

---

*Einführung in die Informatik*

---

Algorithmen zum Suchen und Sortieren

Meik Teßmer

---

# Inhalte

# Lernziele

„Wenn Daten schon das neue Gold sind, wie finde ich dann die Nuggets?“

- Suchen und Sortieren: Möglichkeiten
- Iteration vs. Rekursion
  - Vorteile, Nachteile, Anwendungen

# Algorithmus?

„Was ist der berühmte *Algorithmus*, von dem in den Medien immer die Rede ist?“

# Arbeitsdefinition

Ein *Algorithmus* ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. Algorithmen bestehen aus endlich vielen, wohldefinierten Einzelschritten.

# Suchen

# Erstes Beispiel

## ***Suchproblem***

*Finden einer bestimmten Telefonnummer im Mobiltelefon*

## Suchen: Lineare Suche

- Ansatz: so lange suchen, bis das Gesuchte gefunden ist
- Vorteile:
  - funktioniert immer
  - Liste muss nicht sortiert sein
- Nachteil:
  - Laufzeit: abhängig von der Länge der Liste
- Umsetzung: *Iteration*

# Lineare Suche: Variante 1

Frage: Ist ein bestimmtes Element in der Liste vorhanden?

```
1 def ist_vorhanden(daten, gesucht):
2     for datum in daten:
3         if datum==gesucht:
4             return True
5
6     return False
7
8 meine_daten = ["b", "f", "d", "k"]
9 gesuchtes_datum = "k"
10 ist_vorhanden(meine_daten, gesuchtes_datum)
```

## Lineare Suche: Variante 2

Frage: Wie oft ist ein bestimmtes Element in der Liste vorhanden?

```
1 def auftreten(daten, gesucht):
2     anzahl_auftreten = 0
3
4     for datum in daten:
5         if datum==gesucht:
6             anzahl_auftreten += 1
7     return anzahl_auftreten
8
9 meine_daten = ["b", "f", "d", "k"]
10 gesuchtes_datum = "k"
11 auftreten(meine_daten, gesuchtes_datum)
```

Das ist eine *vollständige Suche*.

# Grenzen der vollständigen Suche

bei großen Datenmengen:

- schon Suche nach Vorhandensein dauert u.U. sehr lange
- vollständige Suche kaum möglich

Verbesserungsvorschläge?

## Informierte Suche: Binäre Suchstrategie

- Ansatz: Teile die (Teil-)Liste und suche in der richtigen
- Vorteil: sehr effizient auch für große Datenmengen
- Nachteil: Daten müssen sortiert vorliegen
- Umsetzung: Iteration oder Rekursion

# Binäre Suche: Konzept

gesucht: G

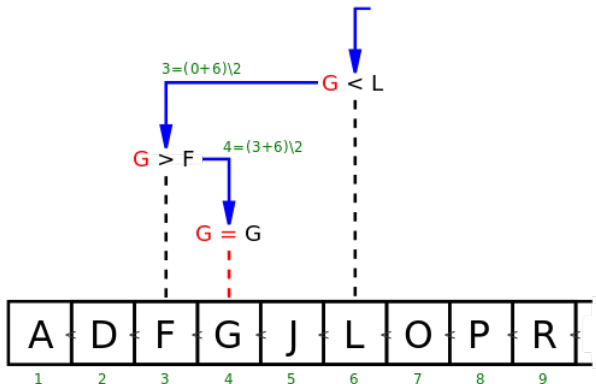


Abbildung 1: Binäre Suche.

# Binäre Suche: Iterative Lösung

Frage: Ist ein bestimmtes Element in der Liste vorhanden?

```
1 def binäre_suche(daten, gesucht):
2     index = 0
3     maxindex = len(daten) - 1 # wg. Start bei 0
4     while index <= maxindex:
5         mitte = index + ((maxindex - index) // 2)
6         if daten[mitte]==gesucht:
7             return mitte
8         elif gesucht < daten[mitte]:
9             # linke Teilliste
10            maxindex = mitte - 1 # wg. echt kleiner
11        else: # rechte Teilliste
12            index = mitte + 1 # wg. echt größer
13    return None
```

# Sortieren

# Sortier-Algorithmen

- verschiedene Algorithmen: Gnomesort, Quicksort, Bubblesort usw.
- unterscheiden sich hinsichtlich Geschwindigkeit, Ressourcenverbrauch
- Effizienz hängen ab von der Datenmenge und ihrer Vorsortierung

## Beispiel: Gnomesort

- Gartenzwerg soll unterschiedliche große Blumentöpfe ordnen
- Ordnung: von links (kleinster Topf) nach rechts (größter Topf)
- Verfahren:
  - vergleicht den Topf, vor dem er gerade steht, und den rechts daneben
  - passt die Reihenfolge, geht er einen Topf nach rechts
  - passt sie nicht, vertauscht er die Reihenfolge und macht einen Schritt nach links
  - kann er nicht weiter nach links gehen, geht er einen Schritt nach rechts (linker Rand)
  - ist er am letzten Topf angekommen, endet der Algorithmus (rechter Rand)

## Beispiel: Gnomesort

```
1  def gnomesort(liste):
2      pos = 0
3      while True:
4          if pos == 0:
5              pos += 1
6          if pos > len(liste): # rechter Rand erreicht?
7              break
8          if liste[pos] >= liste[pos-1]: # zwei Töpfe vergleichen
9              pos += 1 # Sortierung passt; Schritt nach rechts gehen
10         else:
11             # Töpfe vertauschen
12             liste[pos-1], liste[pos] = liste[pos], liste[pos-1]
13             pos -= 1 # Schritt nach links gehen
14     return liste
```

# Analyse

- Laufzeit: quadratisch zur Länge der Liste  
→ recht ineffizient im Vergleich zu anderen Verfahren
- konstanter und geringer Speicherverbrauch

# Bubblesort

- Liste wird von links nach rechts durchlaufen
- das aktuelle Element wird mit dem rechts daneben verglichen
- passt die Sortierung nicht, werden sie getauscht
- passt sie, gehe einen Schritt nach rechts und vergleiche wieder...
- am Ende des Durchlaufs steht das größte Element am Ende der Liste
- große Elemente/Blasen „blubbern“ nach oben (rechts)

[Animation bei Wikipedia](#)

## Beispiel: Bubblesort

```
1  def bubblesort(liste):
2      länge_restliste = len(liste)
3
4      while länge_restliste >= 1:
5          # Restliste durchlaufen
6          for k in range(länge_restliste-1):
7              # Elemente korrekt geordnet?
8              if liste[k] > liste[k+1]:
9                  # nein; Elemente vertauschen
10                 liste[k], liste[k+1] = liste[k+1], liste[k]
11                 # letztes Element ist an passender Stelle
12                 # also kann man das weglassen
13                 länge_restliste -= 1
14     return liste
```

# Analyse

- Laufzeit: quadratisch zur Länge der Liste (Durchschnitt und Worst-Case)

→ ebenfalls recht ineffizient im Vergleich zu anderen Verfahren

- Implementation ist einfacher, schlanker

Ist aber alles nicht optimal. Geht es besser?

# Quicksort

- Ansatz: Liste in eine rechte und linke Teilliste teilen, dann jeweils sortieren
- Teilung erfolgt anhand eines frei gewählten *Pivotelements*
- Liste durchgehen und alle Elemente  $<$  Pivot in die linke Teilliste stecken, alle  $\geq$  Pivot in die rechte
  - es gilt damit: Elemente der linken Teilliste  $<$  Elemente der rechten Teilliste
- die Teillisten werden dann wieder sortiert
- Abbruch erfolgt dann, wenn eine Teilliste leer ist

## Beispiel: Quicksort

```
1  def qsort(liste):
2      if liste == []: # Abbruchbedingung
3          return []
4      else:
5          pivot = liste[0] # Pivotelement wählen
6          # Teillisten sortieren
7          # 1. Element wird ausgespart, da es das Pivotelement
8          # ist
9          kleiner = qsort( [x for x in liste[1:] if x < pivot] )
10         größer = qsort( [x for x in liste[1:] if x >= pivot] )
11
12         # Ergebnisliste zusammensetzen
13         return kleiner + [pivot] + größer
```

# Analyse

- Laufzeit im schlechtesten Fall ebenfalls quadratisch zur Länge der Liste
- im Durchschnitt aber  $n \cdot \log n$  ( $n$ : Länge der Liste)!
- gute Wahl des Pivotelements ist nicht ganz einfach
- Rekursion (Selbstaufruf): elegant, birgt aber Risiken

# Implementierung

## Iteration vs. Rekursion

- Binärsuche sowohl iterativ als auch rekursiv möglich
- Sortieren (Quicksort) ebenfalls
- beide haben eine Abbruchbedingung
- beide Ansätze sind im Wesentlichen „gleich mächtig“
- Rekursion ist aber irgendwie „schöner“

## Rekursion im Alltag

Phänomen der Selbstähnlichkeit: Blumenkohl (Züchtung Romanesco), Farne, Schneeflocken, chemische Reaktionen, Kristallwachstum, moderne Antennen (Mobiltelefon)



Abbildung 2: Blumenkohlsorte Romanesco

# Rekursion im Alltag



Abbildung 3: Barnsley-Farn

# Rekursion im Alltag

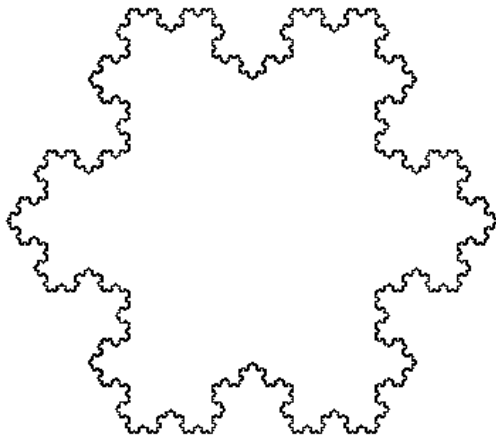
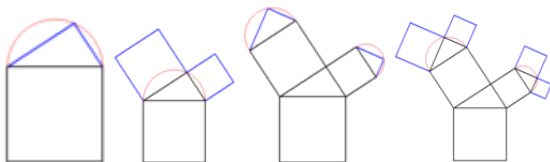


Abbildung 4: Koch-Flocke

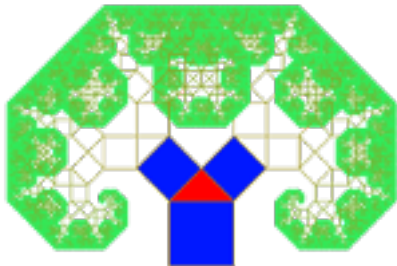
## Beispiel: Pythagoras-Baum

- Grundlinie  $\rightarrow$  Quadrat erzeugen
- Oberseite: Thaleskreis erstellen und beliebig teilen
- mit Teilungspunkt ein rechtwinkliges Dreieck erstellen
- Schenkel des Dreiecks sind zwei neue Grundlinien  
 $\rightarrow$  rekursiver Ansatz



# Beispiel: Pythagoras-Baum

Varianten „klassisch“ und „natürlich“



## Beispiel: Berechnung der Fakultät

rekursiv:

```
1 def fakultät_rekursiv(n):
2     if n <= 1: # Abbruchbedingung
3         return 1
4     else:
5         return ( n * fakultät_rekursiv(n-1) )
```

iterativ:

```
1 def fakultät_iterativ(n):
2     fakultät = 1; faktor = 2
3     while faktor <= n:
4         fakultät = fakultät * faktor
5         faktor = faktor + 1
6     return fakultät
```

## Binäre Suche: Rekursive Implementierung

```
1 def bsuche(daten, gesucht, start, ende):
2     mitte = (start + ende) // 2
3
4     if daten[mitte] == gesucht: # Abbruchbedingung
5         return mitte
6     elif daten[mitte] < gesucht:
7         # suche weiter in rechter Teilliste
8         return bsuche(daten, gesucht, mitte+1, ende)
9     else:
10        # suche weiter in linker Teilliste
11        return bsuche(daten, gesucht, start, mitte-1)
12
13 bsuche(meine_daten, "G", 0, len(meine_daten))
```

# Iteration vs. Rekursion

## ■ Rekursion:

- oft elegantere und kleinere Lösungen als vergleichbares iteratives Verfahren
- Lösungen werden „automatisch“ zusammengesetzt
- Lösungen nicht immer einfach zu verstehen
- Beschränkung durch Rechnerressourcen

## ■ Iteration:

- keine Beschränkung
- kann immer als Alternative zur Rekursion implementiert werden
- z.T. schneller

## Rekursion: Ressourcenproblem

- das Betriebssystem speichert für jedes laufende Programm Verwaltungsdaten
- jeder Funktionsaufruf fügt Daten zu, return baut sie wieder ab
- Problem bei der Rekursion: return-Statements kommen erst am Schluss!  
→ Menge der Verwaltungsdaten wächst immer mehr an
- erzeugt einen sog. „Stack Overflow“-Fehler
- Lösung bei unbekannter Größe der Datenmenge: Übersetzen in eine iterative Lösung

Ende