Induktion

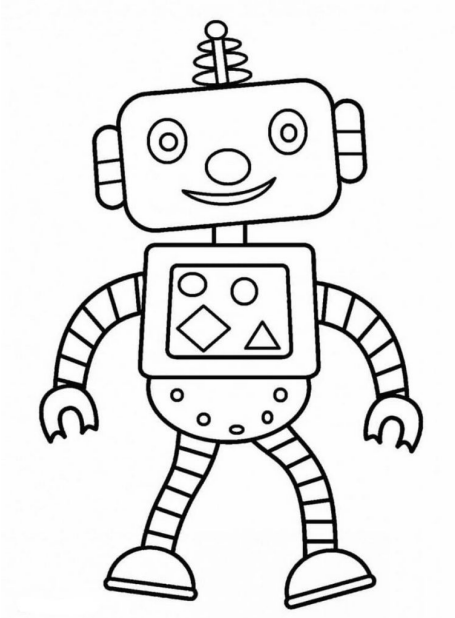
Die vollständige Induktion ist eine wirkungsvolle mathematische Beweismethode,welche auch als Entwurfstechnik für Algorithmen einen hohen Stellenwert hat. Wir schauen im Folgenden zwei Probleme an, welche auf den ersten Blick nicht zusammenhängen:

1 Wie lange dauert es, bis sich ein selbstreproduzierender Roboter eine Milliarde Mal geklont hat?

2 Wie viele Schritte sind notwendig, um eine Liste mit Zahlen zu sortieren?

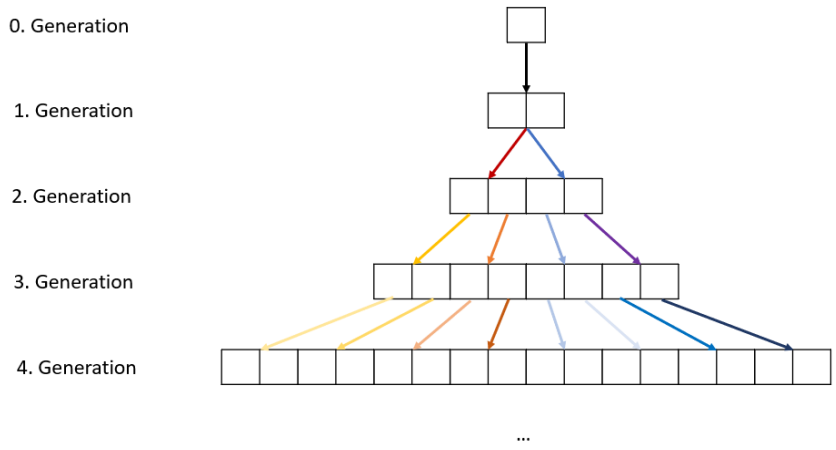
1 Vollständige Induktion

Rätsel 1: In einem unbekannten Land wohnt ein Roboter namens Logarithmus. Das erklärte Ziel von Logarithmus ist sich zu klonen. Er produziert jedes Jahr zwei identische Roboter und bricht dann vor Erschöpfung zusammen. Die neu produzierten Roboter produzieren ihrerseits wieder je zwei Roboter usw. Über wie viele Generationen müssen sich die Roboter klonen, damit eine Milliarde Roboter existieren?



Generation	0	1	2	3	...	h	1'000'000'000
Anzahl Roboter	1	2	4	8	...	?	?

Quelle: BD



1.1 Anzahl Roboter in der h-ten Generation

Wir befinden uns in der h-ten Generation. In der Folge soll bewiesen werden, dass in der h-ten Generation 2^h Roboter existieren. Für den Beweis verwenden wir eine Beweismethode mit dem Namen *vollständige Induktion*.

a) **Voraussetzung** Wir stellen eine Behauptung auf.

b) **Verankerung** Wir zeigen, dass die Behauptung für $h = 0$ korrekt ist, d.h. für die 0-te Generation.

c) **Schritt** Wir zeigen, dass aus der Behauptung für die h-te Generation die Behauptung für die Generation $h + 1$ folgt.

Daraus ergibt sich eine unendliche Kette von Beweisen, welche die Formel für beliebige h verifiziert. Aus der Korrektheit für $h = 0$ folgt $h = 1$. Aus der Korrektheit für $h = 1$ folgt $h = 2$. Aus der Korrektheit für $h = 2$ folgt $h = 3$ etc.

a)	Induktionsvoraussetzung	$N_h = 2^h$
b)	Induktionsverankerung	$N_0 = 2^0 = 1$
c)	Induktionsschritt	$N_{h+1} = 2^{h+1} = 2 \cdot 2^h = 2 \cdot N_h$

Da wir pro Generation eine Verdoppelung erwarten, ist damit der Induktionsschritt c) korrekt.

Quelle: BD

1.2 Gesamtzahl Roboter nach h Generationen

Wie viele Roboter wurden nach h Generationen gebaut, d.h., wie viele Roboter wären in Existenz, wenn die Roboter nicht nach einem Jahr kaputtgingen?

$$\begin{aligned} N_h &= 1 + 2 + 4 + 8 + 16 + \ldots + 2^h \\ &= 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^h \\ &= ? \end{aligned}$$

Beweise durch vollständige Induktion eine Formel, mit welcher du die gesuchte Grösse berechnen kannst.

a)	Voraussetzung	$N_h =$
b)	Verankerung	$N_0 =$
c)	Schritt	$N_{h+1} =$

1.3 Geometrische Summenformel

Die Summe

$$2^0 + 2^1 + 2^2 + \ldots + 2^h =$$

nennt man eine *geometrische Summe*. Du hast soeben durch vollständige Induktion die Summenformel für die geometrische Summe (Basis 2) bewiesen.

1.4 Anzahl Generationen bis zu einer Milliarde Roboter

Der Roboter *Logarithmus* hat über die Jahre ganz schön viele Nachkommen produziert. In welcher Generation wird die Grenze von einer Billion Roboter geknackt? Welchen Wert muss h annehmen, sodass gilt:

$$N_h \leq 1'000'000'000 \leq N_{h+1}$$

Wir schreiben eine Pythonfunktion! Die Funktion akzeptiert zwei Argumente: die Zahl, die erreicht werden soll und den Vervielfachungsfaktor pro Generation (hier 2), genannt *Basis*. Die Ausgabe soll zeigen, wie oft die Zahl durch 2 dividiert werden kann, ohne dass das Resultat unter 1 fällt. Wir nennen die Funktion *log* (kurz für Logarithmus).

```
def log(number, base):
    count = 0

    # dein code hier

    return count

print(log(1000000000000,2))
```

Öffne deinen Lieblingscodeeditor und ergänze die obigen Zeilen, sodass die Ausgabe stimmt. Als Kontrolle kannst du folgende Zahlen testen:

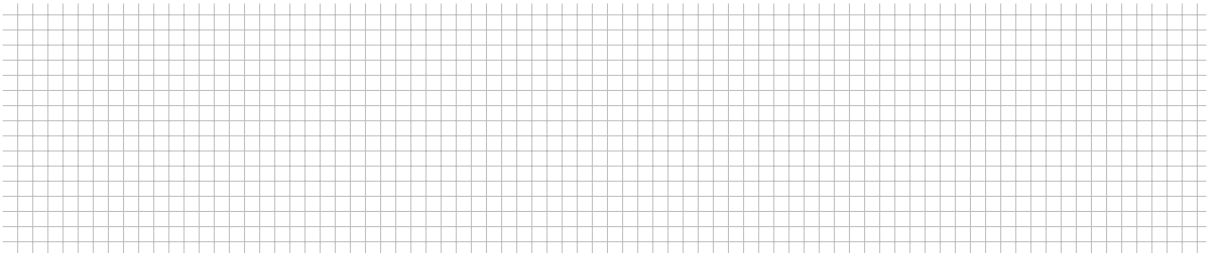
number	4	15	9	27	1'000'000'000
base	2	2	3	3	2
count	2	3	2	3	29

Also sind bereits in der 29. Generation über eine Milliarde Roboter vorhanden.

1.5 Anwendung des Logarithmus

Rätsel 2: Verwende die Funktion *Logarithmus*, um damit folgendes Problem zu lösen.

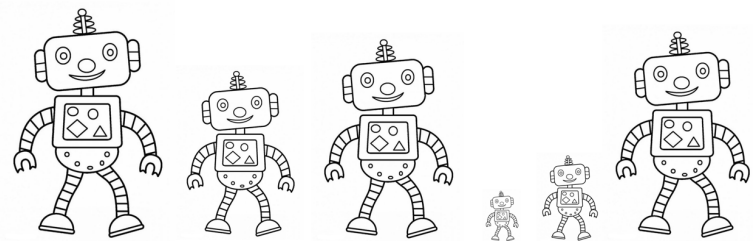
Bei jeder Faltung eines Blattes Papier wird dieses in der Dicke verdoppelt. Wie oft musst du ein Blatt Papier mindestens falten, damit der Stapel bis zum Mond reicht? Ein Blatt Papier hat die Dicke 0.1 mm und die Distanz Erde-Mond beträgt ungefähr 384'400 km.



Quelle: BD

2 Roboter sortieren

Der Roboter *Logarithmus* hat sich geklont. Die geklonten Roboter sind trotz aller Sorgfalt nicht zu 100% identisch. Die Roboter sollen der Grösse nach sortiert werden. Dazu betrachten wir eine Liste mit Körpergrössen.



2.1 Min-Sort Algorithmus

2.1.1 Algorithmus

Wir überlegen uns eine Variante, wie die Liste sortiert werden könnte. Betrachte folgenden Algorithmus.

```
def swap(the_list, first_index, second_index):
    temp = the_list[first_index]
    the_list[first_index] = the_list[second_index]
    the_list[second_index] = temp

def min_sort(the_list):
    result = the_list.copy()

    for index in range(len(result)):
        minimum = result[index]
        minimum_index = index

        for second_index in range(index, len(result)):
            if result[second_index] < minimum:
                minimum_index = second_index
                minimum = result[minimum_index]

        swap(result, index, minimum_index)
    return result

print(min_sort([9,3,2,5,3,4,1,9]))
```

Wie funktioniert das obige Programm? Beschreibe den Algorithmus in Worten.

Quelle: BD

2.1.2 Anzahl Vergleiche

Die Liste soll die Länge n haben.

1. Wir setzen die Variable *index* auf 0, d.h., zuerst wird das Minimum der Liste gesucht. Das Minimum wird beim Index 0 eingesetzt. Um das Minimum zu finden, müssen wir jedes Element einmal anschauen. Es werden also n Iterationen benötigt.
2. Anschliessend wird die Variable *index* auf 1 gesetzt. Wir suchen das Minimum der Liste (ohne das Element beim Index 0, welches ja schon gesetzt wurde). Es werden also noch $n - 1$ Iterationen benötigt.
3. usw. usf.

Insgesamt benötigt man also

$$n + (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = N$$

Iterationen.

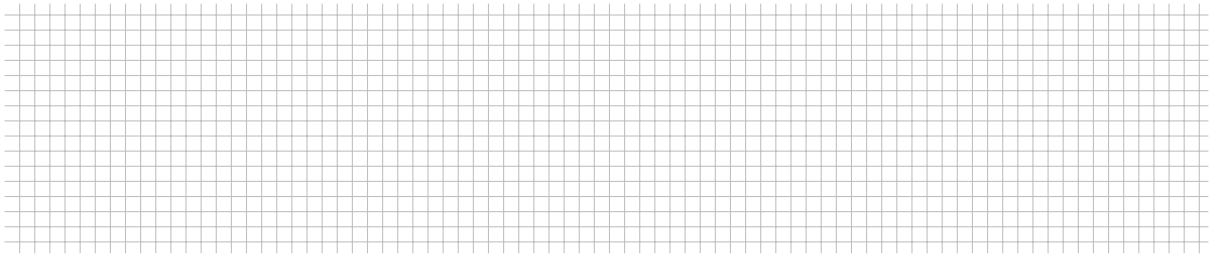
Durch vollständige Induktion zeigen wir, dass der Wert der obigen Summe berechnet werden kann mit:

$$N_n = \frac{n \cdot (n+1)}{2}$$

a)	Behauptung	$N_n = \frac{n \cdot (n+1)}{2}$
b)	Verankerung	$N_1 =$
c)	Schritt	$N_{n+1} =$

Quelle: BD

Wie viele Schritte werden für eine Liste der Länge 32 benötigt, wie viele für die Länge 1'000'000?



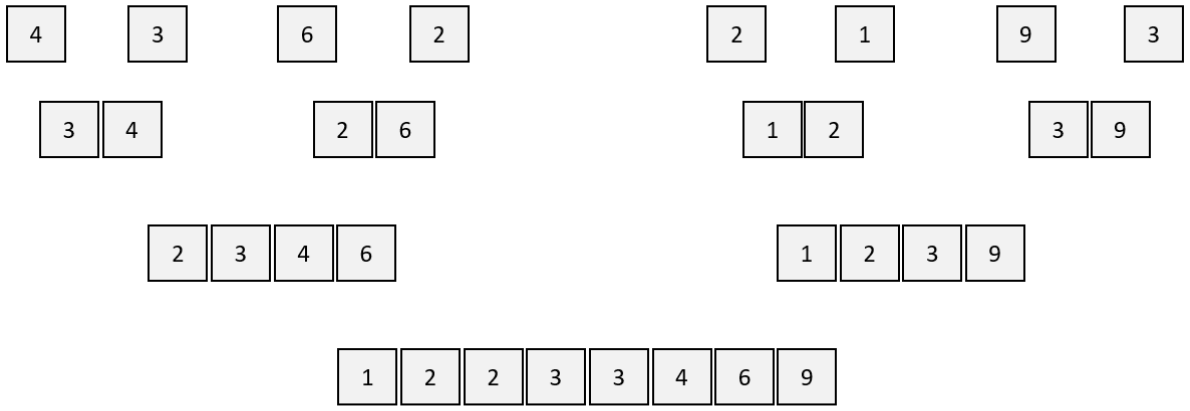
Quelle: BD

2.2 Merge-Sort: Ein induktiver Ansatz

Tatsächlich ist der Min-Sort Algorithmus von oben für grosse Listen kein schneller Algorithmus. Wir betrachten einen Algorithmus, welcher vom Kleinen zum Grossen arbeitet. Man spricht vom *Teile und Herrsche* Prinzip.

2.2.1 Induktiver Ansatz

Wir starten mit zwei Listen der Länge 1 und bauen daraus eine sortierte Liste der Länge 2. Aus zwei sortierten Listen der Länge 2 wird eine sortierte Liste der Länge 4 gebastelt usw.



Das Vorgehen erinnert an die Beweistechnik der Induktion:

- Als Induktionsvoraussetzung dient die Behauptung, dass jeder Schritt des Algorithmus eine sortierte Liste produziert.
- Als Induktionsverankerung dienen zwei Listen der Länge 1, welche bereits als sortiert gelten.
- Als Induktionsschritt werden die beiden Listen zu *einer* sortierten Liste zusammengeführt (engl. *merge*).

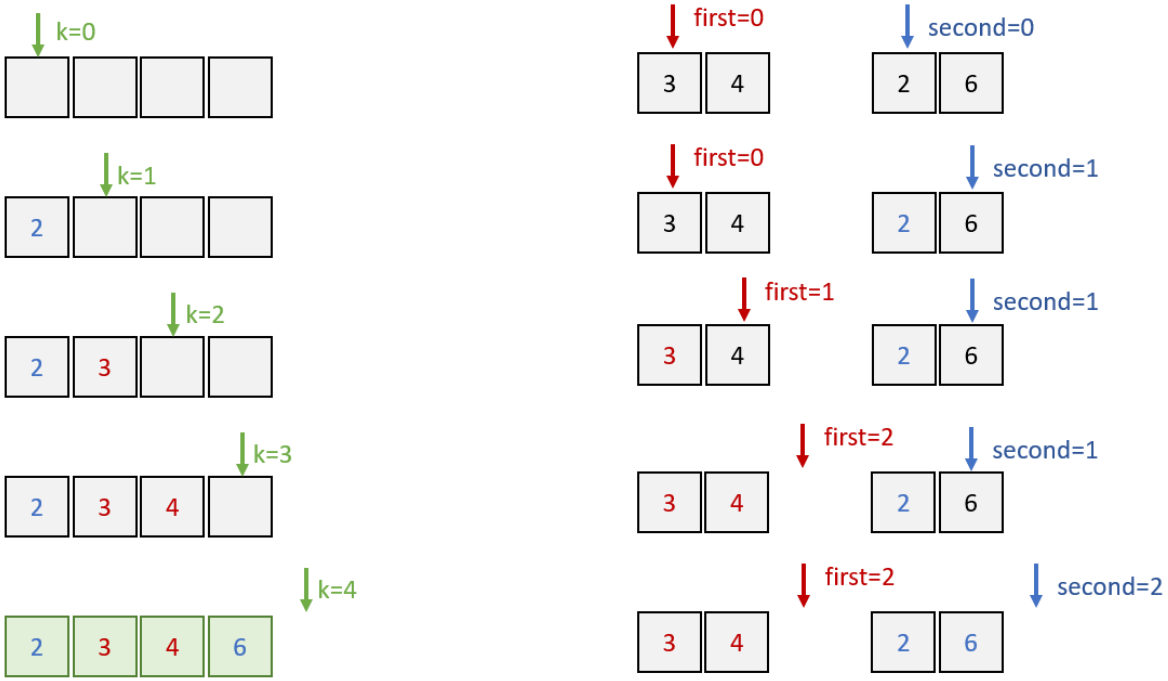
Solange also der Merge-Schritt eine sortierte Liste produziert, funktioniert der obige Algorithmus.

2.2.2 Merge-Schritt

Wir betrachten einen Algorithmus, welcher zwei sortierte Listen zu einer zusammenführt (*merge*). Als Beispiel schauen wir den Merge-Schritt zweier Listen der Länge 2 an: [3,4] wird mit [2,6] gemerged. Die resultierende Liste ist in der Abbildung grün eingefärbt, die erste Eingabeliste in rot, die zweite in blau.

1. Zu Beginn setzen wir den Index $k = 0$, den Index *first*=0 und den Index *second*=0. Die Zeiger auf der rechten Seite zeigen zu Beginn also auf die Zahlen 3 und 2.
2. Nun wird die kleinere der beiden Zahlen in die resultierende Liste eingesetzt.
3. Dieser Schritt wird so lange wiederholt, bis alle Zahlen eingesetzt wurden.

Quelle: BD




```
def merge(first_list, second_list):
    first_length = len(first_list)
    second_length = len(second_list)

    length = first_length + second_length
    result = [0]*length

    k = 0

    first = 0
    second = 0

    while k < length:
        if first == first_length:
            result[k] = second_list[second]
            second += 1
        elif second == second_length:
            result[k] = first_list[first]
            first += 1
        elif first_list[first] < second_list[second]:
            result[k] = first_list[first]
            first += 1
        else:
            result[k] = second_list[second]
            second += 1

        k += 1

    return result
```

2.2.3 Merge-Sort Algorithmus

Die Strategie ist also zusammenfassend wie folgt:

a) Wir spalten unsere Liste auf in Listen der Länge 1 [1,5,4,2] => [[1], [5], [4], [2]].

b) Für zwei aufeinanderfolgende Listen führe einen Merge-Schritt durch, d.h., füge zwei Teillisten zu einer grösseren, sortierten Liste zusammen [[1], [5], [4], [2]] => [[1,5], [2,4]].

Quelle: BD

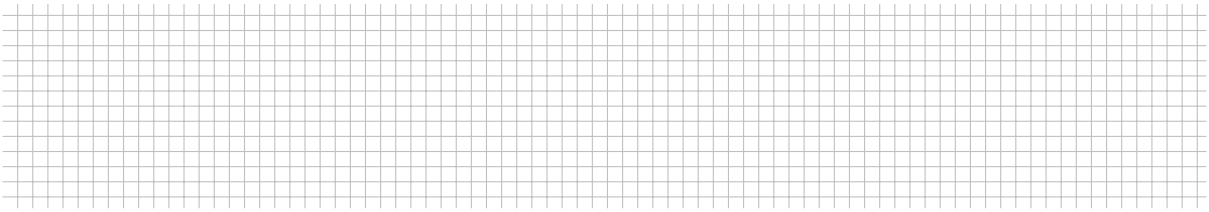
c) Wiederhole solange, bis alle Teillisten zu einer einzigen Liste verschmolzen sind. [[1,5], [2,4]] => [[1,2,4,5]]

```
def merge_sort(the_list):
    merge_list = [0]*len(the_list)

    for index in range(len(the_list)):
        merge_list[index] = [the_list[index]]

    while len(merge_list) > 1:
        temp_merge_list = []

        for index in range(0, len(merge_list), 2):
            if index == len(merge_list)-1:
                temp_merge_list.append(merge_list[index])
            else:
                temp_merge_list.append(merge(merge_list[index], merge_list[index
                                                                    +1]))
        merge_list = temp_merge_list

    return merge_list[0]
```

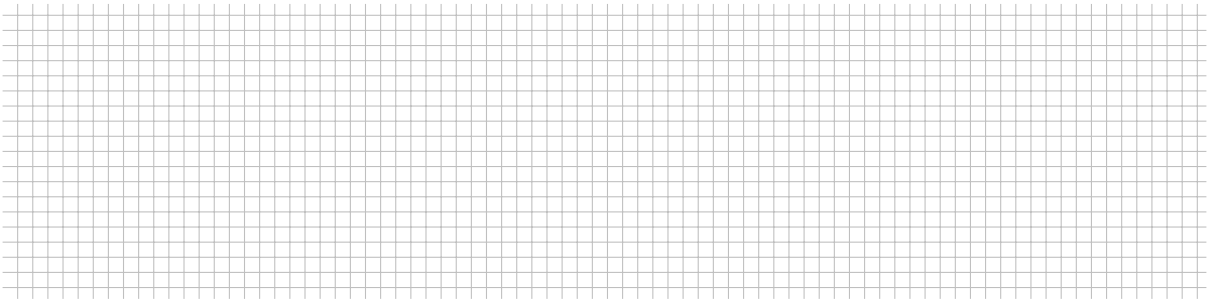
2.2.4 Anzahl Vergleiche

Inwiefern ist der Merge-Sort Algorithmus schneller als der Min-Sort Algorithmus? Die Geschwindigkeit des Algorithmus hängt wesentlich von der **Anzahl Vergleiche** ab: Wie oft wird eine Zahl mit einer anderen verglichen.

Überlege dir eine intuitive Begründung, weshalb beim Merge-Sort weniger Vergleiche notwendig sind als beim Min-Sort Algorithmus.



Überlege dir für den Merge-Sort Algorithmus, wie viele Vergleiche insgesamt notwendig sind. Nimm als Beispiel an, dass eine Liste der Länge 32 sortiert werden soll. Tipp: Um die Frage zu beantworten, benötigst du Wissen aus der Roboterfortpflanzung von *Logarithmus* (siehe oben).



2.3 Vergleich Min-Sort und Merge-Sort

Verwende das Pythonskript *sort_vorlage.py*.

a) Überprüfe mit einer Funktion *checkValidity(my_list)*, ob jeweils eine sortierte Liste vorliegt, d.h., ob die Algorithmen ein gültiges Resultat produzieren.

b) Überprüfe mit einer Funktion *compare(list_one, list_two)*, ob das Resultat von *min_sort* und *merge_sort* identisch ist.

c) Schreibe mit Hilfe der gezeigten Funktionen ein Programm, welches die Laufzeit der Sortieralgorithmen misst. Verwende dazu die Funktion *time.time()* aus dem Modul *time*, welches die Systemzeit in Sekunden zurückgibt.