnis gesucht und die Berechnung von lokalen optimalen Lösungen ist einfach.

Beispiele sind Münzwechselproblem, Reiseplanungsproblem.

Vorsicht: Nicht immer liefert eine gierige Lösung die optimale Lösung!

**Lösungsschema:**

**Initiallösung**

Beginnen mit einer einfachen Lösung und berechne die Kosten.

**Iterative Verbesserung**

Betrachte alle möglichen nächsten Schritte zur Verbesserung und wähle die gerade (lokal) am größte Verbesserung aus. In jedem Schritt verkleinere das Problem.

# Problem: Quickselect

In einer unsortierten Liste soll das k-kleinste Element gesucht werden. Implementieren Sie folgenden Algorithmus:

1. Wählen Sie (analog zu Quicksort) ein Pivotelement aus und partitionieren Sie die Liste.
2. Überlegen Sie sich, ob Sie das zu suchende Element schon gefunden haben oder in welcher der Teillisten es sich befindet.
3. Suchen Sie dann gegebenenfalls das k-kleinste Element in der Teilliste rekursiv.

Klausuraufgabe von ProInformatik 2017