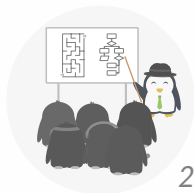


# Täglich grüßt das Murmeltier!

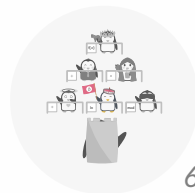
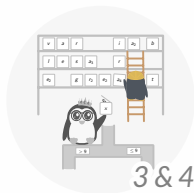
## *Tutorium sieben*



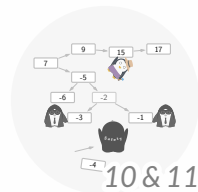
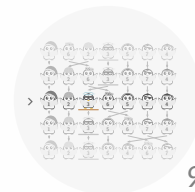
## Theorie



## Grundlagen



## Vertiefungen



- Eine Methode die sich selbst aufruft heißt rekursiv
  - Die Rekursion ist linear, wenn sie sich maximal einmal selbst aufruft
  - Solche sind „End-Rekursiv“, wenn der Aufruf das letzte ausgeführte Statement ist
  - Rekursion und Iteration sind dabei gleichmächtig
- Damit wir Bezeichner verwenden können müssen diese gültig und sichtbar sein
  - Die Gültigkeit hängt an der Deklaration (Überschattung, ...)
  - Die Sichtbarkeit hängt an Modifikatoren wie **public**, **private**, ...
- Methoden haben Seiteneffekte, wenn sie den Programmzustand über den Rückgabewert hinaus verändern
  - Seiteneffekte sind ein großes Problem und sollten erstmal vermieden werden
  - Ganz ohne geht es allerdings nicht (Konsolenausgaben, ...)

*I have learned more from my failures than can ever be revealed in the cold print of a scientific article. [...] [Failures] are much more fun to hear about afterwards; they are not so funny at the time.*  
— C. A. R. Hoare [Hoa21]

# Präsenzaufgabe

1

Irgendwo sind meine Socken doch

*In der Vorlesung wurde der Binary Algorithmus zur Suche in einer geordneten Struktur vorgestellt.*

- a) *Geben Sie an, wie das folgende Array vom Suchalgorithmus Binary Search nach dem Schlüssel „4“ durchsucht wird.*

-1	0	2	3	4	7	8	9
----	---	---	---	---	---	---	---

- b) *Begründen Sie kurz, weshalb der Binary Search Algorithmus eine worst-case Laufzeit von  $T(n)_{\max} \in \mathcal{O}(\log n)$  hat.*

# Ja wie suchen wir denn die 4?

$$\frac{0+7}{2} = 3.5 \rightarrow 3$$

$$3 < 4$$

-1	0	2	3	4	7	8	9
0	1	2	3	4	5	6	7

0. Initial ist  $\min = 0$  und  $\max = 7$ . Wir springen in die Mitte:  $\text{mid} = \lfloor (\min + \max)/2 \rfloor$ .  
Hierbei runden wir im Gleichheitsfall ab. Das ist an sich willkürlich, hier aber von uns festgesetzt.
1. Das betrachtete Element ist kleiner als der Schlüssel, wir springen in die rechte Hälfte.  
Das heißt wir verschieben  $\min = \text{mid} + 1$  und wiederholen die Berechnung von mid.

# Ja wie suchen wir denn die 4?

$$7 > 4$$

-1	0	2	3	4	7	8	9
0	1	2	3	4	5	6	7

$$\frac{0+7}{2} = 3.5 \rightarrow 3$$

$$\frac{4+7}{2} = 5.5 \rightarrow 5$$

0. Initial ist  $\min = 0$  und  $\max = 7$ . Wir springen in die Mitte:  $\text{mid} = \lfloor (\min + \max)/2 \rfloor$ .  
Hierbei runden wir im Gleichheitsfall ab. Das ist an sich willkürlich, hier aber von uns festgesetzt.
1. Das betrachtete Element ist kleiner als der Schlüssel, wir springen in die rechte Hälfte.  
Das heißt wir verschieben  $\min = \text{mid} + 1$  und wiederholen die Berechnung von mid.
2. Das betrachtete Element ist größer als der Schlüssel, wir springen in die linke Hälfte.  
Das heißt wir setzen  $\max = \text{mid} - 1$  und wiederholen die Berechnung von mid.

# Ja wie suchen wir denn die 4?

$$4 = 4$$

-1	0	2	3	4	7	8	9
0	1	2	3	4	5	6	7

$$\frac{0+7}{2} = 3.5 \rightarrow 3$$

$$\frac{4+7}{2} = 5.5 \rightarrow 5$$

0. Initial ist  $\min = 0$  und  $\max = 7$ . Wir springen in die Mitte:  $\text{mid} = \lfloor (\min + \max)/2 \rfloor$ .  
Hierbei runden wir im Gleichheitsfall ab. Das ist an sich willkürlich, hier aber von uns festgesetzt.
1. Das betrachtete Element ist kleiner als der Schlüssel, wir springen in die rechte Hälfte.  
Das heißt wir verschieben  $\min = \text{mid} + 1$  und wiederholen die Berechnung von mid.
2. Das betrachtete Element ist größer als der Schlüssel, wir springen in die linke Hälfte.  
Das heißt wir setzen  $\max = \text{mid} - 1$  und wiederholen die Berechnung von mid.
3. Das betrachtete Element entspricht dem gesuchten, liefere Index: 4 zurück.



# Und warum ist das schnell?

- › Mit jedem Vergleich sind wir entweder fertig, oder wir halbieren den Suchraum
- › Bei  $n$  Elementen können wir maximal  $\log_2(n)$  oft halbieren  
Ist  $n = 8$  so beispielsweise  $\log_2(8) = 3$  mal.
- › So benötigen wir maximal  $\log_2(n)$  viele Vergleichsschritte
- › Bei der  $\mathcal{O}$ -Notation ist die Basis des Logarithmus egal  
So ist für  $\log_a(x)$  der Unterschied zu  $\log_b(x)$  nur ein Faktor, der sich durch den Basiswechsel ergibt. Es ist:  
 $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$  dabei ist  $\log_b(a)$  eine Konstante die von der  $\mathcal{O}$ -Notation verschluckt wird.
- › Alle anderen Operationen benötigen konstante Zeit (sie sind nicht von  $n$  abhängig)
- › So kommen wir auf  $\mathcal{O}(\log n)$

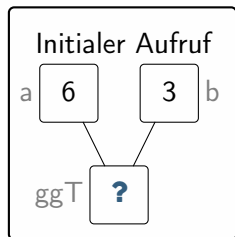
# Übungsblatt 7

# Aufgabe 1: Ablauf eines rekursiven Programms

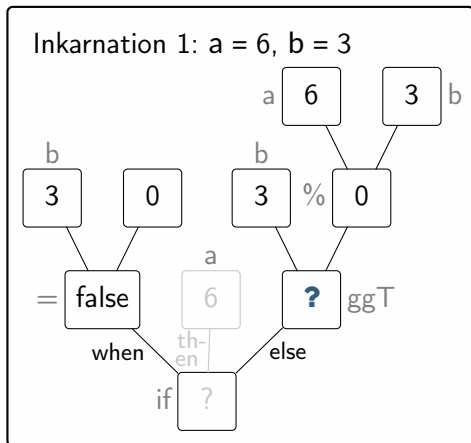
Für einen rekursiv definierten Algorithmus kann man ein „Berechnungsformular“ aufstellen, das für jeden rekursiven Aufruf (Inkarnation) vervielfältigt wird. Das Ergebnis der Berechnung wird dann durch Ein- und Rückübertragung von (Teil-) Lösungen bestimmt, wobei die Termination durch die Reduktion des Arguments ausgelöst wird. In dieser Aufgabe sollen Sie eine derartige Formularmaschine selbst angeben.

Betrachten Sie das folgende rekursive Programm und geben Sie das Formularblatt, sowie die Inkarnationen für den Aufruf `ggt(6, 3)` an. Folgen Sie dabei der Gestaltung und Ausführung des Beispiels aus der Vorlesung (Kap. 8, Folien 15ff.)

```
public int ggt(int a, int b) {  
    if(b == 0) {  
        return a;  
    } else {  
        return ggt(b, a % b);  
    }  
}
```

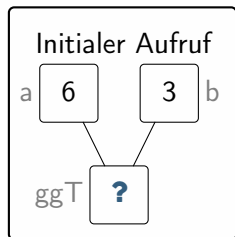


≡

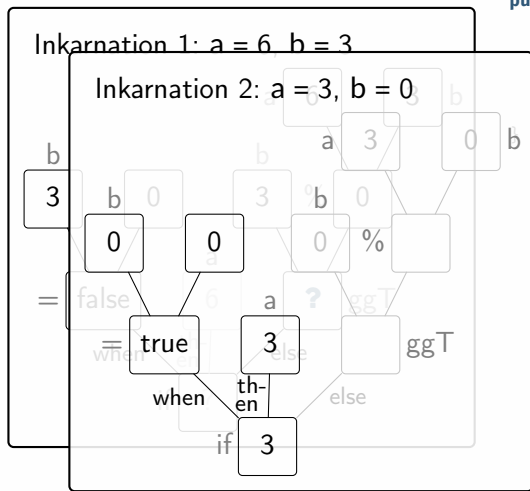


```
public int ggT(int a, int b) {  
    if(b == 0) {  
        return a;  
    } else {  
        return ggT(b, a % b);  
    }  
}
```

*Das Abschneiden von Ästen  
hier durch ausgrauen!*

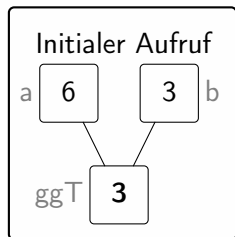


≡

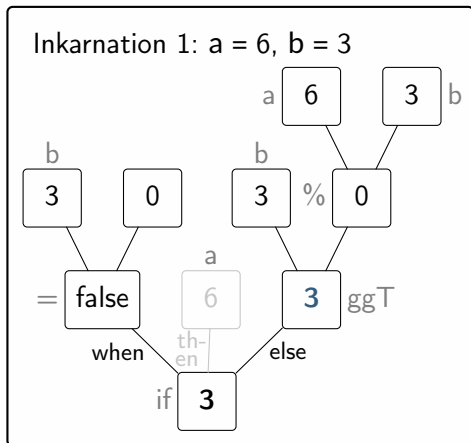


```
public int ggT(int a, int b) {  
    if(b == 0) {  
        return a;  
    } else {  
        return ggT(b, a % b);  
    }  
}
```

*Das Abschneiden von Ästen  
hier durch ausgrauen!*



≡



```
public int ggT(int a, int b) {  
    if(b == 0) {  
        return a;  
    } else {  
        return ggT(b, a % b);  
    }  
}
```

*Das Abschneiden von Ästen  
hier durch ausgrauen!*

## Aufgabe 2: Rekursive Algorithmen implementieren

Unter dem Integral  $R := \int_a^b f(x) dx$  einer Funktion  $f$  versteht man auch die Fläche zwischen der  $x$ -Achse und dem Graphen der Funktion. Im Folgenden sollen Sie eine rekursive Approximation  $A(a, b)$  für das Integral der Funktion  $f(x) = x^2$  im Intervall  $[a, b]$  implementieren. Dazu sollen Sie das Intervall solange in Teilintervalle halbieren, bis diese für eine direkte Berechnung hinreichend klein sind. Anschließend werden alle Teilflächen addiert. Es ergibt sich also folgende Rekursionsgleichung:

$$A^t(a, b) := \begin{cases} (b - a) \cdot f(a) & \text{falls } b - a \leq 0.01 \\ A^{t+1}(a, (a + b)/2) + A^{t+2}((a + b)/2, b), & \text{sonst} \end{cases}$$

Implementieren Sie eine Methode, die das Integral wie beschreiben rekursiv approximiert und das Ergebnis als Resultat zurückgibt. Testen Sie Ihre Implementierung für das Intervall  $[0, 1]$ .

# Mathematische Definitionen übernehmen

Rekursion.java

```
public class Rekursion {  
    public static double f(double x) {  
        return x * x;  
    }  
  
    public static double A(double a, double b) {  
        if(b - a <= 0.01)  
            return (b - a) * f(a);  
        else  
            return A(a, (a + b)/2) + A((a + b)/2, b);  
    }  
  
    public static void main(String[] args) {  
        System.out.println("∫[0,1] x^2 = " + A(0,1));  
    }  
}
```

$$f(x) = x^2$$

$$A^t(a, b) = \begin{cases} (b - a) \cdot f(a) & \text{falls } b - a \leq 0.01 \\ A^{t+1}(a, (a + b)/2) + A^{t+2}((a + b)/2, b), & \text{sonst} \end{cases}$$



# Aufgabe 3: Nicht-monotone Rekursion

Die bisher in der Vorlesung betrachteten rekursiven Algorithmen haben alle eine monoton in der Argumentgröße steigende Laufzeit. Es gibt aber rekursive Funktionen, bei denen diese Annahme nicht zutrifft. Ein Beispiel hierfür ist die Collatz Funktion, die für  $n \geq 1$  wie folgt definiert sei:

$$C(n) := \begin{cases} 1 & \text{falls } n = 1 \\ C(n/2) & \text{falls } n \text{ gerade} \\ C(3n + 1) & \text{sonst} \end{cases}$$

- Implementieren Sie die Collatz Funktion in Java, messen Sie jeweils die Ausführungszeit in Nanosekunden (s. `System.nanoTime()`) für die Eingaben von 1 bis 1000 und lassen Sie sich diese ausgeben.
- Was können Sie über den Zusammenhang von  $n$  und der Ausführungszeit sagen? Schreiben Sie Ihre Antwort als Kommentar in die Java Datei.

# Ein wenig mehr von Collatz

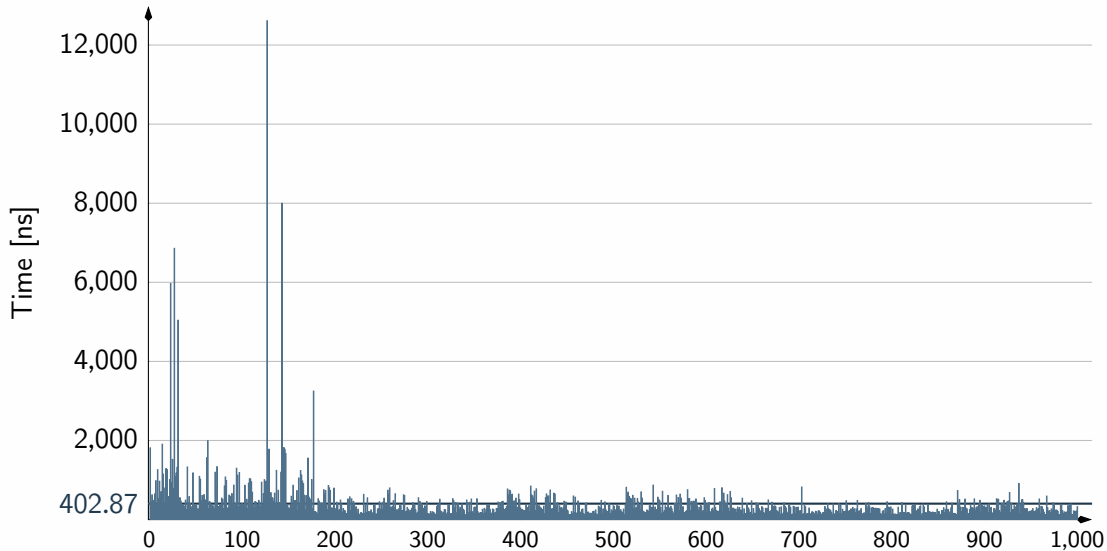
 Collatz.java

```
public class Collatz {  
    public static int C(int n) {  
        if (n == 1)  
            return 1;  
        else if (n % 2 == 0)  
            return C(n / 2);  
        else  
            return C(3 * n + 1);  
    }  
}
```

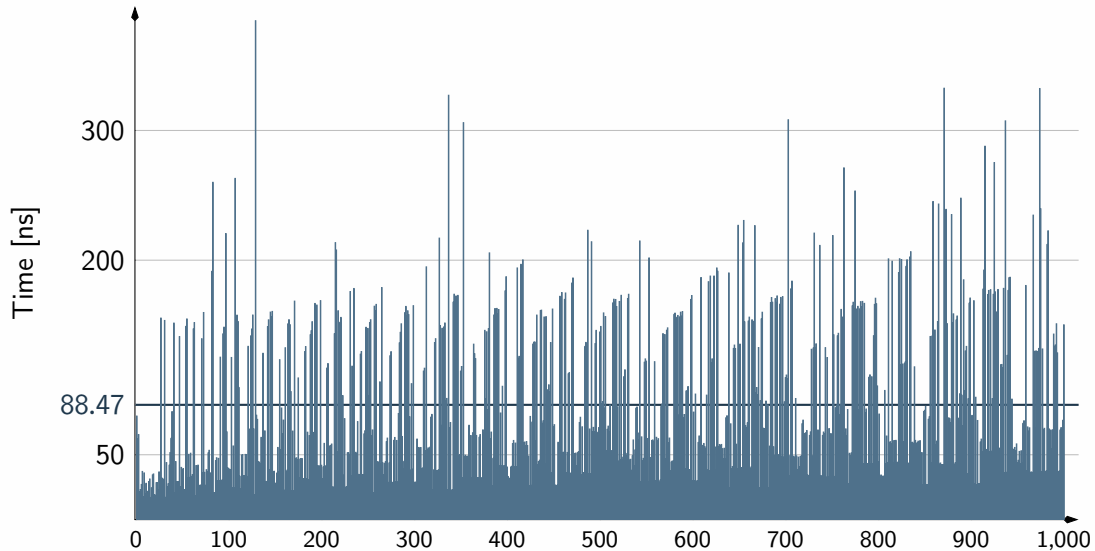
$$C(n) = \begin{cases} 1 & \text{falls } n = 1 \\ C(n/2) & \text{falls } n \text{ gerade} \\ C(3n + 1) & \text{sonst} \end{cases}$$

```
public static void main(String[] args) {  
    for (int i = 1; i <= 1_000; i++) {  
        long start = System.nanoTime();  
        C(i);  
        System.out.println("C(" + i + ") = "  
            + (System.nanoTime() - start));  
    }  
}
```

*Cherry-Picked, 1 Durchlauf (Teils bis 30 000 ns)*



*Gemittelt über 100 000 Läufe*



- › Benchmarking ist eigentlich um einiges tückischer! [RD12]
  - Das Zeitmessen selbst braucht Zeit
  - Die Java Virtual Machine hat zu Beginn einiges zusätzlich zu tun
  - Dienste wie die Garbage-Collection können unkontrolliert dazwischenfunken
- › Dafür gibt es Bibliotheken wie `jmh` oder `caliper`
- › Hier ging es uns aber auch nicht um die exakten Zeiten, sondern nur um seltsame Schwankungen
  - Für die hätte man aber auch einfach die Rekursionstiefe aufzeichnen können
  - Menno...

- › Zwischen  $n$  und der Ausführungszeit existiert kein offensichtlicher Zusammenhang
- › Ganz allgemein zeigt die Collatz-Funktion wie eine einfache Formulierung ein quasi unmöglich vorhersehbares Verhalten an den Tag legen kann [Van05], [AM98]
  - $C(870)$  benötigt beispielsweise 28 Schritte
  - $C(871)$  benötigt hingegen 178 Schritte
  - $C(872)$  benötigt wieder 116 Schritte
  - $C(876)$  benötigt nur 54 Schritte
  - $C(14039)$  bedarf ebenfalls nur 45 Schritte
  - $C(9780657630)$  bedarf hingegen 1132 Schritte
  - $C(9780657633)$  wieder nur 248 Schritte

# Aussicht: Übungsblatt 8

# Aufgabe 1: Naive Sortieralgorithmen

- › Bei manchen Problemen kann gezeigt werden, dass sie sich nicht besser lösen lassen. So zum Beispiel das Sortierproblem, welches immer mindestens in  $\mathcal{O}(n \log n)$  liegt (*average-case*). Man spricht auch von einer unteren Schranke.
- › Ein paar Rechenregeln können hilfreich sein, seien  $k, c$  Konstanten:

$$k = \mathcal{O}(1)$$

$$f(n) = \mathcal{O}(f(n))$$

$$c \cdot \mathcal{O}(f(n)) = \mathcal{O}(f(n))$$

$$\mathcal{O}(f(n)) + \mathcal{O}(f(n)) = \mathcal{O}(f(n))$$

$$\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$$

$$\mathcal{O}(\mathcal{O}(f(n))) = \mathcal{O}(f(n))$$

$$\text{Sowie: } \mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$$



# Die O-Notation

- Für die  $\mathcal{O}$ -Notation gilt  $T(n) \in \mathcal{O}(f(n))$ , wenn  $f(n)$  eine obere Schranke von  $T(n)$  ist, also:

$$T(n) \in \mathcal{O}(f(n)) \iff \exists n_0 \in \mathbb{N} \, c \in \mathbb{R}^+ \, \forall n \geq n_0 : T(n) \leq c \cdot f(n).$$

- Neben den mathematischen Gesetzen (Logarithmus, ...) reichen meist die folgenden:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n!)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$	$14 \cdot 2^n$	$n! \cdot 10^{-42}$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch	exponentiell	faktoriell

- Es existieren auch noch andere Notationen, die uns zunächst egal sind

## Aufgaben 2 und 3

- › Bei Best- und Worst-case Analysen gilt es, Eingaben zu finden die die beste und schlechteste Laufzeit liefern
- › So wie das Nachvollziehen von Algorithmen schwer ist, kann es schwer sein diese für einen Algorithmus zu finden (hier hilft Übung)
- › Für die Transformation von Iteration zu Rekursion hilft ein Blick auf die Präsenzaufgabe letzter Woche

*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*  
— C. A. R. Hoare [Hoa21]

# Abschließendes

- [AM98] Ștefan Andrei und Cristian Masalagiu. „About the Collatz conjecture“. 1998
- [Hoa21] Tony Hoare. „The 1980 ACM turing award lecture“. 2021
- [RD12] John Rose und J. Duke. *MicroBenchmarks*. 2012
- [Van05] Jean Paul Van Bendegem. „The Collatz conjecture. A case study in mathematical problem solving“. 2005
- Rekursion ist neben Iteration ein mächtiger Abstraktionsmechanismus
    - Ein gleichmächtiger, um genau zu sein
    - Es erlaubt beispielsweise die Umsetzung von Divide-And-Conquer-Verfahren
  - Neben der Formularmaschine gibt es viele andere Visualisierungen für Rekursion
  - Selbst einfach aussehende Algorithmen können sehr schwer nachvollziehbar sein!
  - Bei der Komplexitätsbetrachtung hilft die  $\mathcal{O}$ -Notation zum Vergleich
    - $T(n) \in \mathcal{O}(f(n)) \iff \exists n_0 \in \mathbb{N} \ c \in \mathbb{R}^+ \ \forall n \geq n_0 : T(n) \leq c \cdot f(n)$
    - „Der am stärksten wachsende Ausdruck“

