# Such + Sortieralgorithmen

# Hooks/Magic Methods

Student

- `__init__`
- `__lt__`
- `__eq__`
- `__str__`

Vector

- `__add__`
- `__sub__`
- `__neg__`
- `__abs__`

| Common Syntax | Special Method Form | |
|---|---|---|
| a + b | a.__add__(b); | alternatively b.__radd__(a) |
| a − b | a.__sub__(b); | alternatively b.__rsub__(a) |
| a ∗ b | a.__mul__(b); | alternatively b.__rmul__(a) |
| a / b | a.__truediv__(b); | alternatively b.__rtruediv__(a) |
| a // b | a.__floordiv__(b); | alternatively b.__rfloordiv__(a) |
| a % b | a.__mod__(b); | alternatively b.__rmod__(a) |
| a ∗∗ b | a.__pow__(b); | alternatively b.__rpow__(a) |
| a << b | a.__lshift__(b); | alternatively b.__rlshift__(a) |
| a >> b | a.__rshift__(b); | alternatively b.__rrshift__(a) |
| a & b | a.__and__(b); | alternatively b.__rand__(a) |
| a ^ b | a.__xor__(b); | alternatively b.__rxor__(a) |
| a \| b | a.__or__(b); | alternatively b.__ror__(a) |
| a += b | a.__iadd__(b) | |
| a −= b | a.__isub__(b) | |
| a ∗= b | a.__imul__(b) | |
| ... | ... | |
| +a | a.__pos__() | |
| −a | a.__neg__() | |
| ~a | a.__invert__() | |
| abs(a) | a.__abs__() | |
| a < b | a.__lt__(b) | |
| a <= b | a.__le__(b) | |
| a > b | a.__gt__(b) | |
| a >= b | a.__ge__(b) | |
| a == b | a.__eq__(b) | |
| a != b | a.__ne__(b) | |

# Dekoratoren

Implementiere ein Dekorator, der für eine Methode einer Klasse die Laufzeit messt.

# Dekoratoren

Implementiere für ein Repository ein Dekorator, der die Daten auch in einer Datei speichert.

# Inhalt

- Testing
- Komplexität
- Search
- Sort


BUG

# Was ist ein `Fehler`?

1. der Programmierer macht einen Fehler
2. und hinterlässt den Fehler im Programmcode
3. wird dieser Code ausgeführt, haben wir eine Abnormalität im Programmzustand,
4. die sich als ein Fehler nach außen manifestiert

BUGS BUGS BUGS 


I DON'T ALWAYS TEST
VIA 9GAG.COM
BUT WHEN I DO, I TEST DURING PRODUCTION

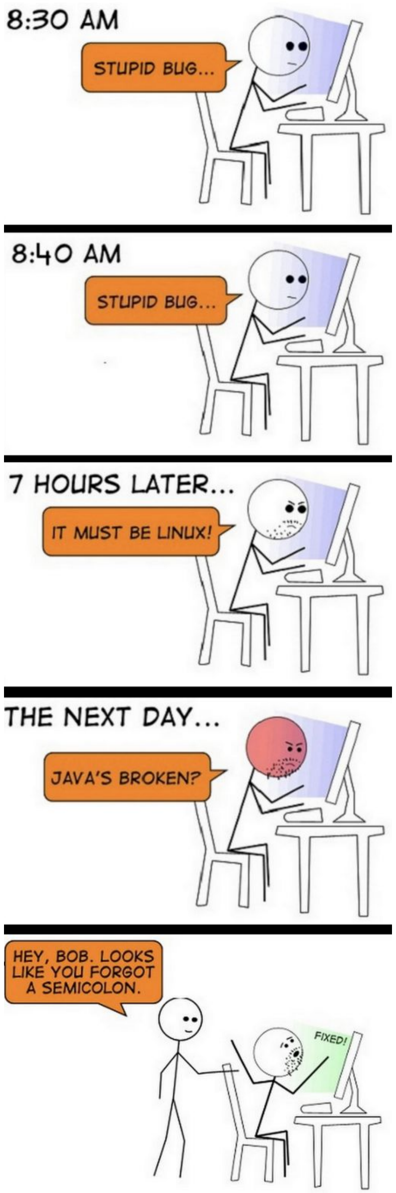
# Softwaretest

- Ziel des Testens ist, durch gezielte Programmausführung Fehler zu erkennen
  - Test Cases (input + output + assert)

- Auswirkung: Testing soll Vertrauen in die Qualität der Software schaffen

- Die Korrektheit eines Programms kann durch Testen (außer in trivialen Fällen) nicht bewiesen werden

*Program testing can be used to show the presence of bugs, but never to show their absence.* (Dijkstra)



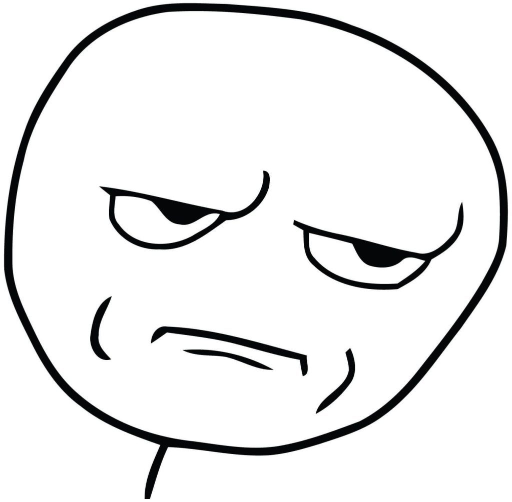
# Methoden beim Testen

**Black-Box-Test**

die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden

**White-Box-Test**

die Tests mit Kenntnissen über die innere Funktionsweise des zu testenden Systems entwickelt werden



# Methoden beim Testen

| Black-Box-Test | White-Box-Test |
| --- | --- |
| • Testfälle gehen von der Spezifikation aus | • Testfälle ausgehend von der Struktur des Testobjekt |
| • Interna des Testobjekts sind bei der Ermittlung der Testfälle unbekannt | • Testfälle werden vom Entwickler beschrieben |
| • Testüberdeckung wird an Hand des spezifizierten Ein/Ausgabeverhaltens gemessen | • Testüberdeckung wird an Hand des Codes gemessen |

```python
def isPrime(nr):
    """
      Verify if a number is prime
      return True if nr is prime False if not
      raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

| Black Box | White Box (cover all the paths) |
| --- | --- |
| • test case for a prime/not prime<br>• test case for 0<br>• test case for negative number | • test case for 0<br>• test case fot negative<br>• test case for 1<br>• test case 3<br>• test case for prime (no divider)<br>• test case for not prime |

```python
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

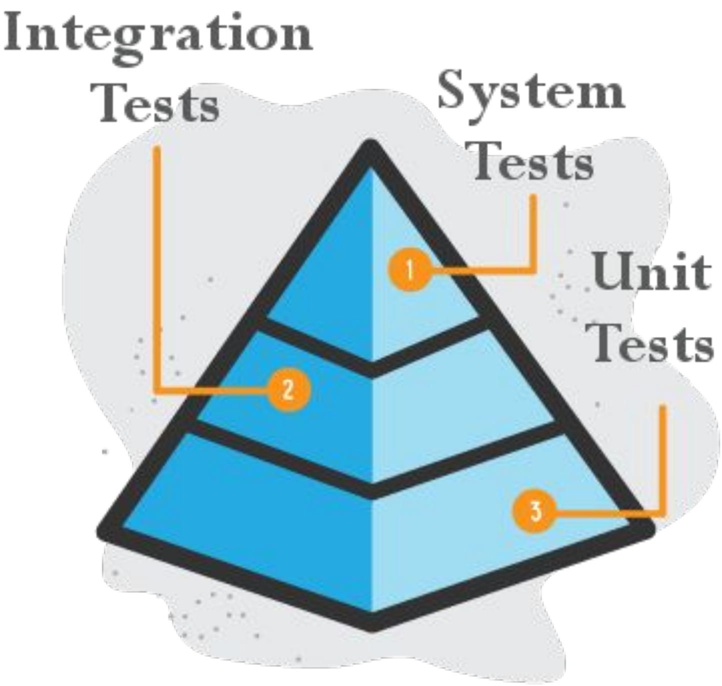
```python
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

# Testen

- Komponententest, Modultest (Unit Test)
  - der erste Schritt im Testing
  - die Tests die wir schon geschrieben haben
  - was ich umgesetzt habe

- Integrationstest (Integration Test)
  - was wir als team umgesetzt haben

- Systemtest (System Test)
  - das ganze Ding



# Unit Testing

- Unit Test: **Automatischer White-Box Test** welcher eine Einheit (z.B. Modul, Klasse, Komponente etc.) testet

- Unit Testing: Erstellen, Verwalten und Ausführen aller Unit Tests

- Unit Testing ist das Fundament aller agilen Softwareentwicklung Methodologien.

# Beurteilung von Algorithmen

- viele Algorithmen, um dieselbe Funktion zu realisieren
  - Welche Algorithmen sind besser?

- um Algorithmen vergleichen zu können muss man andere Faktoren analysieren

- nicht-funktionaler Eigenschaften:
  - Zeiteffizienz
    - Wie lange dauert die Ausführung?
  - Speichereffizienz
    - Wie viel Speicher wird zur Ausführung benötigt?
  - Benötigte Netzwerkbandbreite
  - Einfachheit des Algorithmus
  - Aufwand für die Programmierung

# Beurteilung von Algorithmen

- viele Algorithmen, um dieselbe Funktion zu realisieren
  - Welche Algorithmen sind besser?

- um Algorithmen vergleichen zu können muss man andere Faktoren analysieren

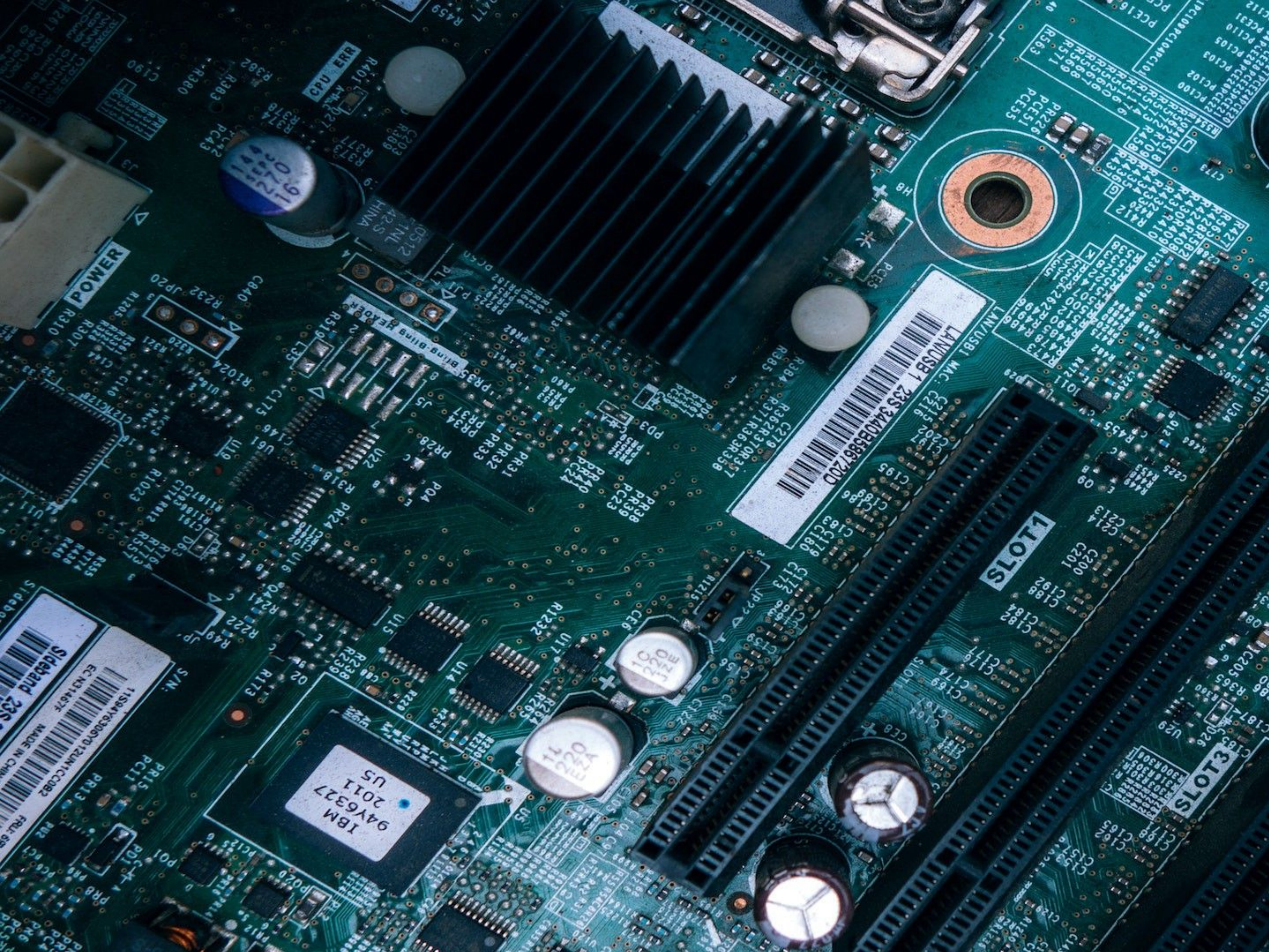
- nicht-funktionaler Eigenschaften:
  - Zeiteffizienz
    - Wie lange dauert die Ausführung? ← unser Fokus
  - Speichereffizienz
    - Wie viel Speicher wird zur Ausführung benötigt?
  - Benötigte Netzwerkbandbreite
  - Einfachheit des Algorithmus
  - Aufwand für die Programmierung

# Ressourcenbedarf

- Prozesse verbrauchen:
  - Rechenzeit
  - Speicherplatz

- das Problem: viele Faktoren beeinflussen die Laufzeit

- typische Beispiele:
  - der konkreten Programmierung
  - Prozessorgeschwindigkeit
  - Programmiersprache
  - Qualität des Compilers


POWER
CPU ERR
SLOT1
SLOT3
IBM
94Y6327
2011
US

# Beispiel

```
def fibonacci(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = StopWatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"


measureFibo(32)
```

```
fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

[ ] ja [ ] nein [X] jein

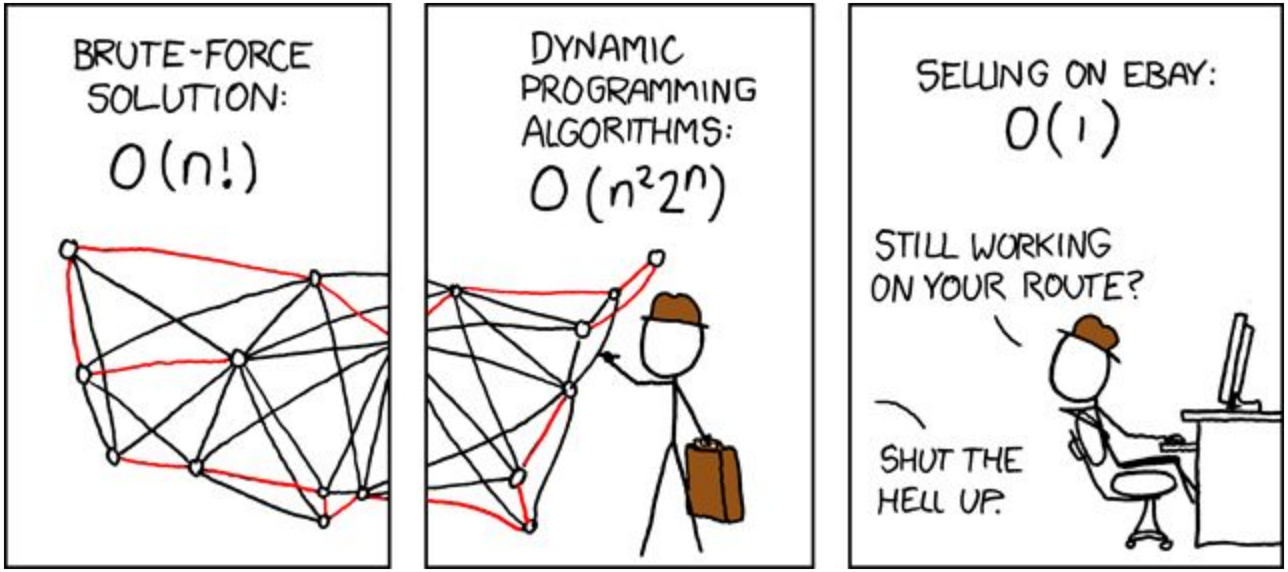
# Leistungsverhalten

- **Laufzeitkomplexität:** Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- deswegen werden Laufzeiten nicht direkt vergleichen
- wir werden aber ein theoretisches Modell für Laufzeit verwenden
  - und mit diesem Modell Funktionen vergleichen
- **Theorie:** liefert untere Schranke, die für jeden Algorithmus gilt, der das Problem löst
- Spezieller Algorithmus liefert obere Schranke für die Lösung des Problems

# Laufzeit

Die Laufzeit **T(x)** eines Algorithmus **A** bei Eingabe **x** ist definiert als die Anzahl von Basisoperationen, die Algorithmus **A** zur Berechnung der Lösung bei Eingabe **x** benötigt

**Ziel:** Laufzeit = Funktion der Größe der Eingabe



# Laufzeit

- Sei **A** ein gegebener Algorithmus und **x** Eingabe für **A**, **|x|** Länge von **x**, und **T(x)** die Laufzeit von **A** auf **x**

- **Ziel:** beschreibe den Aufwand eines Algorithmus als Funktion *der Größe des Inputs*

- **Der beste Fall:**

  `T(n) = inf {T(x) | |x| = n, x Eingabe für A}`

- **Der schlechteste Fall:**

  `T(n) = sup {T(x) | |x| = n, x Eingabe für A}`

# Basisoperationen und deren Kosten

Für eine präzise mathematische Laufzeitanalyse benötigen wir **ein Rechenmodell**, das Basisoperationen und deren Kosten definiert.

Als Basisoperationen definieren wir:

- Arithmetische Operationen
- Datenverwaltung
- Kontrolloperationen

**Kosten:** Zur Vereinfachung nehmen wir an, dass jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt

# Minimum-Suche

**Eingabe:** Array von `n` Zahlen

**Ausgabe:** index `i`, so dass `a[i]<a[j]`, für alle `j`

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
    return min
```

# Minimum-Suche

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
    return min
```

| **Kosten:** | **Max Anzahl:** |
|---|---|
| c1 | 1 |
| c2 | n-1 |
| c3 | n-1 |
| c4 | n-1 |

**Zeit:**

$T(n) = c1 + (n-1)\ (c2+c3+c4) < c5n + c1$

$n$ = Größe des Arrays

# Fakultät

$f(n) = 1! \cdot 2! \cdots (n-2)! \cdot (n-1)!$

```
def f (n):
   fn = 1
   while n:
      for i in range(1, n):
         fn *= i
      n -= 1
   return fn
```

# Asymptotische Komplexität

- drei Operationen: Multiplikation, Inkrementierung, Vergleichen
- Anzahl von Multiplikationen

```
M(n) = (n - 1) + M(n - 1) =  (n - 1) + (n - 2) + M(n - 2)
=  (n-1) * n / 2
```

- Anzahl von Inkrementierungen

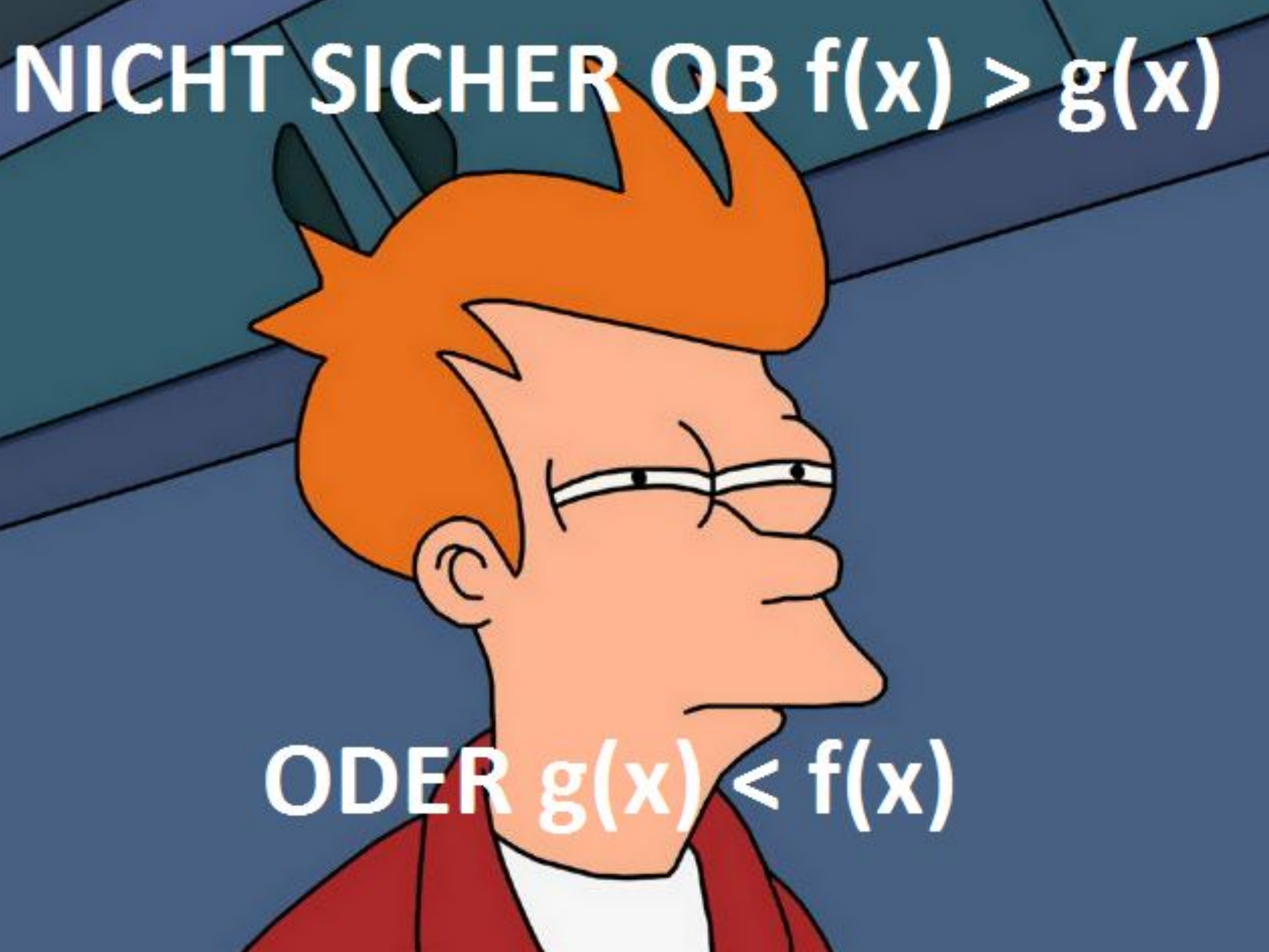
```
I(n) = n + M (n + 1) = n + (n - 1) + M (n - 1)
= n * (n + 1) / 2
```

- Anzahl von Vergleichen

```
V(n) = (n + 1) + M(n + 2) = (n+1) * (n+2) / 2
```

- benötigte Anweisungen

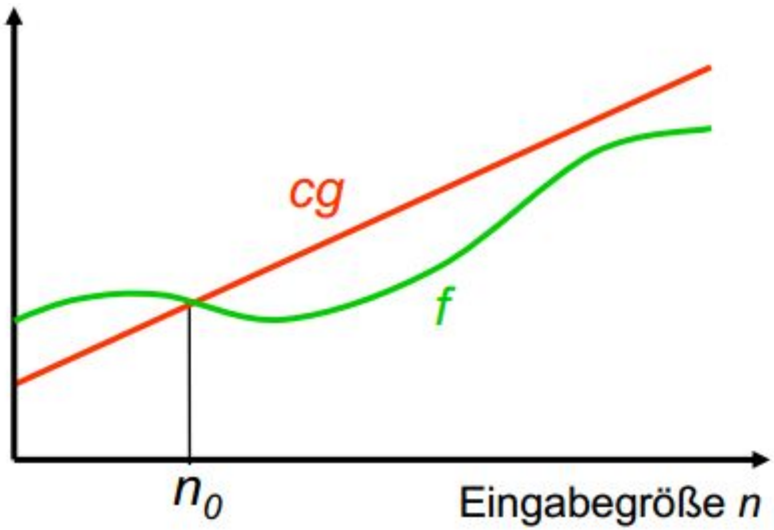
```
Z(n) = 1 + n + n(n+1)/2 =  n(n+3)/2
```


NICHT SICHER OB f(x) > g(x)
ODER g(x) < f(x)

# Asymptotische Komplexität

**O-Notation:** wenn eine Funktion `f(n)` höchstens so schnell wächst wie eine andere Funktion `g(n)`. `g(n)` ist also **die obere Schranke** für `f(n)`.

`f(n)` ist in `O(g(n))`, wenn es ein $c > 0$ und ein $n0 \in N$ gibt, so dass für alle $n \geq n0$ $f(n) \leq c*g(n)$ gilt



# Asymptotische Komplexität

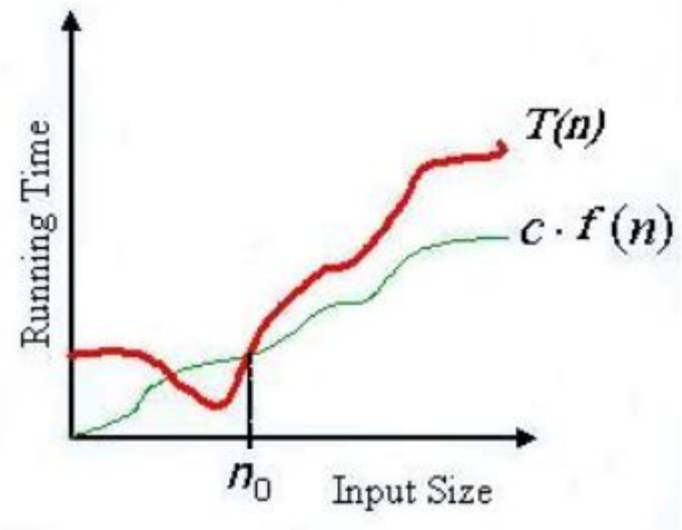
Häufige Größenordnung der Komplexität:

- `O(1)`: In konstanter (von n unabhängiger) Zeit ausführbar
- `O(log n)`: Bei Verdoppelung von `n` läuft das Programm um eine konstante Zeit länger
- `O(n)`: Linear Laufzeit proportional zu `n`
- `O(n*n)`: Quadratische Laufzeit
- `O(n^3)`: Kubische Laufzeit; nur für kleinere `n` geeignet
- `O(2^n)`: Exponentielles Wachstum; solche Programme sind in der Praxis fast immer wertlos

# Asymptotische Komplexität

**Ω-Notation:** wenn eine Funktion `f(n)` mindestens so schnell wächst wie eine andere Funktion `g(n)`. `g(n)` Ist also **die untere Schranke** für `f(n)`.

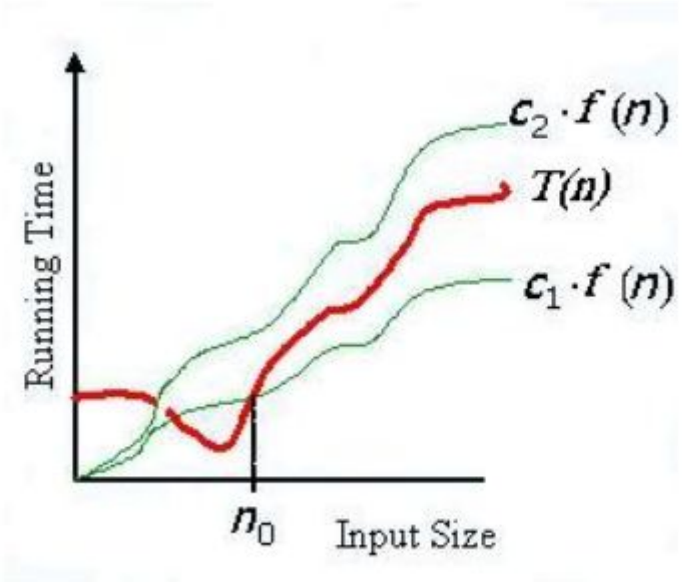
`f(n)` ist in $\Omega$`(g(n))`, wenn es ein $c > 0$ und ein $n0 \in N$ gibt, so dass für alle $n \geq n0$ $f(n) \geq c*g(n)$ gilt



# Asymptotische Komplexität

**Θ-Notation**: wenn eine Funktion `f(n)` sowohl von oben als auch von unten durch `g(n)` beschränkt ist. `g(n)` ist also **die exakte Schranke** für `f(n)`.

θ(g(n)) ist definiert durch θ(g(n)) = O(g(n)) ∩ Ω(g(n)).



# Beispiele

```python
def f1(n):
    s = 0
    for i in range(1,n+1):
        s=s+i
    return s
```

$$T(n)=\sum_{(i=1)}^{n} 1=n \rightarrow T(n)\in\Theta(n)$$

Overall complexity $\Theta(n)$

Best/Average/Worst case is the same

```python
def f2(n):
    i = 0
    while i<=n:
        #atomic operation
        i = i + 1
```

$$T(n)=\sum_{(i=1)}^{n} 1=n \rightarrow T(n)\in\Theta(n)$$

Overall complexity $\Theta(n)$

Best/Average/Worst case is the same

```python
def f3(l):
    """
    l - list of numbers
    return True if the list contains
an even nr
    """
    poz = 0
    while poz<len(l) and l[poz]%2 !=0:
        poz = poz+1
    return poz<len(l)
```

**Best case**:

The first element is an even number: $T(n)=1\in\Theta(1)$

**Worst case**: No even number in the list: $T(n)=n\in\Theta(n)$

**Average Case**:

While can be executed 1,2,..n times (same probability).

Number of steps = the average number of while iterations

$$T(n)=(1+2+\ldots+n)/n=(n+1)/2\rightarrow T(n)\in\Theta(n)$$

Overall complexity $O(n)$

# Suchverfahren - Charakteristiken

- Eingabe: Folge von Zahlen `a, n, <a1,a2,...,an>` und `k`
  - Vorbedingung: `n∈ N, n≥0;`

- Ausgabe: `p`
  - Nachbedingungen : `(0≤p≤n-1 and k = a[p]) or (p=-1)`

In der Praxis:

- gespeichert in Arrays, Linked Lists, …
- Charakterisierung der gesuchten Objekte durch Such-Schlüssel
- Such-Schlüssel können z.B. Attribute der Objekte sein
- Beispiel: **ID von Personen**

# Einfache Suchverfahren

- aufwand für alle Verfahren etwa gleich groß
  - außer Linearem Suchen
- einfachstes Suchverfahren verwenden
  - binäres Suchen
- exponentielles Suchen
  - bei großen Daten

# Lineare Suche

Gegeben sei `A[1 .. n]` und `k` (ein Schlüssel)

Idee der linearen Suche:

- sequentielles Durchlaufen des Feldes `A`
- Vergleich der Schlüssel `A[i]`, `i=1, …, n` mit dem Suchschlüssel `k`

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of the element
         or -1 if the element is not in l
    """
    poz = -1
    for i in range(0,len(l)):
        if el==l[i]:
            poz = i
    return poz
```

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of first occurrence
            or -1 if the element is not in l
    """
    i = 0
    while i<len(l) and el!=l[i]:
        i=i+1
    if i<len(l):
        return i
    return -1
```

# Laufzeitanalyse

- Best-Case: sofortiger Treffer: `T(n) = 1`, also `T(n) = O(1)`
- Worst-Case: alles durchsuchen: `T(n) = n`, also `T(n) = O(n)`
- Average-Case: erfolgreiche Suche unter der Annahme, dass jede Anordnung der Elemente gleich wahrscheinlich ist:

  `T(n) = (1+2+...+n-1)/n also T(n) = O(n)`

# Sortiertes Feld

- wenn das Element nicht vorkommt, man kann den Index zurückgeben, wo das Element potentielles gehört

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    poz = -1
    for i in range(0,len(l)):
        if el<=l[i]:
            poz = i
    if poz==-1:
        return len(l)
    return poz
```

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[i]:
        i=i+1
    return i
```

# Fazit

- sehr einfaches Verfahren
- eignet sich auch für einfach verkettete Listen
- das Verfahren ist auch für unsortierte Felder geeignet
- aber das Verfahren ist nur für kleine Werte von `n` **praktikable**

# Binäre Suche

- Falls in einer Folge häufig gesucht werden muss, so lohnt es sich, die Feldelemente sortiert zu speichern

- **Eingabe**: Sortiertes Feld
  - halbieren des Suchraums in jedem Schritt, indem **der gesuchte Wert mit dem Wert auf der Mittelposition des geordneten Feldes** verglichen wird
  - gesuchter Wert ist kleiner: weiterarbeiten mit linkem Teilfeld
  - gesuchter Wert ist größer: weiterarbeiten mit rechtem Teilfeld

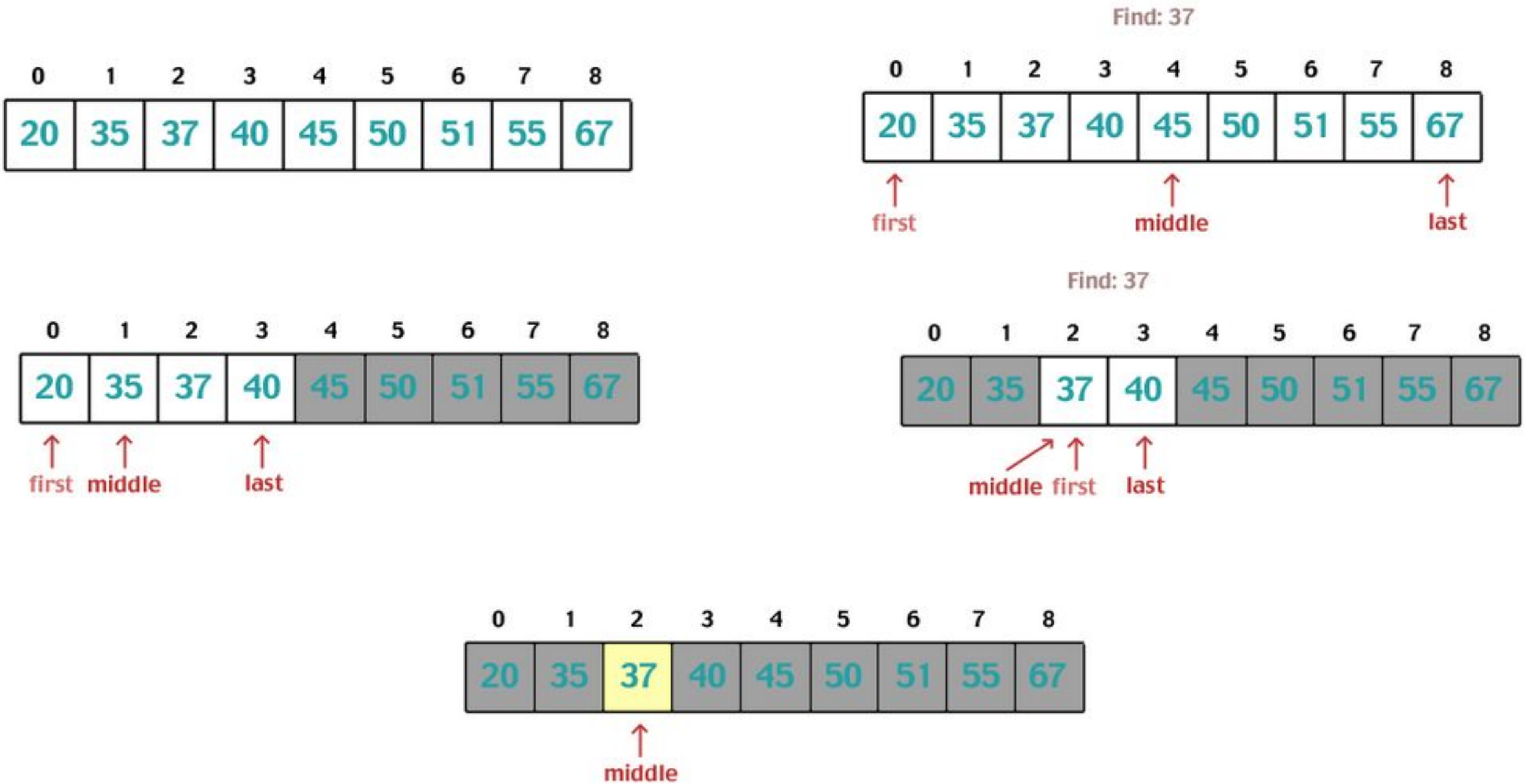
# Laufzeitanalyse

- **Zählen der Anzahl der Vergleiche**

- Best-Case:
  - sofortiger Treffer: `T(n) = 1`, also `T(n) = O(1)`

- Worst-Case:
  - Suchraum muss solange halbiert werden, bis er nur noch 1 Element enthält,
  - oft logarithmisch
  - `T(n) = T(n/2) + 1 = log(n + 1), T(n) = O(logn)`

# Binäre Suche



# Binäre Suche

```
def binary (arr, start, end, el):

        mid = (end + start) // 2

        if el == arr[mid]:
            return mid

        if arr[mid] > el:
            return binary(arr, start, mid - 1, el)
        else:
            return binary(arr, mid + 1, end, el)

    return -1
```

# Binäre Suche

```
def binary (arr, start, end, el):
    if start <= end:
        mid = (end + start) // 2

        if el == arr[mid]:
            return mid

        if arr[mid] > el:
            return binary(arr, start, mid - 1, el)
        else:
            return binary(arr, mid + 1, end, el)

    return -1
```

# Fazit

- gut geeignet für große Werte `n`

- Beispiel:
  - sei `n = 2` Millionen
  - Lineare Suche benötigt im Worst Case 2 Millionen Vergleiche
  - Binäre Suche benötigt: **log (2 * 10^6) ~ 20** Vergleiche

- nicht gut geeignet, wenn sich die Daten häufig ändern

# In Python

## index()

```
l = range(1,10)
try:
    poz = l.index(11)
except ValueError:
    # element is not in the list
```

## __eq__, __gt__, __lt__, …, __cmp__

```
class MyClass:
    def __init__(self,id,name):
        self.id = id
        self.name = name

    def __eq__(self,ot):
        return self.id == ot.id

#    def __cmp__(self,ot):
#        return self.id.__cmp__(ot.id)


def testIndex():
    l = []
    for i in range(0,200):
        ob = MyClass(i,"ad")
        l.append(ob)

    findObj = MyClass(32,"ad")
    print "positions:" +str(l.index(findObj))
```

# In Python

## in

```
l = range(1,10)
found = 4 in l
```

## __iter__,next

```
class MyClass2:
    def __init__(self):
        self.l = []

    def add(self,obj):
        self.l.append(obj)

    def __iter__(self):
        """
          Return an iterator object
        """
        self.iterPoz = 0
        return self
```

```
def next(self):
        """
          Return the next element in the iteration
        raise StopIteration exception if we are at the end
        """
        if (self.iterPoz>=len(self.l)):
            raise StopIteration()

        rez = self.l[self.iterPoz]
        self.iterPoz = self.iterPoz +1
        return rez

def testIn():
    container = MyClass2()
    for i in range(0,200):
        container.add(MyClass(i,"ad"))
    findObj = MyClass(20,"asdasd")
    print findObj in container
```

# In Python

```
def measureBinary(e, l):
    sw = StopWatch()
    poz = searchBinaryRec(e, l)
    print "    BinaryRec in %f sec; poz=%i" %(sw.stop(),poz)

def measurePythonIndex(e, l):
    sw = StopWatch()
    poz = -2
    try:
        poz = l.index(e)
    except ValueError:
        pass #we ignore the error..
    print "    PythIndex in %f sec; poz=%i" %(sw.stop(),poz)

def measureSearchSeq(e, l):
    sw = StopWatch()
    poz = searchSeq(e, l)
    print "    searchSeq in %f sec; poz=%i" %(sw.stop(),poz)
```

```
search 200
    BinaryRec in 0.000000 sec; poz=200
    PythIndex in 0.000000 sec; poz=200
     PythonIn in 0.000000 sec
    BinaryNon in 0.000000 sec; poz=200
    searchSuc in 0.000000 sec; poz=200
```

```
search 10000000
    BinaryRec in 0.000000 sec; poz=10000000
    PythIndex in 0.234000 sec; poz=10000000
     PythonIn in 0.238000 sec
    BinaryNon in 0.000000 sec; poz=10000000
    searchSuc in 2.050000 sec; poz=10000000
```

# Sortierproblem

- **Eingabe:** Folge von Zahlen `a,n,<a1,a2,...,an>`

- **Ausgabe:** sortierte Folge der Eingabe `<a1',a2',...,an'>` mit `a1'≤a2'≤...≤an'`

- Eingabemenge als Feld oder verkettete Liste repräsentiert

- Sortierverfahren lösen das durch die Eingabe-Ausgabe-Relation beschriebene Sortierproblem

# Struktur der Daten

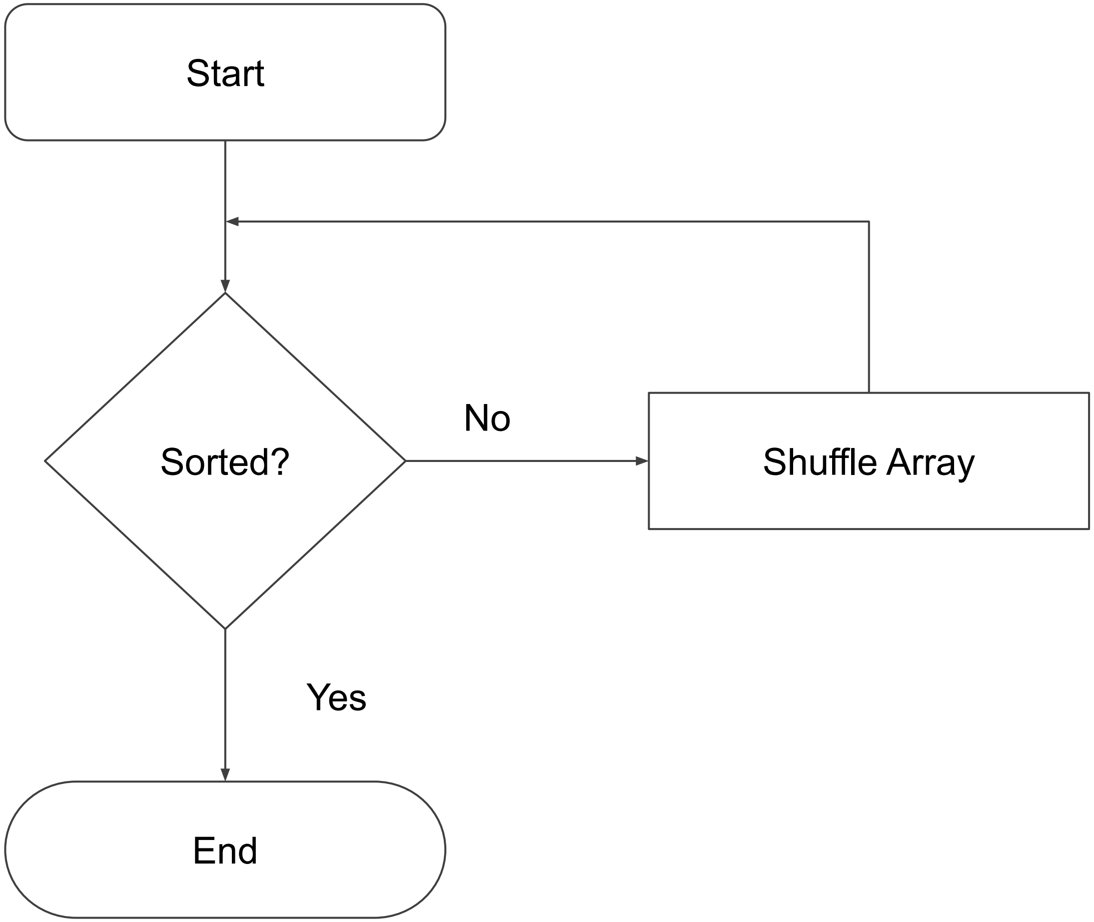
- zu sortierende Werte (Schlüssel) sind selten isoliert
  - sondern Teil einer größeren Datenmenge (Objekte, Datensatz/Record)
- Daten bestehen aus Schlüssel und Satellitendaten
- Satellitendaten werden mit Schlüssel umsortiert
- wie beim Suchen werden Satellitendaten ignoriert

# Eigenschaften

- Effizienz
  - Best-, Average-, Worst-Case
- Speicherbedarf
  - in-place
  - out-of-place
    - braucht zusätzlichen Speicher
- rekursiv oder iterativ
- Stabilität
  - stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht
- verwendete Operationen
  - Vertauschen, Auswählen, Einfügen
- Verwendung spezieller Datenstrukturen

# Gigachad Sorting: Bogosort

## Complexity: Yes



# Sortieren von Spielkarten

- Bubble Sort
  - Aufnehmen aller Karten vom Tisch
  - vertausche ggf. benachbarte Karten, bis Reihenfolge korrekt

- Selection Sort
  - Aufnehmen der jeweils niedrigsten Karte vom Tisch
  - Anfügen der Karte am Ende des sortierten Teil

- Insertion Sort
  - Aufnehmen einer beliebigen Karte
  - Einfügen der Karte an der korrekten Position

# Bubble-Sort

- durchlaufe die Menge
- vertausche zwei aufeinanderfolgende Elemente, wenn ihre Reihenfolge nicht stimmt
- durchlaufe die Menge gegebenenfalls mehrmals, bis bei einem Durchlauf keine Vertauschungen mehr durchgeführt werden mussten

# Bubble-Sort

sortiert = true

| | | | | | |
|---|---|---|---|---|---|
| 55 | 7 | 78 | 12 | 42 | 55<7? tausche(7,55); sortiert = false; |
| 7 | 55 | 78 | 12 | 42 | 55<78? |
| 7 | 55 | 78 | 12 | 42 | 78<12? tausche(78,12); |
| 7 | 55 | 12 | 78 | 42 | 78<42? tausche(78,42); Ende: sortiert? sortiert=true; |
| 7 | 55 | 12 | 42 | 78 | 7<55? |
| 7 | 55 | 12 | 42 | 78 | 55<12? tausche(55,12); sortiert=false; |
| 7 | 12 | 55 | 42 | 78 | 55<42? tausche(55,42); |
| 7 | 12 | 42 | 55 | 78 | 55<78? Ende: sortiert? sortiert=true; |
| 7 | 12 | 42 | 55 | 78 | 7<12? |
| 7 | 12 | 42 | 55 | 78 | 12<42? |
| 7 | 12 | 42 | 55 | 78 | 42<55? |
| 7 | 12 | 42 | 55 | 78 | 55<78? Ende: sortiert? Fertig. |

# Bubble-Sort

6 5 3 1 8 7 2 4

# Python

```
def bubble(arr):




        for i in range(1, len(arr)):
            if arr[i-1] > arr[i]:
                arr[i], arr[i-1] = arr[i-1], arr[i]
```

# Python

```
def bubble(arr):
    sorted = False

    while not sorted:
        sorted = True

        for i in range(1, len(arr)):
            if arr[i-1] > arr[i]:
                arr[i], arr[i-1] = arr[i-1], arr[i]
                sorted = False
```

# Eigenschaften

- iterativ
- stabil
  - (gleiche benachbarte Schlüssel werden nicht getauscht)
- in-place
  - (konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen

# Laufzeit

- schlechtester Fall
  - Eingabe ist umgekehrt sortiert: `n, n-1, …, 2, 1`
  - `n-1` Vertauschungen im ersten Durchlauf
  - `n-2` Vertauschungen im zweiten Durchlauf
  - …
  - `1` Vertauschung im n-ten Durchlauf

$$T(n) = n(n-1)/2 \in O(n*n)$$

- bester Fall
  - Eingabe ist sortiert: `1,2, … n-1, n`
  - ein Durchlauf `O(n)`

# Selection-Sort

- durchlaufe die Menge, finde das kleinste Element
- vertausche das kleinste Element mit dem ersten Element
- vorderer Teil ist sortiert (`k` Elemente nach Durchlauf `k`), hinterer Teil ist unsortiert (`n-k` Elemente)
- durchlaufe die hintere, nicht sortierte Teilmenge, finde das `n`-kleinste Element
- vertausche das n-kleinste Element mit dem `n`-ten Element

# Selection-Sort

| a[1] | a[2] | a[3] | a[4] | a[5] | |
|---|---|---|---|---|---|
| 55 | 7 | 78 | 12 | 42 | 1. Durchlauf: 7 = min(1..n); tausche 7 mit a[1]; |
| 7 | 55 | 78 | 12 | 42 | 2. Durchlauf: 12 = min(2..n); tausche 12 mit a[2]; |
| 7 | 12 | 78 | 55 | 42 | 3. Durchlauf: 42 = min(3..n); tausche 42 mit a[3]; |
| 7 | 12 | 42 | 55 | 78 | 4. Durchlauf: 55 = min(4..n); tausche 55 mit a[4]; |

# Selection-Sort

5 3 4 1 2

# Python

```
def selection (l):

        i = ?, idx = i
        for j in range(i + 1, len(l)):
            if l[j] < l[idx]:
                idx = j
```

# Python

```
def selection (l):

        i = ?, idx = i
        for j in range(i + 1, len(l)):
            if l[j] < l[idx]:
                idx = j

        l[i], l[idx] = l[idx], l[i]
```

# Python

```
def selection (l):
    for i in range(len(l) - 1):
        idx = i
        for j in range(i + 1, len(l)):
            if l[j] < l[idx]:
                idx = j

        l[i], l[idx] = l[idx], l[i]
```

# Eigenschaften

- iterativ
- instabil
  - gleiche benachbarte Schlüssel werden getauscht
  - lässt sich auch stabil implementieren
- in-place
  - konstanter zusätzlicher Speicheraufwand

# Laufzeit

- bester, mittlerer, schlechtester Fall
  - für `n` Einträge werden `n-1` Minima gesucht
  - `n-1` Vergleiche für erstes Minimum
  - `n-2` Vergleiche für zweites Minimum
  - …
  - `1` Vergleich für Minimum `n-1`
  - `T(n) = n(n-1)/2` ∈ `O(n*n)`

# Insertion-Sort

- erstes Element ist sortiert, hinterer Teil mit `n-1` Elementen ist unsortiert
- entnehme der hinteren, unsortierten Menge ein Element und füge es an die richtige Position der vorderen, sortierten Menge ein (`n-1` mal)
- Einfügen in die vordere, sortierte Menge erfordert das Verschieben von Elementen

# Insertion-Sort

| | | | | | | |
|---|---|---|---|---|---|---|
| 5 | 2 | 4 | 6 | 1 | 3 | Elemente 1..1 sind sortiert, 2..n unsortiert |
| 5 | 2 | 4 | 6 | 1 | 3 | Vergleiche 2 mit allen Elementen der sortierten Menge beginnend mit dem größten. Wenn ein Element größer als 2 ist, schiebe es eins nach rechts, sonst füge 2 ein. |
| 2 | 5 | 4 | 6 | 1 | 3 | Elemente 1..2 sind sortiert, 3..n unsortiert |
| 2 | 5 | 4 | 6 | 1 | 3 | Vergleiche 4 mit allen Elementen der sortierten Menge. Wenn ein Element größer als 4 ist, verschiebe es. |
| 2 | 4 | 5 | 6 | 1 | 3 | Elemente 1..3 sind sortiert. Einfügen von 6. |
| 2 | 4 | 5 | 6 | 1 | 3 | Elemente 1..4 sind sortiert. Einfügen von 1 (Dazu werden Elemente 6,5,4 und 2 jeweils um eins nach rechts verschoben. Danach wird 1 an Pos. eins eingefügt. |
| 1 | 2 | 4 | 5 | 6 | 3 | … |
| 1 | 2 | 3 | 4 | 5 | 6 | |

# Insertion-Sort

6 5 3 1 8 7 2 4

# Insertion-Sort

```
def insert(l):
        pos = ?
        idx = pos - 1
        el = l[pos]

        while idx >= 0 and el < l[idx]:
            l[idx + 1] = l[idx]
            idx = idx - 1
```

# Insertion-Sort

```
def insert(l):
        pos = ?
        idx = pos - 1
        el = l[pos]

        while idx >= 0 and el < l[idx]:
            l[idx + 1] = l[idx]
            idx = idx - 1

        l[idx + 1] = el
```

# Insertion-Sort

```
def insert(l):
    for i in range(1, len(l)):
        idx = i - 1
        el = l[i]

        while idx >= 0 and el < l[idx]:
            l[idx + 1] = l[idx]
            idx = idx - 1

        l[idx + 1] = el
```

# Eigenschaften

- iterativ
- stabil
    - gleiche benachbarte Schlüssel werden nicht getauscht
- in-place
    - konstanter zusätzlicher Speicheraufwand
- effizient für vorsortierte Mengen

# Laufzeit

- bester Fall
  - Menge ist vorsortiert
  - innere while-Schleife wird nicht durchlaufen
  - `O(n)`

- schlechtester Fall
  - Menge ist umgekehrt sortiert
  - `k-1` Verschiebeoperationen für das `k`-te Element
  - `O(n*n)`

# Quick-Sort

- Problem
  - Elemente müssen teilweise über große Bereiche verschoben werden

- Mengen mit 1, 0 Elemente sind sortiert
- ansonsten
  - Aufteilung des Problems in zwei Teilmengen, wobei alle Elemente einer Teilmenge kleiner als alle Elemente der anderen Teilmenge sind
  - rekursiver Aufruf des Algorithmus für beide Teilmengen

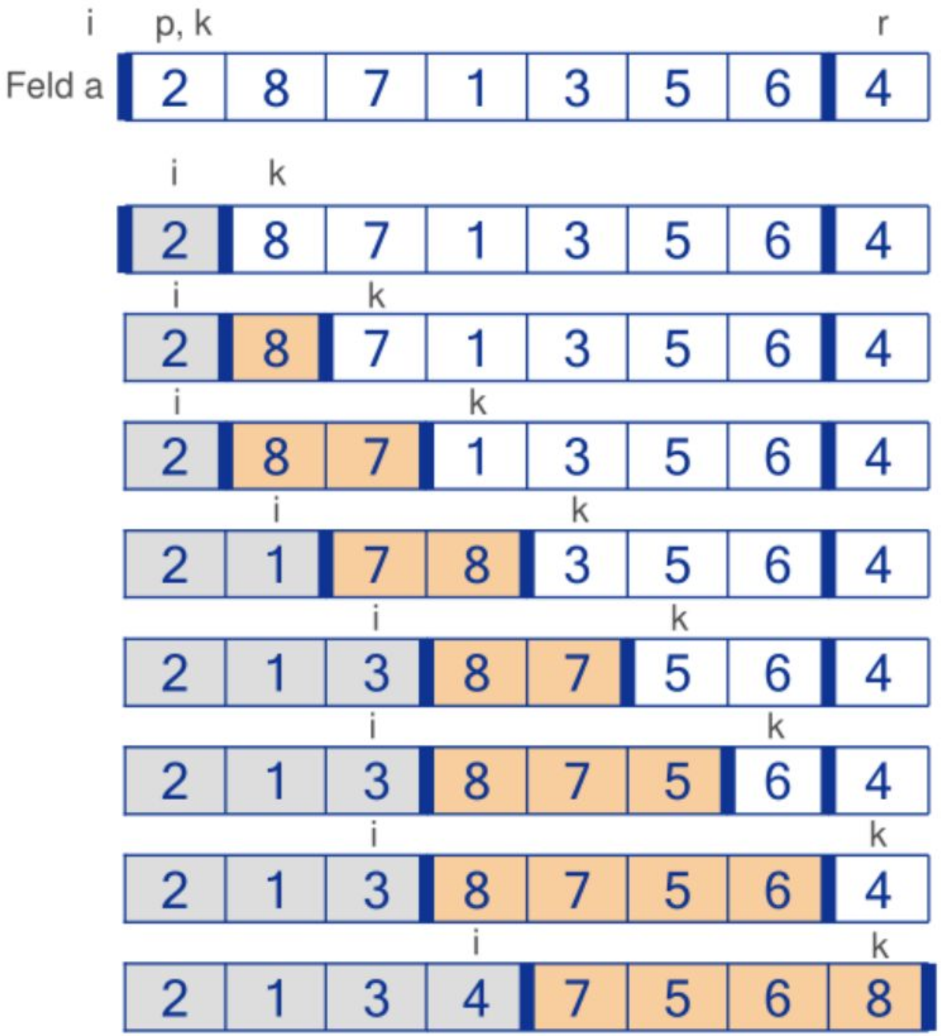
# Quick-Sort

```
Quicksort (L)
    if (|L| <= 1) return L;
    else
        waehle Pivotelement p aus L;
        L1 = { a in L | a < p };
        L2 = { a in L | a > p };
        Quicksort (L1);
        Quicksort (L2);
```

Aufteilung in zwei Teilmengen L1, L2


kein merge-Schritt, da in-place-Sortierung:
L1 < p < L2

# Quick-Sort




p – erstes Element,
r – letztes Element (Pivotelement)

i – letztes Element von L1
k – erstes Element hinter L2

a [ k ] <= a [ r ] → i+=1; tausche ( a [ i ], a [ k ] ); k+=1;

a [ k ] > a [ r ] → k+=1;

a [ k ] > a [ r ] → k+=1;

a [ k ] <= a [ r ] → i+=1; tausche ( a [ i ], a [ k ] ); k+=1;

a [ k ] <= a [ r ] → i+=1; tausche ( a [ i ], a [ k ] ); k+=1;

a [ k ] > a [ r ] → k+=1;

a [ k ] > a [ r ] → k+=1;

i+=1; tausche ( a [ i ], a [ k ] );

# Quick-Sort

```
def partition (arr, start, end):
    pivot = arr[end]

    i = start - 1
    k = start

    while k <= end - 1:
        if arr[k] < pivot:
            i += 1
            arr[i], arr[k] = arr[k], arr[i]

        k += 1

    arr[i+1], arr[end] = arr[end], arr[i+1]

    return i + 1
```

```
def quick_sort (arr, start, end):
    if start < end:
        p=partition(arr,start,end)

        quick_sort(arr,start,p-1)
        quick_sort(arr,p+1,end)
```

# Quick-Sort

- rekursiv
- in-place
  - konstanter zusätzlicher Speicheraufwand
- nicht stabil
- optimale mittlere Laufzeit von O (n log n)
- ungünstige schlechteste Laufzeit von O (n*n )

# Quick-Sort

bester Fall:

Aufteilung

n $\rightarrow$ (n/2) + (n/2)

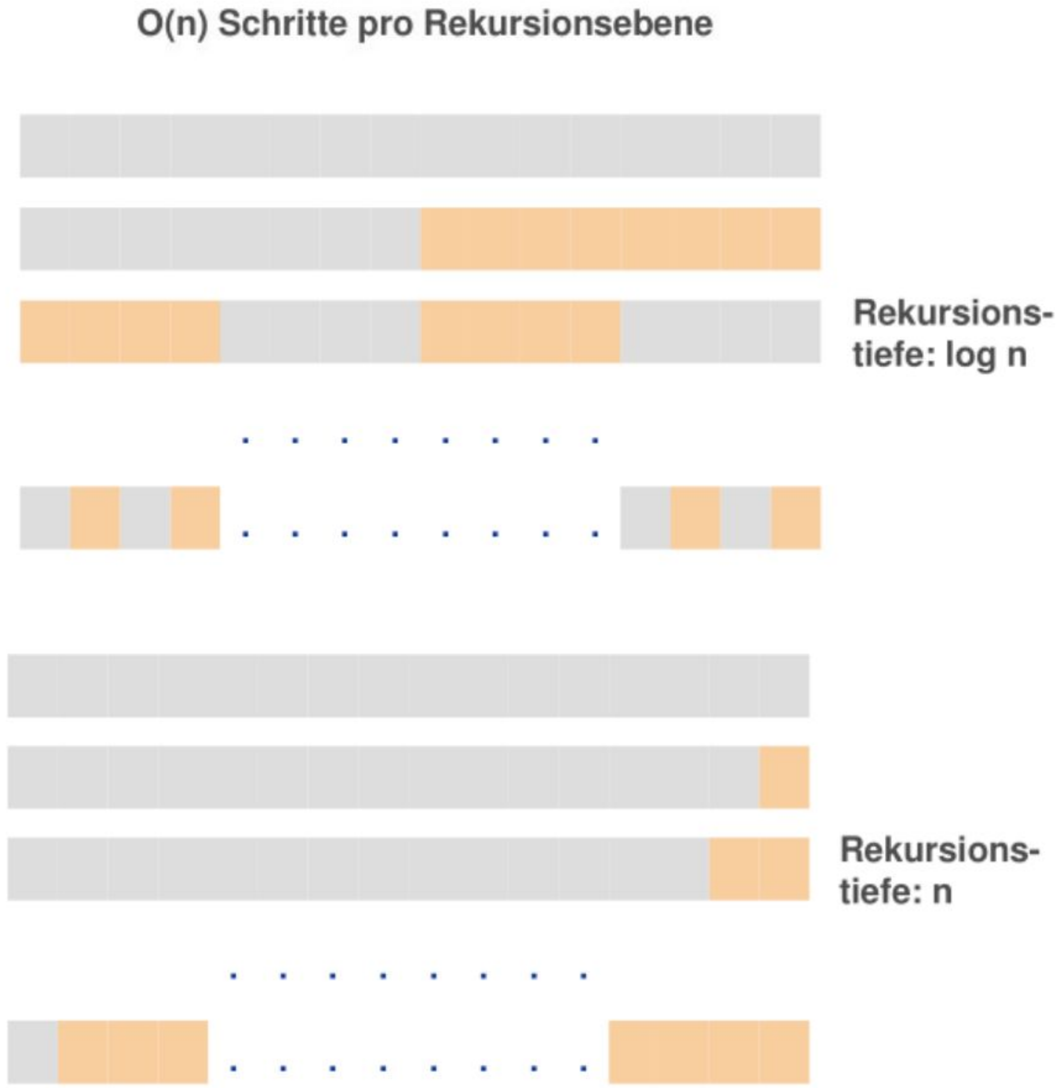
O(n log n)

Schlechtester Fall:

Aufteilung

n $\rightarrow$ (n-1) + (1)

O(n*n)



# Merge-Sort

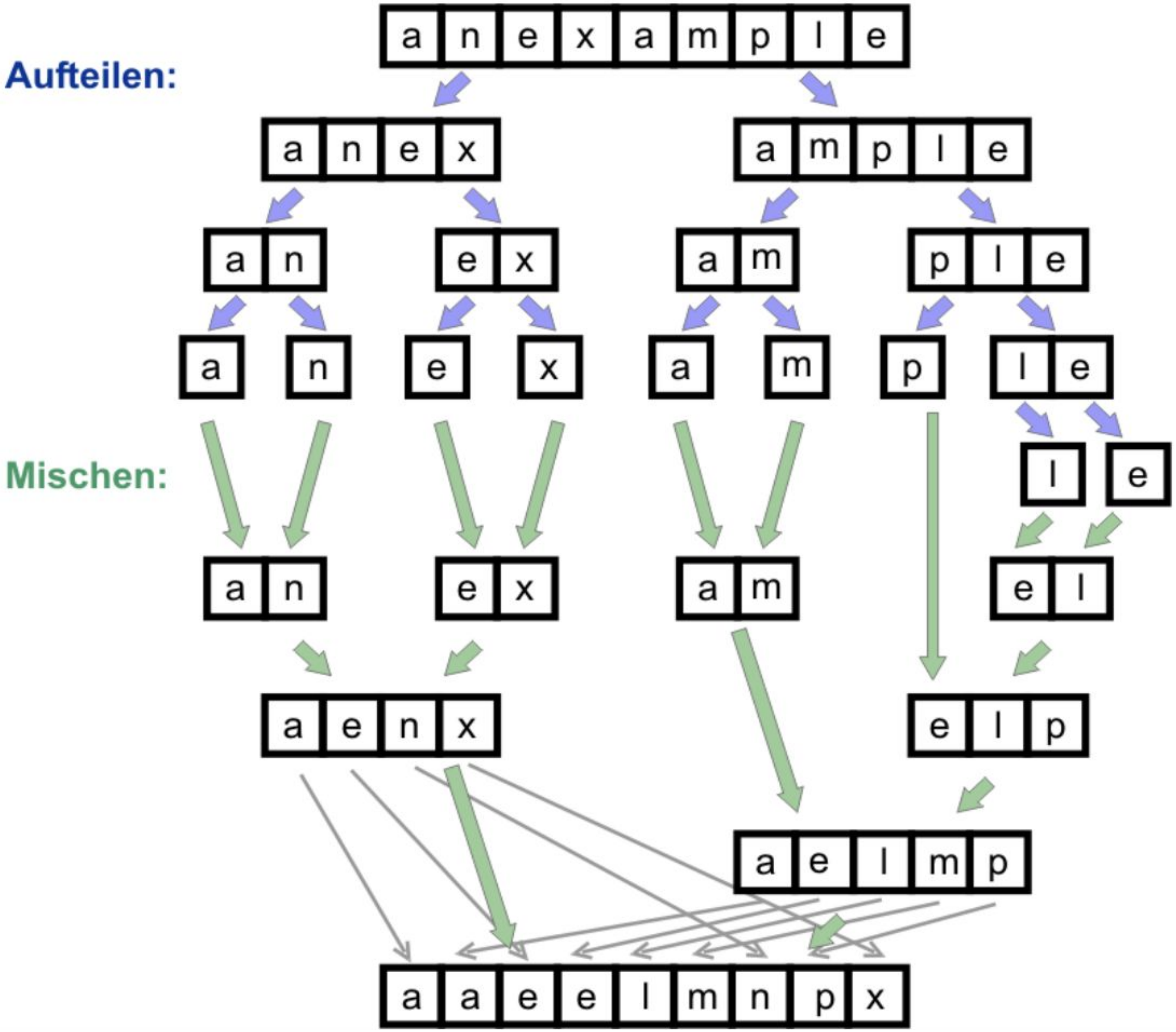
- Teile-und-Herrsche-Ansatz
  - Mengen mit 1, 0 Elemente sind sortiert

- ansonsten
  - Aufteilung des Problems in zwei gleich große Teilmengen
  - rekursiver Aufruf des Algorithmus für beide Teilmengen
  - Verbindung der Teilmengen

# Merge-Sort

- Motivation gegenüber Quick-Sort
- simple Unterteilung im Vergleich zu Quick-Sort
- optimale Rekursionstiefe von log
- aufwendiger Merge-Schritt

# Merge-Sort


Aufteilen:
a n e x a m p l e
a n e x
a m p l e
a n
e x
a m
p l e
a
n
e
x
a
m
p
l e
l
e
Mischen:
a n
e x
a m
e l
a e n x
e l p
a e l m p
a a e e l m n p x

# Merge-Sort

```
def merge(left, right):
    l = []
    i = 0
    j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            l.append(left[i])
            i = i + 1

        else:
            l.append(right[j])
            j = j + 1


    l += left[i:]
    l += right[j:]

    return l
```

```
def merge_sort(arr):
    if len(arr) == 1:
        return arr

    mid = len(arr) // 2

    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)
```

# Merge-Sort

- rekursiv
- je nach Implementierung in-place oder
- zusätzlicher Speicheraufwand von O(n) oder O(n log n)
- stabil
- optimale mittlere Laufzeit von O(n log n)
- beste und schlechteste Laufzeit von O(n log n)
    - im Gegensatz zu Quick-Sort immer garantiert
    - aufteilung in zwei gleichgroße Teilprobleme

# Zusammenfassung

| Verfahren | bester Fall O (x) | mittlerer Fall O (x) | schlechtester Fall O (x) | stabil | rekursiv | Speicher O (x) |
|---|---|---|---|---|---|---|
| Bubble | n | $n^2$ | $n^2$ | ja | nein | 1 |
| Selection | $n^2$ | $n^2$ | $n^2$ | ja | nein | 1 |
| Insertion | n | $n^2$ | $n^2$ | ja | nein | 1 |
| Heap | n log n | n log n | n log n | nein | nein | 1 |
| Quick | n log n | n log n | $n^2$ | nein | ja | 1 |
| Merge | n log n | n log n | n log n | ja | ja | n |
| Counting | n | n | n | ja | nein | n |
| Radix | n | n | n | ja | nein | n |
| Bucket | n | n | n | ja | nein | n |