# Komplexität

Simplifizierung:
- Arithmetische Operation braucht 1ZE (Zeiteinheit)
- Vergleich braucht 1ZE
- Zuweisung braucht 1ZE

## O-Notation

$O(x)$ beweisen
- Positive oder negative reele Zahl → Klasse passt
- Null → es gibt eine passendere Klasse
- $\infty$ / $-\infty$ → Klasse passt nicht

Beispiel 1.

$3x^2 + 2x \;;\; O(x^2)$

$$\frac{3x^2+2x}{x^2} = \frac{3x^2}{x^2} + \frac{2x}{x^2} = \frac{3}{x} + \frac{2}{x^2}$$

durch die Komplexität teilen

$\lim_{n\to\infty} \frac{3}{x} + \frac{2}{x^2} = 0 \in \mathbb{R}$ → Gibt passendere Klasse aber passt trotzdem

Beispiel 2:

$3x^2 + 2x \;;\; O(x^2)$

$$\frac{3x^2+2x}{x^2} = 3 + \frac{2}{x}$$

$\lim_{n\to\infty} 3 + \frac{2}{x} = 3 \in \mathbb{R}$ → Klasse ist korrekt

### Wichtige Ableitungen

| $f(x)$ | $f'(x)$ |
|---|---|
| $\log_a(x)$ | $\frac{1}{x \cdot \ln(a)}$ |
| $\ln(x)$ | $\frac{1}{x}$ |
| $a^x$ | $a^x \cdot \ln(a)$ |

## MASTERTHEOREM

Gibt die Komplexität von rekursiven Funktionen

$$R(n) = a \cdot R\left(\frac{n}{b}\right) + c(n)$$

- $a$: Anzahl Rekursionsaufrufe pro Durchgang
- $b$: wiederaufgerufener Teil
- $c(n)$: höchste Potenz $d$ von allen z.B. $n^2 + 16^1$ → $d = 1$

1) $a < b^d \rightarrow O(n^d)$
2) $a = b^d \rightarrow O(\log_b(n) \cdot n^d)$
3) $a > b^d \rightarrow O(n^{\log_b a})$

Rekurrenzrelation immer nur bei Funktionen mit Rekursion

Komplexitätsklassen sortiert

$O(1), O(\sqrt{n}), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n!), O(n^n), O(n^{2n})$

## LAUFZEITKOMPLEXITÄT ABLESEN

```
int fun(int n)
{
  int i,k;

  for (i=0; i<n; i++) {
    int j = i;
    while(j < i * i) {
      j = j + 1;
      if (j % 3 == 0) {
        for (k=0; k<j; k++) {
          printf("Hello World!");
        }
      }
    }
  }
  return 0;
}
```

$n^2 \cdot n$ $O(n^2)$ × $\frac{1}{3}$ $O(1)$ → $n^2 - n$ $O(n^2)$ × $n$ $O(n)$ → $O(n^3)$

## LANDAU

- $\Omega$ : mindestens so schnell ≥
- $\Theta$ : genau so schnell, anderer Faktor ≈
- $\sim$ : genau so schnell, selber (1) Faktor =
- $O$ : höchstens so schnell ≤

# SORTIEREN

**in-place**: in einem Array, nur swapping von Werten

**out-of-place**: zweites Array benötigt

**Stabil**: ursprüngliche Reihenfolge gleicher Werte bleibt gleich

**instabil**: ursprüngliche Reihenfolge gleicher Werte verändert sich

0. 14, 3, 9, 22 — Vergleiche: $\frac{(4-1)\cdot 4}{2} = 6$
1. 3, 14, 9, 22 — Vertauschungen: 3
2. 3, 9, 14, 22
3. 3, 9, 14, 22

**Selection Sort** $O(n^2)$ Rechts kleinstes Element mit Index tauschen

Anzahl Vergleiche: $\frac{(n-1)\cdot n}{2}$ Anzahl Vertauschungen: $n-1$

Index 0, 1, 2 ..., wird mit dem kleinsten Element im rechten Teil vertauscht

0. 14, 3, 9, 22 I — Vergleiche: 4
1. 3, 14, 9, 22 II — Vertauschungen: 2
2. 3, 9, 14, 22 I
3. 3, 9, 14, 22

**Insertion Sort** $O(n^2)$ Kleinere Elemente werden nach Links gebubbelt

Anzahl Vergleiche: je nachdem Anzahl Vertauschungen: je nachdem

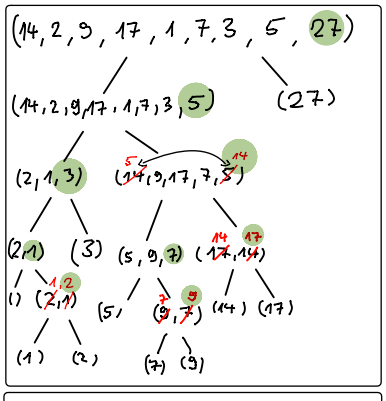
Vergleiche Index mit Element rechts, wenn rechtes Element kleiner, mit Index tauschen und so lange nach links vergleichen und tauschen bis links vom Element ein kleineres oder kein Element ist.

0. 14, 3, 9, 22 — Vergleiche: $\frac{(4-1)\cdot 4}{2} = 6$
1. 3, 9, 14, 22 — Vertauschungen: 2
2. 3, 9, 14, 22
3. 3, 9, 14, 22

**Bubble Sort** $O(n^2)$ Größtes Element an richtige Stelle

Anzahl Vergleiche: $\frac{(n-1)\cdot n}{2}$ Anzahl Vertauschungen: abhängig

Vertauscht mit nächstem Element, wenn dieses kleiner ist. Nach einem Durchgang ist letztes Element an korrekter Position.

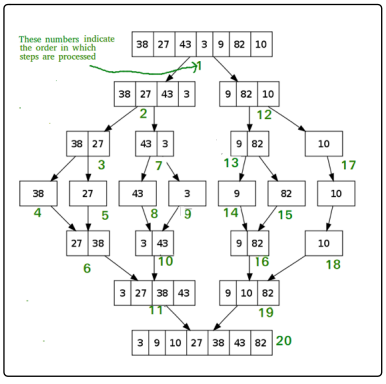



**Quicksort** (mit Lomuto) $O(n^2)$ Partitionieren an Pivot

Anzahl Vergleiche: je nachdem Anzahl Vertauschungen: je nachdem

1. Pivot Element wählen (idealerweise von mittlerer Größe)
2. Teile das Array in Teillisten
   - Eine Teilliste mit Elementen < dem Pivot
   - Eine Teilliste mit Elementen ≥ als Pivot. / Pivot wird an korrekte Stelle gesetzt
3. Wende Quicksort auf beide Teillisten an (Rekursiv)
4. Füge am Ende alle Teillisten zusammen.

Der Algorithmus wird beendet, wenn alle Teillisten die Länge 0 oder 1 haben oder wenn jedes Element in der Liste gleich.




**Mergesort** $O(n \log n)$ Teilen in der Mitte und gespiegelt wieder zusammenfügen

Anzahl Vergleiche: je nachdem Anzahl Vertauschungen: je nachdem

Array wird immer in der Mitte geteilt bis nur noch Arrays mit einzelnen Elementen übrig sind. Die Listen werden dann rekursiv zusammengefügt. Beim Zusammenfügen wird immer das erste von Liste 1 mit dem ersten von Liste 2 verglichen, das kleinere wird in die neue Liste geschrieben usw.

| | Selection Sort | Insertion Sort | Bubble Sort | Quicksort | Mergesort |
|---|---|---|---|---|---|
| in-place | ✓ | ✓ | ✓ | X | X |
| out-of-place | (✓) | ✓ | X | ✓ | ✓ |
| Stabilität | instabil | stabil | stabil | instabil | stabil |

# HASHING

„kleinhacken": Aus den Schlüsselwerten Kleinholz machen

Beispiel-Hashfunktion $h(x) = (x+3) \bmod 13$

$A = (10, 16, 5)$

| $x$ | $h(x)$ |
|---|---|
| 10 | 0 |
| 16 | 6 |
| 5 | 8 |

- **Linear Probing**: wenn Platz in Hash-Tabelle schon belegt, gehe ein Platz weiter.
- **Re-Hashing**: wenn Platz belegt, berechne neuen Platz mit Hilfe der Re-Hashing-Funktion $h(x) + m \cdot rh(x)$
  → Wertebereich muss kleiner $x \pmod x$ sein

Erstes unsortiertes Element von rechts in das sortierte Array einschieben. So lange nach links tauschen bis an richtiger Stelle.

# PARTITIONIERUNG

## LOMUTO

Ist ein Element kleiner als das Pivot, wird es mit dem ersten unsortierten Element getauscht.

Am Ende wird das Pivot an die erste Stelle vom rechten (größeren) Array getauscht.

## HOARE

- sucht von links größer als Pivot
- sucht von rechts kleiner als Pivot
- Wenn beide Indices Element gefunden haben, werden diese vertauscht
- Pivot wird in die Mitte eingefügt.

# BÄUME

Baum < Binärbaum < Heap < Max/Min-Heap
< binärer Suchbaum < AVL

## AVL Bäume

- Ungleichgewicht (Balance) höchstens $|n| = 1$
- Binärer Suchbaum

### Einfügen / Löschen

- normal wie beim binären Suchbaum
- Balance ggf. mit Hilfe von Rotationen anpassen

### Binärer Suchbaum

linker Kindknoten kleiner als Elternknoten
rechter Kindknoten größer als Elternknoten

### Wurzel extrahieren

- Tausche Wurzel mit letztem Element (ganz unten rechts)
- Bringe neue Wurzel an richtigen Ort (Bubble-down)

### Knoten löschen

1. kein Nachfolger: Knoten einfach löschen
2. Ein Nachfolger: Löschen und Nachfolger an Stelle des gelöschten Knoten
3. Zwei Nachfolger: Tausche Knoten mit größtem Knoten im linken Teilbaum
   Den zu löschenden Knoten löschen (Rekursion möglich)

### Balancierte Bäume

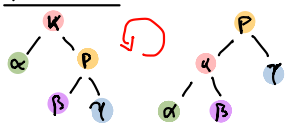
Größenbalancierter Baum: Teilbäume haben ähnlich viele Knoten
Höhenbalancierter Baum: Teilbäume haben ähnliche Höhe
Gewichtsbalancierter Baum: Wahrscheinlichkeit in beide Teilbäume zu gehen ist etwa gleich

**Höhe**
Anzahl der Knoten in der längsten Kette

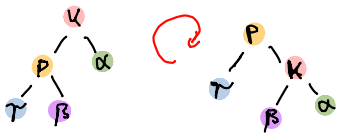
**Balance berechnen**
Höhe vom rechten Teilbaum − Höhe vom linken Teilbaum

### Rotationen (Binäre Suchbäume und AVL)

Linksrotation

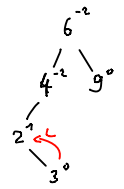



Rechtsrotation

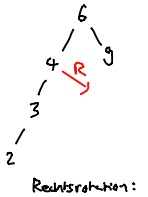



Linksrotation wenn $n = 2$
Rechtsrotation wenn $n = -2$
Doppelrotation wenn K anderes Vorzeichen als P
Bsp: $K = 2$, $P = -1$
1. Rechtsrotation mit P als Wurzel
2. Linksrotation mit K als Wurzel

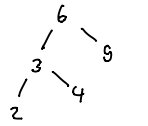
Beispiel:



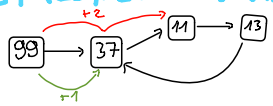

Linksrotation mit Teilbaum:




Rechtsrotation:




### ZYKLENERKENNUNG




Zwei Iteratoren gehen gehen die Liste durch. Wenn sie sich irgendwann auf selbem Feld treffen → Zyklus

### ALPHABET

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

### Beispiel: Matrix-Multiplikation

Eingabe zwei $n \times n$-Matrizen a, b
Ausgabe Matrizenprodukt ($n \times n$-Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```

n · (3n + 4n²)
= 3n² + 4n³
n · (1 + 3n + 4n²)
n · 4
⇒ 4n³ + 3n² + n + 1
$O(n^3)$

Anmerkung: range(n) läuft von 0 … (n − 1)

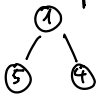
# HEAPS

- Jeder Knoten hat maximal zwei Nachfolger
- (fast) vollständiger Binärbaum
  - Alle Ebenen bis auf die unterste sind vollständig
  - Die unterste Ebene ist von links aus durchgehend besetzt (Es fehlen Knoten nur von rechts)

**Max-Heap**


jeder Nachfolger kleiner als Elternknoten

**Min-Heap**




Jeder Nachfolger größer als Elternteil

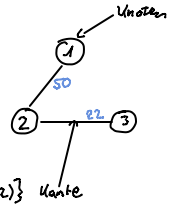
**Heapify:**
Alle inneren Knoten, angefangen beim letzten (Index) werden "bubble-downed"

MAX / MIN
**Bubble-up:**
Falls der Wert von Kind größer / kleiner als Eltern, dann tauschen.
So lang bis nicht mehr größer / kleiner

**Bubble-down:**
Falls der Wert von Eltern kleiner / größer als Kind, dann mit größtem / kleinsten Kind tauschen.
So lang bis nicht mehr größer / kleiner

**Heap-Sort:**
1. Max-Heap erstellen (heapify)
2. Wurzel mit letztem unsortierten Element tauschen
3. Vertauschte Wurzel als sortiert markieren
4. Bubble-down mit neuer Wurzel
5. Schritt 2 bis nur noch Wurzel unsortiert

# GRAPHEN




Knotenmenge V
$V = \{1, 2, 3\}$
Kantenmenge E
$E = \{(1,2), (2,1), (2,3), (3,2)\}$
$E \subseteq V \times V$

**gerichteter Graph**
- Besteht aus Knoten- und Kantenmenge

**Beispiele**
$G = (V, E)$
"Werte von V" $L_V = \{$Köln, Bremen…$\}$
"Werte von E" $L_E = \mathbb{N}$ (von Werten aus E)
"Zuweisung von V" $V = \{1 \mapsto$ Köln, $2 \mapsto \ldots\}$
"Zuweisung von E" $e = \{(1,2) \mapsto 50, \ldots\}$

**Begriffe**
Pfad = Folge von Knoten verbunden = jeder Knoten von überall erreichbar
Zyklus = Pfad, jede Kante einmal betreten (nicht leer)
Baum = verbunden, azyklisch

**ungerichteter Graph**
- ist ein gerichteter Graph
- bi-direktional:

$(a,b) \in E \wedge (b,a) \in E$

**Adjazenzmatrix**

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 |
| 3 | 0 | 1 | 0 |

Knoten (Zeile, Spalte)
Hauptdiagonale
wenn an dieser gespiegelt werden kann ⇒ ungerichtet

Gewichte in Tabelle
1 = Kantengewicht
0 = ∞

**Adjazenzliste**
1 ↦ (2)
2 ↦ (1,3)
3 ↦ (2)
Gewicht: 1 ↦ ((2,50))

Auf welchen Knoten er zeigt

**PRIM**
$O(|V| \cdot |E|)$
Die kleinste Verbindung des besuchten und unbesuchten Knotennetzes finden und Knoten zu besucht hinzufügen

**Dijkstra**
$O(|V|^2)$
Den kürzesten Weg von jedem Knoten zum Anfangs Knoten suchen und nach aufsteigender Größe sortiert.

Greedy-Algorithmen optimieren nur nächsten Schritt

# BEISPIELE

**Insertion**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4 | 12 | 24 | 3 | 6 | 19 | 1 | 7 | 5 |
| 1 | 4 | 12 | 24 | 3 | 6 | 19 | 1 | 7 | 5 |
| 2 | 4 | 12 | 24 | 3 | 6 | 19 | 1 | 7 | 5 |
| 3 | 3 | 4 | 12 | 24 | 6 | 19 | 1 | 7 | 5 |
| 4 | 3 | 4 | 6 | 12 | 24 | 19 | 1 | 7 | 5 |
| 5 | 3 | 4 | 6 | 12 | 19 | 24 | 1 | 7 | 5 |
| 6 | 1 | 3 | 4 | 6 | 12 | 19 | 24 | 7 | 5 |
| 7 | 1 | 3 | 4 | 5 | 6 | 7 | 12 | 19 | 24 |

$V_g$: 26
$V_t$: 6

**Selection**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 4 | 12 | 24 | 3 | 6 | 19 | 1 |
| 1 | 1 | 12 | 24 | 3 | 6 | 19 | 4 |
| 2 | 1 | 3 | 24 | 12 | 6 | 19 | 4 |
| 3 | 1 | 3 | 4 | 12 | 6 | 19 | 24 |
| 4 | 1 | 3 | 4 | 6 | 12 | 19 | 24 |
| 5 | 1 | 3 | 4 | 6 | 12 | 19 | 24 |

$V_g$: $\frac{42}{2} = 21$
$V_t$: 5

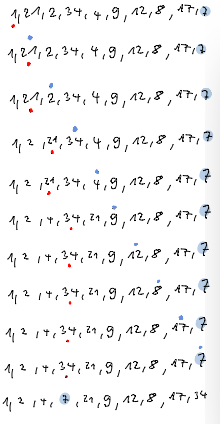
**Bubble-Sort**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4 | 12 | 24 | 3 | 6 | 19 | 1 | 7 | 5 |
| | 4 | 12 | 3 | 6 | 12 | 1 | 7 | 5 | 24 VI |
| | 4 | 3 | 6 | 12 | 1 | 7 | 5 | 19 | 24 V |
| | 3 | 4 | 6 | 7 | 5 | 12 | 19 | 24 | IV |
| | 3 | 4 | 1 | 6 | 5 | 7 | 12 | 19 | 24 III |
| | 3 | 1 | 4 | 5 | 6 | 7 | 12 | 19 | 24 II |
| | 1 | 3 | 4 | 5 | 6 | 7 | 12 | 19 | 24 |
| | 1 | 3 | 4 | 5 | 6 | 7 | 12 | 19 | 24 |

$V_t$: 19

**LOMUTO**

• Index • Marker



**Beispiel-Code um längste Reihe an roten Karten zu zählen.**

```
int find_max_red_series(int sample[], int n)
{
    int maxred = 0;
    int countred = 0;
    int i = 0;

    for (i = 0; i < n; i++)
    {
        if (sample[i] == 0)
            countred = 0;
        else
        {
            countred++;
            if (countred > maxred)
                maxred = countred;
        }
    }

    return maxred;
}
```