Such + Sortieralgorithmen I

Inhalt

- Komplexität
- Search
- Sort

Beurteilung von Algorithmen

- viele Algorithmen, um dieselbe Funktion zu realisieren

- Welche Algorithmen sind die besten?

- 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

Ressourcenbedarf

- Prozesse verbrauchen:
  - Rechenzeit
  - Speicherplatz

- Die Ausführungszeit hängt ab von:
  - der konkreten Programmierung
  - Prozessorgeschwindigkeit
  - Programmiersprache
  - Qualität des Compilers

Beispiel

```python
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)
```

```python
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
```

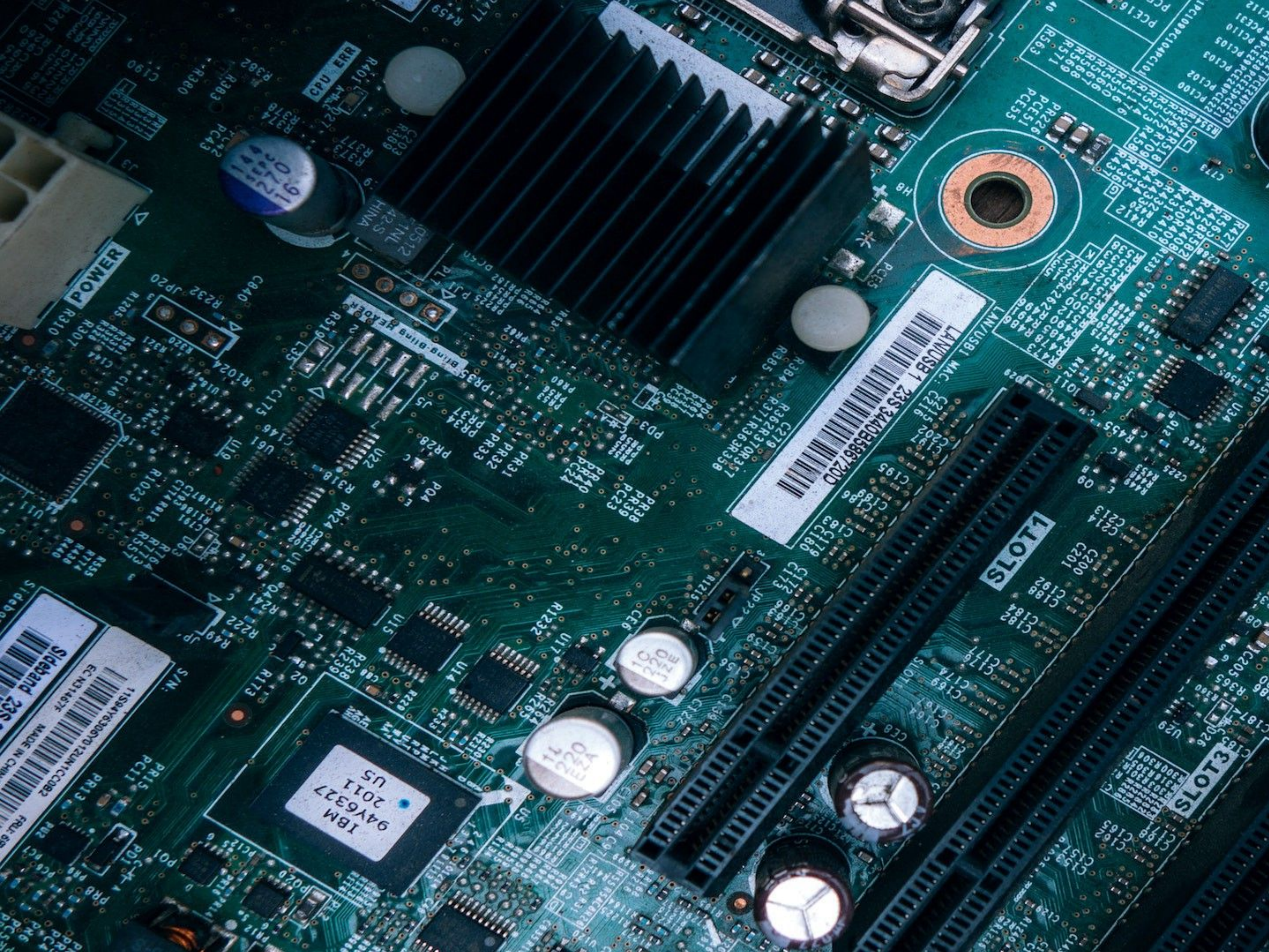
```python
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    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

Beispiel

```python
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)
```


POWER
CPU ERR
SLOT1
SLOT3
IBM
94Y6327
2011
US

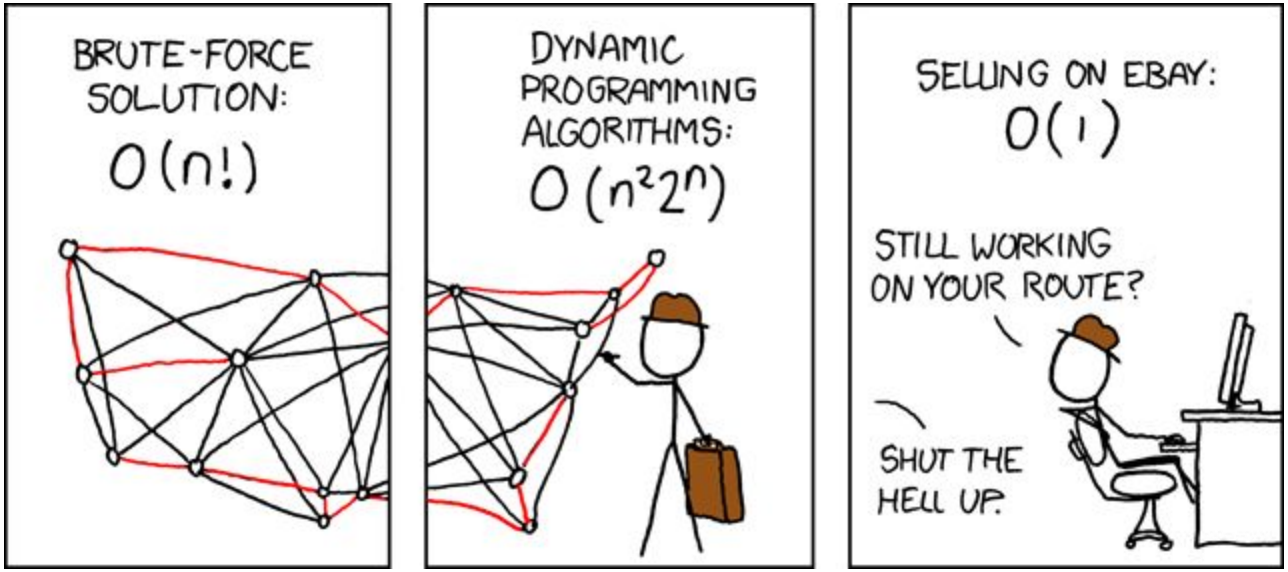
Leistungsverhalten

- **Speicherplatzkomplexität:** Wird primärer & sekundärer Speicherplatz effizient genutzt?
- **Laufzeitkomplexität:** Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- **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

```python
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

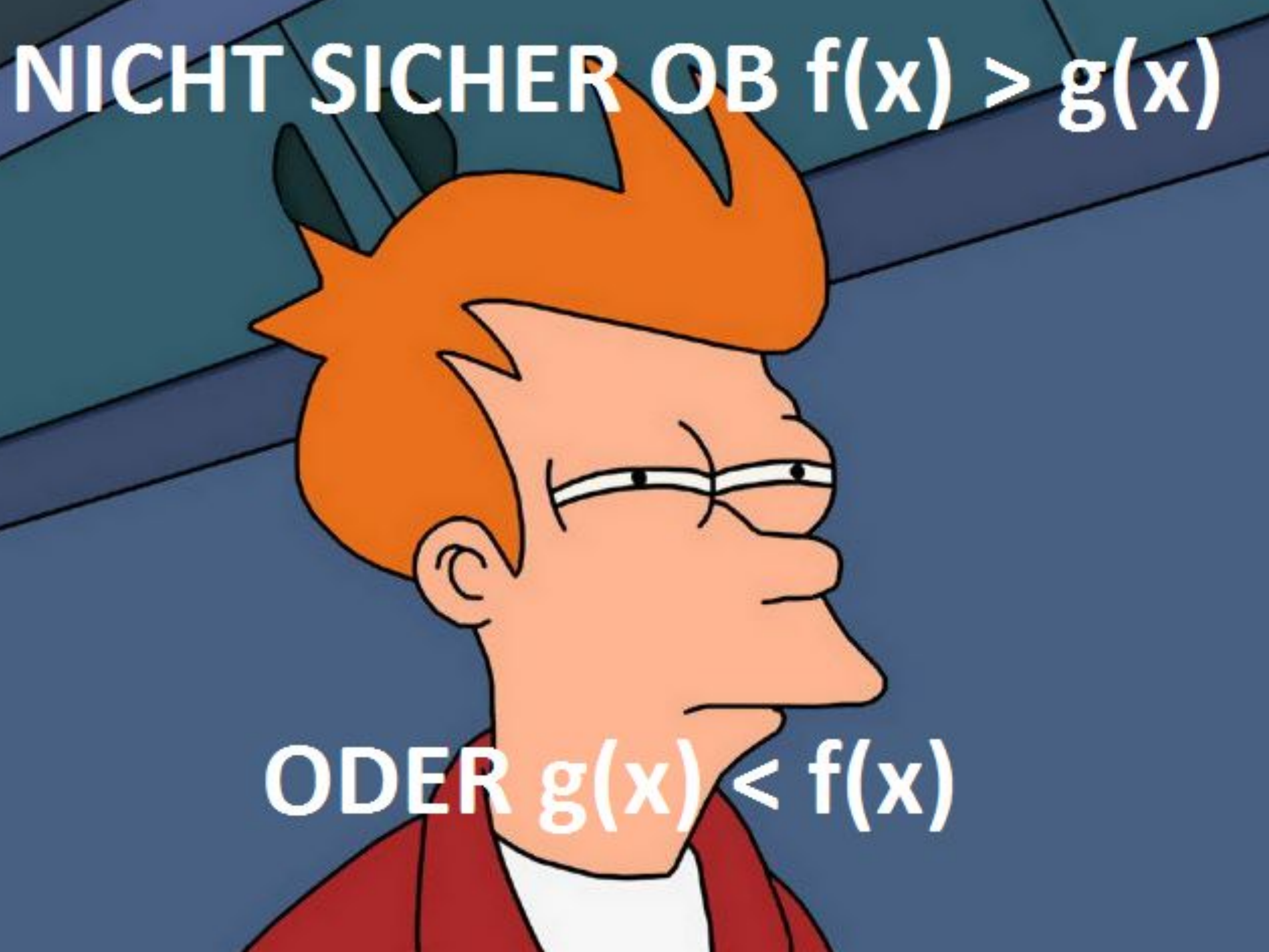
Zeit:

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

n = Größe des Arrays

Asymptotische Komplexität

- Laufzeit und Speicherverbrauch wird in einer asymptotischen Notation beschrieben, die weitgehend von unwesentlichen Details abstrahiert
- wir machen nur Aussagen über das Verhalten für **sehr große** Eingabegrößen
- vergleich von zwei Komplexitäten über alle natürlichen Zahlen ist nicht ganz einfach

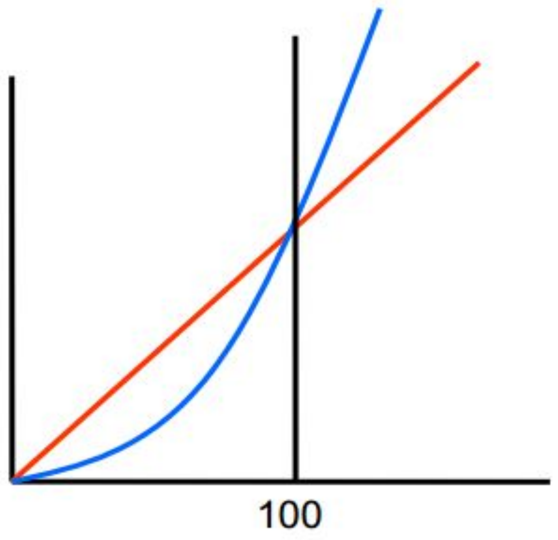

NICHT SICHER OB f(x) > g(x)
ODER g(x) < f(x)

Asymptotische Komplexität

- Vergleich von zwei Komplexität Funktionen über alle natürlichen Zahlen ist nicht ganz einfach

- Wir übernehmen eine mathematische Notation, die zum Vergleichen von Funktionen bis auf einen Faktor verwendet wird

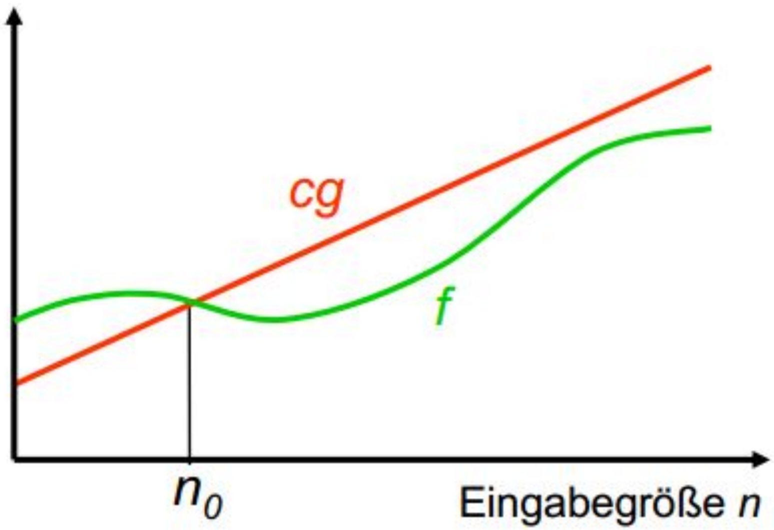
$T_1(n) = 100 \cdot n$
$T_2(n) = n^2$



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 ∈ N` gibt, so dass für alle `n ≥ n0 f(n) ≤ c*g(n)` gilt



Asymptotische Komplexität

- O-Notation: Abstraktion durch
  - ignorieren endlich vieler Anfangswerte (Spezialfälle) durch
    `n ≥ n0`
  - Einführung des konstanten Faktors `c` in der Definition, der von nur durch Konstanten hervorgerufenen Unterschieden abstrahiert
- Beispiel:
  - `T1(n) = 100 * n ∈ O(n): T1(n)` wächst höchstens so schnell wie `n`
  - `T2(n) = n*n ∈ O(n*n): T2(n)` wächst höchstens so schnell wie `n*n`

Asymptotische Komplexität

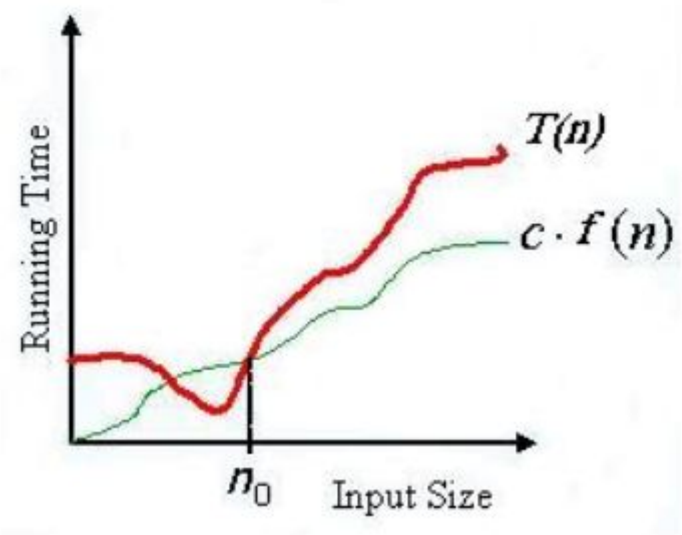
Häufige Größenordnung der Komplexität:

- `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)`.

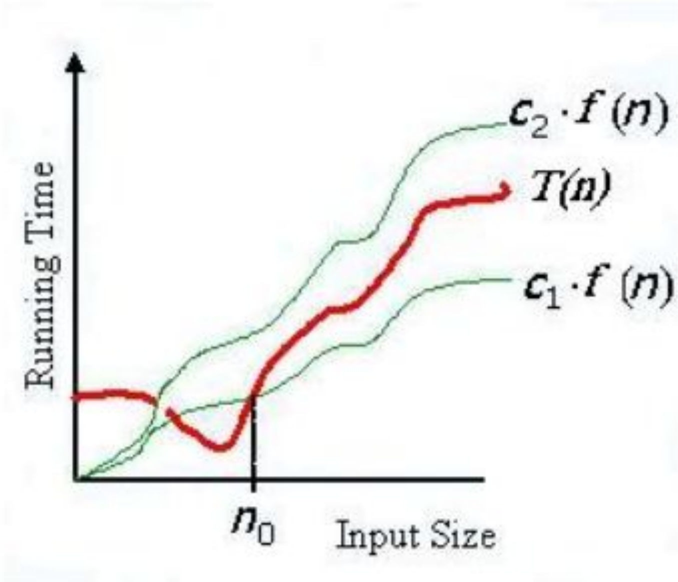
`f(n)` ist in `Ω(g(n))`, wenn es ein `c > 0` und ein `n0 ∈ N` gibt, so dass für alle `n ≥ n0 f(n) ≥ c*g(n)` gilt



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

```
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

```
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

```
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

- Verfahren, das in einem Suchraum nach Mustern oder Objekten mit bestimmten Eigenschaften sucht

- **vielfältige Anwendungsbereiche**
  - Suchen in Datenbanken, Google-Search
  - Suchen nach ähnlichen Mustern: z.B. Viren, Malware
  - Bilderkennungsverfahren: Suchen nach Pattern

- **für uns**: einfache Suchverfahren auf Listen

Anforderungen

- statische, kleine Menge, selten Operations notwendig
  - **Lösung**: Feld als Datenstruktur und sequentielles Suchen `O(n)`
- statische, kleine Menge, häufige Operations
  - **Lösung**: Vorsortiertes `O(nlogn)` Feld, binäres Suchen `O(logn)`
- dynamisch, große Menge von Elementen
  - **Lösung:** Baum als dynamische Datenstruktur, organisiert als binärer Suchbaum `O(h)`, h ist Baumhöhe. Worst-Case: `h = n`, Best-Case: `h=logn` **(balanciert)**
- dynamisch, große Menge, viele, effiziente Zugriffe notwendig
  - **Lösung:** Binärer Suchbaum, der eine möglichst geringe Höhe h garantiert: z.B.**B-Baum**

Charakteristiken

- Eingabe: Folge von Zahlen `a, n, <a1,a2,...,an>`
  - Vorbedingung: $n \in N$, $n \geq 0$;
- Ausgabe: `p`
  - Nachbedingungen : $(0 \leq p \leq n-1$ 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 ausgelagerten 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 Such-Schlüssel `k`

```python
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
```

```python
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

```
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

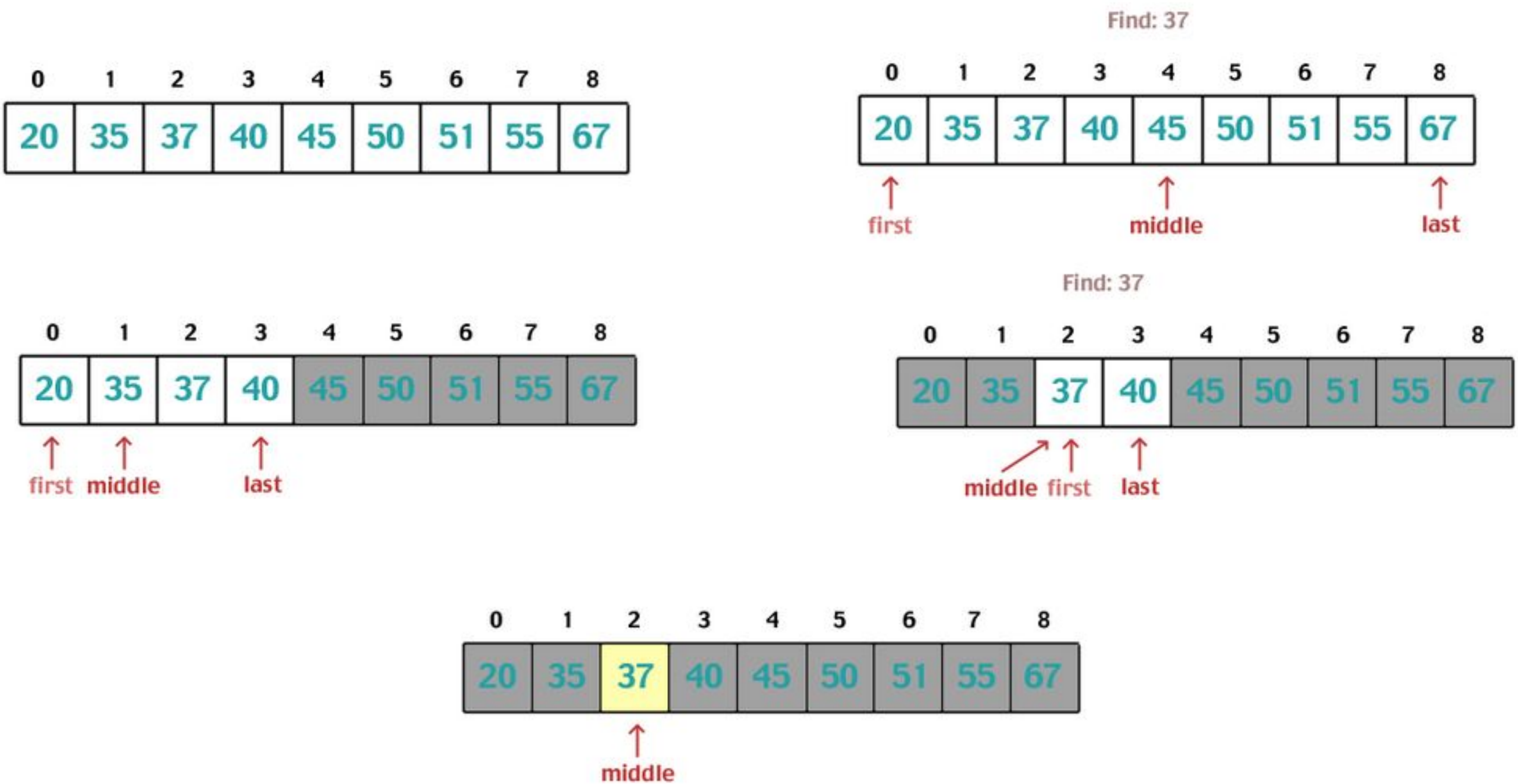
Laufzeitanalyse

- **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

Binäre Suche

```
def searchBinaryNonRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a 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)
    right=len(l)
    left = 0
    while right-left>1:
        m = (left+right)/2
        if el<=l[m]:
            right=m
        else:
            left=m
    return right
```

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 (Datensatz, Record)
- Daten bestehen aus Schlüssel und Satellitendaten
- Satellitendaten werden mit Schlüssel umsortiert
- im Folgenden werden Satellitendaten ignoriert

Eigenschaften

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

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

- 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.

Python

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` ∈ `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
7	55	78	12	42
7	12	78	55	42
7	12	42	55	78

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

Python

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` $\in$ `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

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)`

Nächste Woche

- weitere Algorithmen
- Quicksort
- Mergesort
- Counting
- Bucket