

# Grundlagen der Programmierung

*Vorlesung und Übung*

## 03 – Einfache Algorithmen

Prof. Dr. Andreas Biesdorf

Wirtschaft  
Hauptcampus

H O C H  
S C H U L E  
T R I E R

# WIEDERHOLUNG - SCHLEIFEN

## Kubikwurzel – Guess-and-Check Ansatz

```
x = int(input('Bitte einen ganzzahligen Wert eingeben: '))
ans = 0
while ans**3 < abs(x):
    ans = ans + 1
if ans**3 != abs(x):
    print(str(x) + ' hat keine ganzzahlige Kubikwurzel!')
else:
    if x < 0:
        ans = -ans
print('Kubikwurzel von ' + str(x) + ' ist ' + str(ans))
```

### Wiederholung Begriffe

- Guess-and-Check
- Exhaustionsmethode
- Iteration
- Lineare Suche

## Grundlagen

- Buchstaben, Zeichen, Sonderzeichen, Leerzeichen
- **Syntax:** Einfache oder doppelte Hochkommata
- **Konkatenierung** von Strings durch +
- Vergleich mit ==, <, > möglich
- **len()** ist eine Funktion, die die Länge des Strings ausgibt
- Strings können **indexiert** werden, Achtung: das erste Zeichen hat Index 0!

```
len("HST")      → 3
"HST"[0]        → 'H'
"HST"[1]        → 'S'
"HST"[3]        → IndexError
```

## Zerlegungen von Strings

- Zerlegung eines Strings möglich: [start:stop:schrittweite]

```
"hstmeanstuasmeanshst"[::-1]      → 'tshsnaemsautsnaemtsh'  
"hstmeanstuasmeanshst"[4:12:2]    → 'enta'  
"hstmeanstuasmeanshst"[-1]        → 't'
```

- Strings sind nicht veränderlich („immutable“)

```
s ="hstmeanstuasmeanshst"          → OK  
s[0]='H'                           → TypeError  
s = 'H' + s[1:len(s)]             → OK, da neues Objekt erzeugt wird
```

# WIEDERHOLUNG FOR-SCHLEIFEN

for

```
for <variable> in range(...):  
    <expression>  
    <expression>  
    ...
```

## Achtung:

Eine For-Schleife muss nicht zwingend über Zahlwerte iterieren!

- Für jeden Durchlauf der Schleife, `<variable>` nimmt iterativ die Werte in `range` an

range

```
range(5)  
range(2,5)  
range(3,10,2)  
range(5)
```





# WIEDERHOLUNG: STRINGS IN SCHLEIFEN

Wirtschaft  
Hauptcampus

H O C H  
S C H U L E  
T R I E R

## Option 1

```
shoutOut="HST"  
for position in range(len(shoutOut)):  
    print("Gib mir ein: " + shoutOut[position] + "!")
```

## Option 2 (eleganter)

```
shoutOut="HST"  
for zeichen in shoutOut:  
    print("Gib mir ein: " + zeichen + "!")
```

# WIEDERHOLUNG IM BEISPIEL: WHILE / FOR / STRING / IN

## Option 1

```
# Englisches Beispiel
an_Zeichen= "aefhilmnorsxAEFHILMNORSX"

wort = input("Gib mir ein Wort: ")
anzahlWiederholungen = int(input("Wie begeistert seid ihr? (1-10): "))

i = 0

while i < len(wort):
    zeichen = wort[i]
    if zeichen in an_Zeichen:
        print("Give me an " + zeichen + "! " + zeichen)
    else:
        print("Give me a " + zeichen + "! " + zeichen)
    i += 1

for i in range(anzahlWiederholungen):
    print(wort, "!!!!")
```

### Erläuterung:

Unterscheidung im Englischen zwischen „a“ und „an“

# ÜBUNG 1 – 1/3

```
iteration = 0
count = 0
while iteration < 5:
    for letter in "hello, world":
        count += 1
    print("Iteration " + str(iteration) + "; count is: " + str(count))
    iteration += 1
```

- 1] Was ist der Wert von count in Iteration 0 der While-Schleife?
- 2] Was ist der Wert von count in Iteration 1 der While-Schleife?
- 3] Was ist der Wert von count in Iteration 2 der While-Schleife?
- 4] Was ist der Wert von count in Iteration 3 der While-Schleife?
- 5] Was ist der Wert von count in Iteration 4 der While-Schleife?

# ÜBUNG 1 – 2/3

```
iteration = 0
while iteration < 5:
    count = 0
    for letter in "hello, world":
        count += 1
    print("Iteration " + str(iteration) + "; count is: " + str(count))
    iteration += 1
```

- 1] Was ist der Wert von count in Iteration 0 der While-Schleife?
- 2] Was ist der Wert von count in Iteration 1 der While-Schleife?
- 3] Was ist der Wert von count in Iteration 2 der While-Schleife?
- 4] Was ist der Wert von count in Iteration 3 der While-Schleife?
- 5] Was ist der Wert von count in Iteration 4 der While-Schleife?

# ÜBUNG 1 – 3/3

```
iteration = 0
while iteration < 5:
    count = 0
    for letter in "hello, world":
        count += 1
        if iteration % 2 == 0:
            break
    print("Iteration " + str(iteration) + "; count is: " + str(count))
    iteration += 1
```

- 1) Wie häufig wird das `print`-Statement ausgeführt?
- 2) Was ist der höchste Wert der Variable `iteration`, der ausgegeben wird?
- 3) Was ist der höchste Wert der Variable `count`, der ausgegeben wird?
- 4) Was ist der kleinste Wert der Variable `count`, der ausgegeben wird?

## Beispiele / Erklärungen

- Nachteil der Linearen Suche: kann sehr rechenintensiv werden – manche Probleme können so nicht gelöst werden (siehe Schach-Beispiel aus Vorlesung 1)

## Problem: Ziehen der Kubikwurzel

- Suche nach der Kubikwurzel aus einer beliebigen nicht-negativen Zahl
- Ziel: möglichst nahen Wert an der Kubikwurzel zurückgeben
- Erster Ansatz: Lineare Suche
  - Möglichst kleine Schritte generieren
  - Nach jedem Schritt prüfen, ob nah genug am korrekten Ergebnis

## Problem: Ziehen der Kubikwurzel

- Aufgabe: Ziehe die Kubikwurzel aus Zahl
- Randbedingung: Zulässiger Fehler durch Wert epsilon gegeben
- Idee:
  - Start mit einem **Schätzwert**
  - In möglichst **keinen Schritten** verändern
  - **Prüfung:**  $|schaetzwert^3| - Zahl \leq \text{epsilon}$
  - Veränderung Schätzwert durch Schrittweite delta

Veränderung von delta  
führt zu schlechter  
**Performance**

Veränderung von epsilon  
führt zu geringerer  
**Genauigkeit** des Ergebnis

## MÖGLICHE LÖSUNG - KUBIKWURZELZIEHEN

```
zahl = int(input("Zahl eingeben:"))

epsilon = 0.01
schaetzung = 0.0
schrittweite = 0.0001
anzahlSchaetzungen= 0

while abs(schaetzung**3 - zahl) >= epsilon:
    schaetzung += schrittweite
    anzahlSchaetzungen += 1

print('anzahlSchaetzungen=', anzahlSchaetzungen)

if abs(schaetzung**3 - zahl) >= epsilon:
    print('Fehler bei der Ermittlung der Kubikwurzel', zahl)
else:
    print(schaetzung, 'ist die Näherung an die Kubikwurzel von', zahl)
```

### Übung:

Probieren Sie verschiedene Werte für epsilon, schaetzung und schrittweite aus!

Was passiert, wenn schrittweite = 0.1?

# MÖGLICHE LÖSUNG – KUBIKWURZELZIEHEN KORRIGIERT

```
zahl = int(input("Zahl eingeben:"))

epsilon = 0.01
schaetzung = 0.0
schrittweite = 0.0001
anzahlSchaetzungen= 0

while abs(schaetzung**3 - zahl) >= epsilon and schaetzung <= zahl:
    schaetzung += schrittweite
    anzahlSchaetzungen += 1

print('anzahlSchaetzungen=', anzahlSchaetzungen)

if abs(schaetzung**3 - zahl) >= epsilon:
    print('Fehler bei der Ermittlung der Kubikwurzel', zahl)
else:
    print(schaetzung, 'ist die Näherung an die Kubikwurzel von', zahl)
```

Wie viele Schritte braucht man typischerweise bis zur Lösung?



Wie könnte man das Problem effizienter lösen?

## ÜBUNG 2 – 1/3

1] Was ist die Ausgabe des folgenden Programms?

```
x = 25
epsilon = 0.01
step = 0.1
guess = 0.0
while guess <= x:
    if abs(guess**2 - x) < epsilon:
        break
    else:
        guess += step
if abs(guess**2 - x) >= epsilon:
    print('failed')
else:
    print('succeeded: ' + str(guess))
```

## 1] Was ist die Ausgabe des folgenden Programms?

```
x = 25
epsilon = 0.01
step = 0.1
guess = 0.0
while guess <= x:
    if abs(guess**2 - x) >= epsilon:
        guess += step
    if abs(guess**2 - x) >= epsilon:
        print('failed')
    else:
        print('succeeded: ' + str(guess))
```

## ÜBUNG 2 – 3/3

1] Was ist die Ausgabe des folgenden Programms?

```
x = 25
epsilon = 0.01
step = 0.1
guess = 0.0
while abs(guess**2-x) >= epsilon:
    if guess <= x:
        guess += step
    else:
        break

if abs(guess**2 - x) >= epsilon:
    print('failed')
else:
    print('succeeded: ' + str(guess))
```

2] Was ist die Ausgabe des Programms, wenn x=23?

**Frage: Wie können wir das finden der Schätzwerte verbessern?**

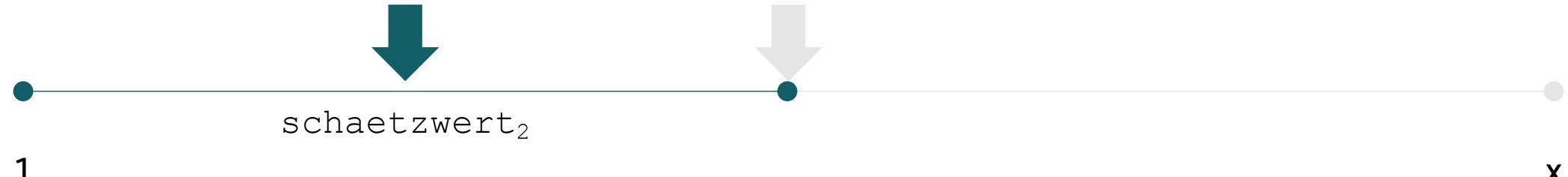
- “Educated guess”: Wir wissen, dass die Quadratwurzel einer ganzzahligen Zahl  $x$  immer zwischen 1 und  $x$  liegt
- Idee: nicht bei Null oder 1 starten, sondern in der Mitte zwischen 1 und  $x$

Initialer Schätzwert für das Näherungsverfahren

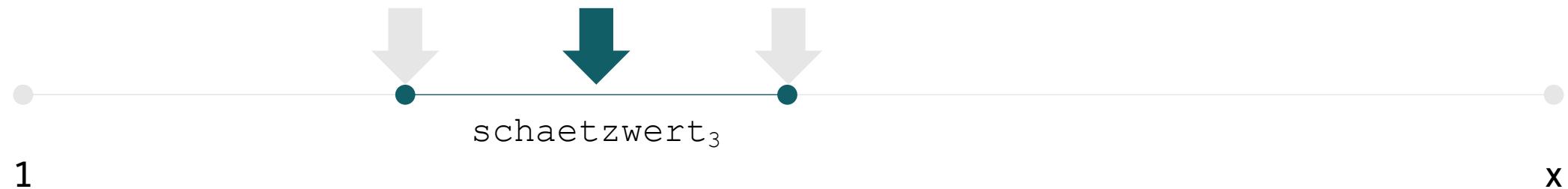


Frage: Wie können wir das finden der Schätzwerte verbessern?

- Falls der initiale Schätzwert nicht passt – prüfe ob zu groß oder zu klein
- Wenn  $schaetzwert_1^2 > wert$



- Wenn nun  $schaetzwert_2^2 < wert$



## Implementierung

```
zahl = int(input("Zahl eingeben:"))
epsilon = 0.01
anzSchritte= 0
unteresLimit = 1.0
oberesLimit = zahl
naeherung = unteresLimit + (oberesLimit - unteresLimit)/2.0

while abs(naeherung**2-zahl) >= epsilon:
    print('unteres Limit= ' + str(unteresLimit) + ' oberes Limit = ' + str(oberesLimit) +
          ' aktuelle Näherung= ' + str(naeherung))
    anzSchritte+= 1
    if naeherung**2< zahl:
        unteresLimit = naeherung
    else:
        oberesLimit = naeherung
    naeherung = unteresLimit + (oberesLimit - unteresLimit)/2.0

print('Anzahl Schritte= ' + str(anzSchritte))
print(str(naeherung) + ' ist unsere Näherung für die Quadratwurzel aus ' + str(zahl))
```

## Implementierung

```
zahl = int(input("Zahl eingeben:"))
epsilon = 0.01
anzSchritte= 0
unteresLimit = 1.0
oberesLimit = zahl
naeherung = unteresLimit + (oberesLimit - unteresLimit)/2.0

while abs(naeherung**3-zahl) >= epsilon:
    print('unteres Limit= ' + str(unteresLimit) + ' oberes Limit = ' + str(oberesLimit) +
          ' aktuelle Näherung= ' + str(naeherung))
    anzSchritte+= 1
    if naeherung**3< zahl:
        unteresLimit = naeherung
    else:
        oberesLimit = naeherung
    naeherung = unteresLimit + (oberesLimit - unteresLimit)/2.0

print('Anzahl Schritte= ' + str(anzSchritte))
print(str(naeherung) + ' ist unsere Näherung für die Kubikwurzel aus ' + str(zahl))
```

- **Suchraum wird iterativ halbiert**
  - Erste Näherung:  $N/2$
  - Zweite Näherung:  $N/4$
  - $n$ -te Näherung:  $N/2^n$
- Die **binäre Suche** konvergiert in der Größenordnung von  $\log_2 N$  Schritten  
Vergleich: die **lineare Suche** konvergiert in der Größenordnung von  $N$  Schritten!
- **Achtung:** Binäre Suche funktioniert nur, wenn die Werte sortiert vorliegen!

## ÜBUNG 3

- 1] Passen Sie den Code für das Ziehen der Quadrat- und Kubikwurzel so an, dass er auch für Werte <1 funktioniert!

### Hinweis:

- Wenn  $x < 1$  muss der Suchraum von 0 bis  $x$  gehen, da die Wurzel größer als  $x$  und kleiner als 1 ist

## Binärsuche

- Massive Reduktion der Rechenzeit durch kluge Einschränkungen des Suchraums
- Anwendbar auf alle Probleme, in denen die Objekte entsprechend eines Kriteriums **monoton aufsteigend (oder fallend) sortiert** sind:
  - In unserem Fall: sowohl  $g^2$  als auch  $g^3$  steigen monoton mit  $g$  für  $g > 0$

## Schreiben Sie ein Spiel nach folgenden Regeln:

- 1) Der Nutzer wird gebeten sich eine Zahl zwischen 0 und 100 auszudenken und zu merken
- 2) Das Programm versucht diese Zahl zu erraten
- 3) Bei jedem Vorschlag des Programms muss der Nutzer bewerten, ob der Vorschlag für die geheime Zahl
  - korrekt (k),
  - zu hoch (h) oder
  - zu niedrig (n) ist.
- 4) Auf Basis des Feedbacks wird ein neuer (möglichst kluger) Vorschlag erzeugt

### Mögliche Terminal-Ausgabe

Denke Dir eine Zahl zwischen 0 und 100 aus und merke sie Dir!

Ist die geheime Zahl 50?

Tippe k für korrekt, n für zu niedrig, h für zu hoch! - h

Ist die geheime Zahl 25?

Tippe k für korrekt, n für zu niedrig, h für zu hoch! - n

[... weitere Fragen und neue Schätzungen ...]

Ende. Deine geheime Zahl war 37

Oder: Warum gibt mir Python bei Divisionen teilweise seltsame Ergebnisse zurück?

- Werte in Float sind angenäherte reelle Zahlen mit einer durch den Computer limitierten Genauigkeit

**Zahlen im Dezimalsystem:**

$$42_{10} = 4 \cdot 10^1 + 2 \cdot 10^0$$

**Binäre Zahlen:**

$$\begin{aligned}101010_2 &= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\&= 32 + 0 + 8 + 2 + 0 + 0 = 42_{10}\end{aligned}$$

- Ein Computer muss auch reelle Zahlen intern durch binäre Zahlen annähern!

## Umrechnung der Integer 19 mit % und //

Idee:

- Modulo gibt uns das jeweils letzte Bit der Zahl
- Division durch 2 verschieben die Bits um eine Stelle nach links

shift	dez	binär	Ermittlung des jeweils letzten Bits					
	19	$10011_2$	$1 \cdot 2^4$	$+ 0 \cdot 2^3$	$+ 0 \cdot 2^2$	$+ 1 \cdot 2^1$	$+ [19 \% 2] \cdot 2^0$	
$19//2$	9	$1001_2$	$1 \cdot 2^3$	$+ 0 \cdot 2^2$	$+ 0 \cdot 2^1$	$+ [9 \% 2] \cdot 2^0$		
$9//2$	4	$100_2$	$1 \cdot 2^2$	$+ 0 \cdot 2^1$	$+ [4 \% 2] \cdot 2^0$			
$4//2$	2	$10_2$	$1 \cdot 2^1$	$+ [2 \% 2] \cdot 2^0$				
$2//2$	1	$1_2$	$[1 \% 2] \cdot 2^0$					

There are only 10 types of people in the world:  
Those who understand binary numbers  
and those who don't.

**Apologies for posting old jokes,  
but I still like it ;-)**

## Implementierung

```
intZahl = int(input("Zahl im Dezimalsystem eingeben:"))
zahl = intZahl
if zahl < 0:
    isNeg = True
    num = abs(zahl)
else:
    isNeg = False

result = ''

if zahl == 0:
    result = '0'
while zahl > 0:
    result = str(zahl%2) + result
    zahl = zahl//2

if isNeg:
    result = '-' + result

print (intZahl, "dargestellt als binäre Zahl ist", result)
```

## Umrechnung der Dezimalzahl 3/8 als binäre Zahl

**Dezimalzahl:**  $3/8_{10} = 0.375_{10} = 3 \cdot 10^{-1} + 7 \cdot 10^{-2} + 5 \cdot 10^{-3}$

### Idee für die Umrechnung:

- **Multiplikation** des Bruchs mit einer Zahl  $x = 2^y$ , so dass ein ganzzahliger Wert entsteht:

$$3/8_{10} \cdot 2^3_{10} = 3_{10}$$

- **Konversion** zu binär:

$$3_{10} = 11_2$$

- **Division** mit  $2^3_{10}$ : entspricht **Verschiebung nach rechts** um drei Stellen

$$11_2 / 2^3_{10} = 0.011_2$$

## Implementierung

```
x = float(input('Bitte Dezimalzahl zwischen 0 und 1 eingeben: '))
p = 0
while((2**p)*x)%1!= 0:
    print(str(p) + ' Rest= ' + str((2**p)*x-int((2**p)*x)))
    p+= 1

num = int(x*(2**p))

result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2

for i in range(p-len(result)):
    result = '0'+ result

result = result[0:-p] + '.' + result[-p:]
print('Die Binärdarstellung der Dezimalzahl ' + str(x) + ' lautet '+ str(result))
```

### Übung:

Probieren Sie verschiedene Eingabewerte aus!

0.375, 0.37, 0.25, 0.2, 0.1, 0.5

## Konsequenzen für Vergleiche von Float-Werten

- Wenn es keine Zahl  $y$  gibt, die  $x \cdot 2^y$  auf eine ganze Zahl abbildet, so ist die interne Darstellung immer nur eine Annäherung
- Empfehlung: Test auf Gleichheit durch

`abs (floatA - floatB) < epsilon`

anstatt von

`floatA == floatB`

## ÜBUNG 4

- 1) Wahr oder falsch? Jede interne Darstellung einer Float-Zahl im Computer ist immer eine Annäherung.
- 2) Wie wird die Zahl 11 binär dargestellt?
- 3) Wahr oder falsch? Die interne Darstellung der Zahl 0.1 benötigt unendlich viele binäre Ziffern.
- 4) In Ihrem Programm gibt es nach vielen Berechnungen zwei Floats: floatA und floatB.  
Sie vergleichen die beiden Werte mit `floatA = floatB`.  
Funktioniert Ihr Programm ...
  - a) *immer,*
  - b) *manchmal oder*
  - c) *nie?*

## Iterativer Approximationsalgorithmus zur numerischen Lösung nichtlinearer Gleichungen

- Allgemeine Form des Polynoms:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

- Mit Hilfe des Newtonverfahrens lassen sich **Näherungswerte zu den Nullstellen** von  $f(x)$  finden:

$$f(x) = 0$$

- Das Iterationsverfahren konvergiert im optimalen Fall asymptotisch
- Gegeben sei ein Startwert  $x_0$ , Berechnung von  $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

## Am Beispiel der Quadratwurzel

- Wir suchen die Wurzel aus einem Wert  $k$ :  $\sqrt{k} = x$
- Auflösen nach  $k$ :  $k = x^2$  und umstellen nach 0:  $0 = x^2 - k$

**Frage nun:** Was sind die Nullstellen von  $f(x) = x^2 - k$ ?

- Erste Ableitung:  $f'(x) = 2x$
- Initiale Schätzung:  $x_0$
- Iterative Bestimmung:


$$x_1 = x_0 - \frac{f(g)}{f'(g)} = x_0 - \frac{x_0^2 - k}{2x_0}$$
$$x_2 = x_1 - \frac{x_1^2 - k}{2x_1}$$
$$x_3 = x_2 - \frac{x_2^2 - k}{2x_2}$$
$$x_4 = \dots$$

## Implementierung

```
# Abbruchkriterium
epsilon = 0.01

# Suche Wurzel zu eingabewert
eingabewert = 24.0

# Grober Schatzwert
schaetzung = eingabewert/2

anzahlSchaetzungen = 0

while abs(schaetzung*schaetzung - eingabewert) >= epsilon:
    anzahlSchaetzungen += 1
    schaetzung = schaetzung - (((schaetzung**2) - eingabewert)/(2*schaetzung))

print('Anzahl Schätzungen= ' + str(anzahlSchaetzungen))
print('Quadratwurzel aus ' + str(eingabewert) + ' ist annäherungsweise ' + str(schaetzung))
```

**Übung:** wie sähe die Implementierung des Newton-Verfahrens für das Kubikwurzelziehen aus?

## Guess-and-Check Methoden

- **Standardisierte Ansätze**, um iterativ zu einer Lösung zu gelangen
  - Prinzip: Schleife, die die Schätzung optimiert
- Mögliche Verfahren, um **Schätzungen** zu generieren
  - **Lineare Suche** / Sequentielle Suche / Exhaustionsmethode
  - **Binäre Suche** / Divide-and-Conquer Verfahren
  - **Newton-Verfahren** / Newton-Raphson

## Vergleich der Implementierungen

- Bitte notieren Sie die Anzahl Iterationen der verschiedenen Implementierungen für die Werte in der Tabelle

Quadratwurzel	23	2.300	2.300.000
Lineare Suche (step = 0.001)	4795	31.622	n/a
Binäre Suche	13	21	37
Newton	4	8	13

Kubikwurzel	23	2.300	2.300.000
Lineare Suche (step = 0.001)	2840	13.200	n/a
Binäre Suche	13	25	41
Newton	6	14	26