



## Simon Knott

Developer Tooling Engineer



17 Sep 2017 • 11 mins read

# Rekursion

Rekursion ist eine Problemlösungsstrategie. Ihr besonderes Merkmal ist, dass sie komplexe Probleme aufteilt und sich dann für diese Unterprobleme selbst aufruft. Dabei besteht ein rekursiver Algorithmus immer aus zwei Teilen:

- **Rekursionsbasis:** Falls ein Problem so einfach ist, dass man es sofort lösen kann, wird das Ergebnis zurückgegeben. Bei einer rekursiven Zahlenfolge könnten das zum Beispiel die Grundwerte sein.
- **Rekursionsschritt:** Im Rekursionsschritt wird das Problem in beliebig viele Sub-Probleme unterteilt. Diese werden dann gelöst, zusammengesetzt und zurückgegeben.

In Pseudocode könnte es also so aussehen:

```
funktion rekursiv(n) {
    wenn einfache zu lösen
        dann gib lösung zurück
    sonst
        teile Problem in Unterprobleme auf
        führe dich selbst erneut aus
        gib zusammengesetzte Ergebnisse zurück
}
```

Hierbei sind Zeile 2-3 die *Rekursionsbasis*, Zeile 4-7 *Rekursionsschritt*.

## Fibonacci

Ein Standardbeispiel für die Rekursion ist die *Fibonacci-Sequenz*. Diese ist definiert wie folgt:

$$fib(n) = \begin{cases} fib(n - 1) + fib(n - 2) & n > 1 \\ 1 & n = 0 \vee n = 1 \end{cases}$$

Das bedeutet: Ein Element der Fibonacci-Sequenz ist immer die Summe seiner beiden Vorgänger, wobei die ersten beiden Stellen den Wert eins haben.

Fibonacci ist also Rekursion im Paradebeispiel: Ein Problem wird gelöst, indem die Funktion sich selbst aufruft.

Die ersten zehn Elemente der Fibonacci-Folge sind:

1; 1; 2; 3; 5; 8; 13; 21; 34; 55

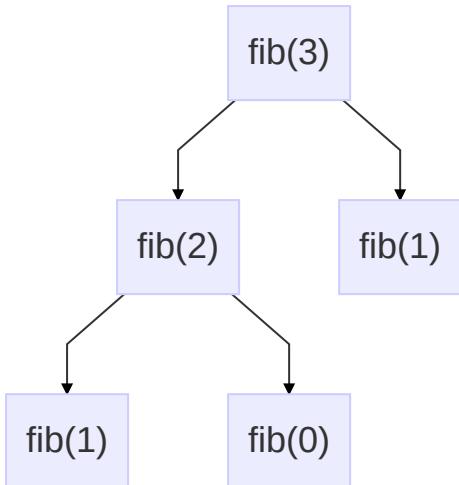
In der Java-Implementierung sieht das ganze so aus:

```
int fib(int n) {
    if (n == 0 || n == 1) return 1; // Base Cases

    return fib(n - 1) + fib(n - 2); // Rekursionsschritt
}
```

Zeichnet man nun ein Aufrufdiagramm, so wird ein Problem der Rekursion schnell deutlich: Der gleiche Wert wird mehrmals berechnet.

Am Aufrufdiagramm der dritten Stelle kann man das ganze schon sehen:



**fib(1)** wird zweimal ausgeführt. Nun macht dies beim ersten Wert keinen großen Unterschied, bei höheren Werten kann dies aber sehr schnell ins Gewicht fallen - schließlich steigt die anzahl der Aufrufe mit  $2^n$ . → Laufzeit:  $O(2^n)$

Als Lösung lässt sich Caching einsetzen: Wenn wir einen Wert berechnen, speichern wir ihn ab - Dann müssen wir beim nächsten Aufruf der Funktion nicht wieder die ganze Rechnung von vorne machen. Eine Implementierung in Java könnte so aussehen:

```

List<Integer> cache = new ArrayList<>(Arrays.asList(0, 1));
int fib(int n) {
    if (cache.size() <= n) cache.add(fib(n - 1) + fib(n - 2));

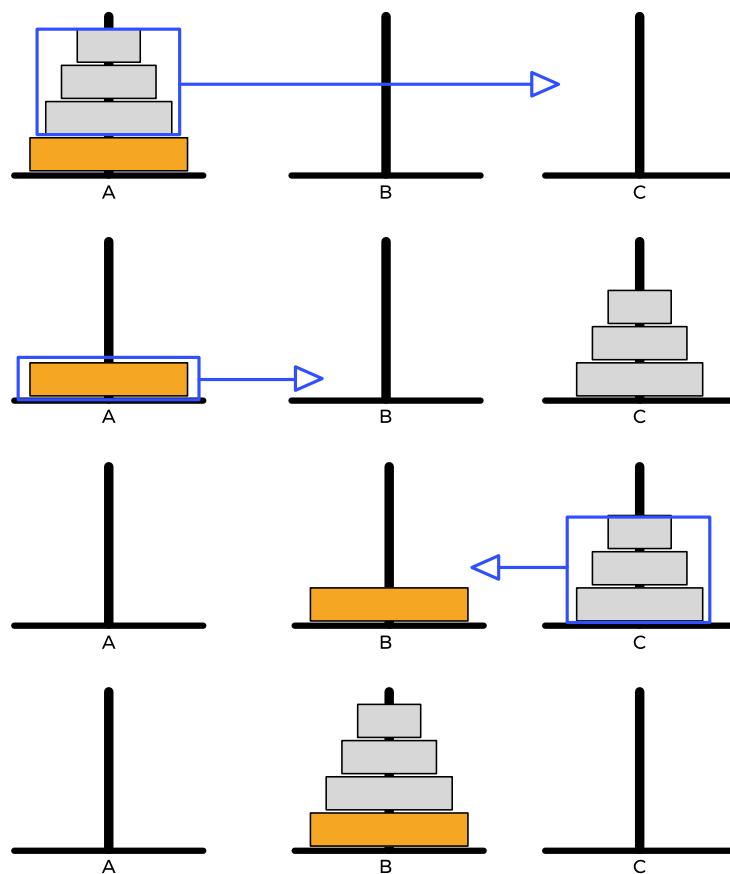
    return cache.get(n);
}
  
```

In Zeile 1 wird eine **ArrayList** initialisiert, die unsere Werte zwischenspeichert. Wird nun die Funktion **fib(n)** aufgerufen, prüft diese als erstes, ob der Wert schon darin enthalten ist. Dafür muss sie bloß die Länge der Liste mit dem angefragten Index vergleichen. Dies funktioniert, da an die Liste nur angehängt wird und dank der Rekursion alle Werte von klein nach groß eingetragen werden. Ist die Länge der Liste kleiner als der angefragte Index, dann ist der Wert noch nicht berechnet und er wird angehängt (Z. 4). Nun wird der Wert zurückgegeben (Z. 6) So wird jeder Wert also nur ein mal berechnet, die Laufzeit ist  $\Theta(n)$ .

# Hanoi

“Die Türme von Hanoi” ist ein Spiel des französischen Mathematikers Édouard Lucas. Darin existieren drei gleich große Stäbe A, B und C. Auf A steckt ein Stapel Scheiben, jede kleiner als die darunter. Ziel des Spiels ist es, alle Scheiben von A auf einen der beiden anderen Stäbe zu verschieben. Dabei ist die entscheidende Einschränkung, dass eine Scheibe nie auf einer kleineren aufliegen darf.

## Lösungsstrategie



Der Lösungsansatz lässt sich in folgende Schritte aufteilen:

1. Alle Scheiben bis auf die unterste von A nach B verschieben ( $Z(n - 1)$ )
2. Die übrige Scheibe auf C setzen (1)
3. Den Stapel auf B nach C verschieben ( $Z(n - 1)$ ) Die Anzahl der zu verschiebenden Scheiben ist in Klammern notiert,  $n$  bezeichnet die Anzahl der Scheiben im Anfangszustand.

Möchte man die Anzahl der minimal benötigten Schritte bestimmen, so lautet die Formel

$$Z(n) = 2 * Z(n - 1) + 1$$

$2 * Z(n - 1)$  stellt dabei Schritte eins und drei,  $+1$  Schritt zwei dar.

## Implementierung

### Rekursiv

Die Implementierung sieht wie folgt aus:

```
int hanoiRec(int n) {
    if (n == 1) return 1; // Base Case

    return 2 * hanoiRec(n - 1) + 1; // Rekursionsschritt
}
```

Laufzeit:  $\Theta(2^n)$

Die Rekursionsbasis ist  $n = 1$ : Wenn nur eine Scheibe verschoben werden soll, dann dauert das genau einen Zug. Der Rekursionsschritt implementiert exakt die oben genannte Funktion.

### Explizit

Die Anzahl der notwendigen Züge lässt sich auch explizit bestimmen:

$$Z(n) = 2^n - 1$$

Laufzeit:  $\Theta(n)$

## Lösen des Rätsels

Wir haben nun berechnet, in wie vielen Schritten man das Rätsel lösen kann - doch wie löse ich es nun wirklich? Dafür nutzen wir den oben erklärten Ansatz.

## Implementierung

```

/**
 * Solves the TowersOfHanoi problem to move the specified
 * number of disks from one pile to another.
 *
 * @param disks the number of disks to be moved
 * @param from the number of the pile where the disks are now
 * @param to the number of the pile to which the disks are to be moved
 * @param spare the number of the pile that can be used as a spare.
 */
private void solve(int disks, int from, int to, int spare) {
    /**
     * Die Methode moveOne(int fromStack, int toStack) bewegt eine Scheibe
     * mit der Nummer fromStack zum Stapel mit der Nummer toStack
     */
    // Rekursionsbasis: Nur eine Scheibe
    if (disks == 1) {
        moveOne(from, to);
        return;
    }

    // Bewege alle oberen auf den Spare-Haufen
    solve(disks - 1, from, spare, to);

    // Bewege die untere auf den Target-Haufen
    solve(1, from, to, spare);

    // Bewege die vom Spare-Haufen auf den Target-Haufen
    solve(disks - 1, spare, to, from);
}

```

## Hofstadter Q-Sequenz

Douglas R. Hofstadters Q-Sequenz ist eine Abwandlung der Fibonacci-Folge. In dieser wird durch die Vorgänger  $n - 1$  und  $n - 2$  bestimmt, aus welchen Vorgängern das Ergebnis zusammengesetzt wird.

Die Formel sieht so aus:



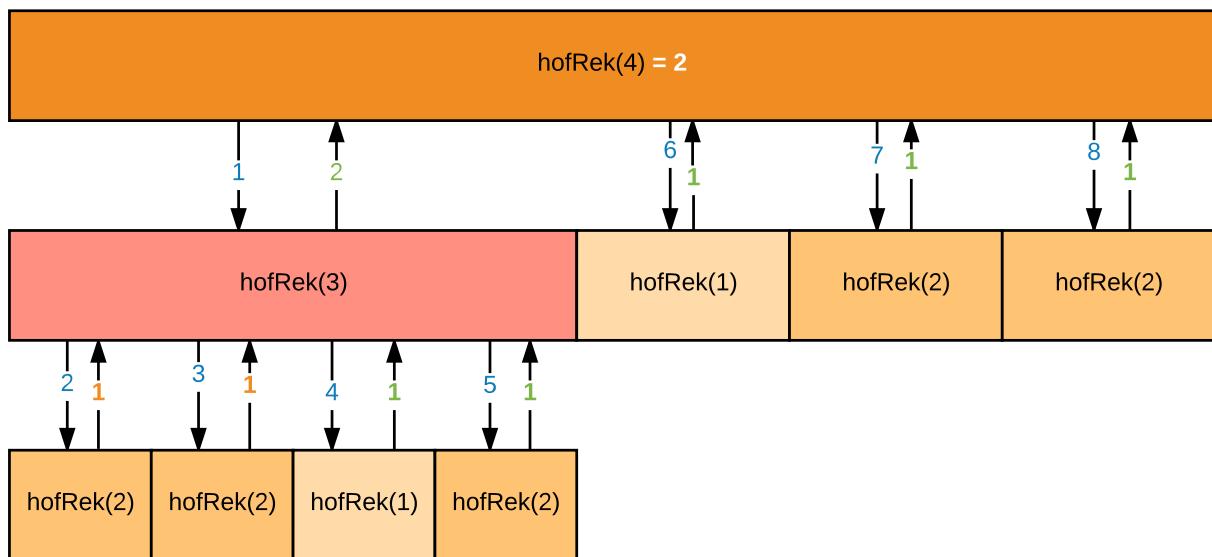
Die Implementation sieht so aus:

```
int hofstadterQRec(int n) {
    if (n <= 2) return 1;
    return hofstadterQRec(n - hofstadterQRec(n - 1)) + hofstadterQRec(
}
```

In Zeile 3 ist die Rekursionsbasis notiert: Wenn  $n$  kleiner/gleich 2 ist, wird 1 zurückgegeben. In allen anderen Fällen wird der Rekursionsschritt ausgeführt, nach dem gleichen Schema wie in der Formel zu sehen ist.

## Aufrufdiagramm

für  $Q(4)$



## Web-Implementation

Hier kannst du dir die besprochenen Sequenzen einmal genauer anschauen.

Fibonacci Hanoi Hofstadter Q

## Quellcode

Mein Quellcode ist einsehbar unter

<https://github.com/Skn0tt/lkAlgorithmik/tree/master/Rekursion>

---

SHARE ON:

[Twitter](#)[Hacker News](#)[Reddit](#)

## Comments (0)

[≡ best](#)[Leave a comment](#)[Sign In](#)

---

Please sign in to create comments.

CommentBox 

© 2024 Simon Knott. [impressum](#). [datenschutz](#). powered by [Jekyll](#), [leonids theme](#). made with

