WIF-Objekorientiertes Programmieren (WIF-OOP)

07 - Rekursion

Fakultät Informatik, TH Rosenheim

Rekursion

Um Rekursion zu verstehen müssen wir zuerst Rekursion verstehen. Ok, Google:

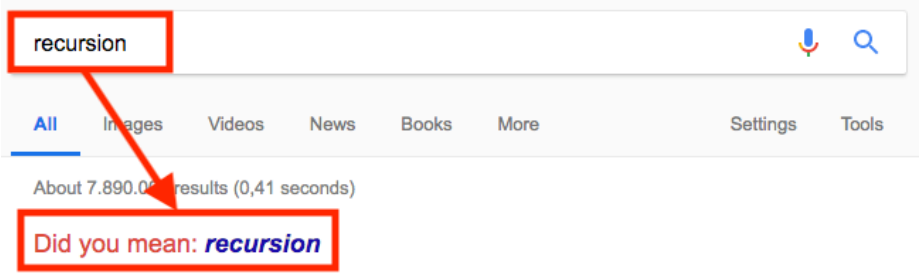

Figure 1: Rekursion

Spaß beiseite. Eine *rekursive Funktion* ist eine Funktion, die sich selbst mit veränderten Argumenten wieder aufruft.

Fakultät

Dieses Prinzip kennen wir bereits aus der Mathematik: Die Fakultät, also $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$, wird in manchen Formelsammlungen *rekursiv* definiert als

$$n! = \begin{cases} 1 & \text{für n} = 1 \text{ (terminal).} \\ n \cdot (n-1)! & \text{für n} > 1 \text{ (rekursiv).} \end{cases}$$

Folgt man dieser Vorschrift, so erhält man durch Ersetzen z.B.

$$
\begin{aligned}
4! &= 4 \cdot 3! && \text{Regel 2} \\
&= 4 \cdot 3 \cdot 2! && \text{Regel 2} \\
&= 4 \cdot 3 \cdot 2 \cdot 1! && \text{Regel 2} \\
&= 4 \cdot 3 \cdot 2 \cdot 1 && \text{Regel 1 (terminal)}
\end{aligned}
$$

Würde man das nun nicht in mathematischer Notation schreiben, sondern in Java, könnte das so aussehen:

```
class Fakultaet {
    static int fakultaet(int n) {
        if (n == 1) {
            return 1;  // Regel 1: terminal
        } else {
            return n * fakultaet(n - 1);  // Regel 2: rekursiv
        }
    }
}
```

Wir sehen also: eine rekursive Funktion hat (mind.) einen **Terminalfall**, in dem ein fester Wert zurückgegeben wird, und (mind.) einen **Rekursionsfall**, in dem die Methode mit verändertem Argument (hier: `n-1`) aufgerufen wird. Das aus der Mathematik bekannte Substitutionsprinzip gilt übrigens auch für Programmiersprachen; dort ist es als Liskovsches Substitutionsprinzip (nach Barbara Liskov) für objektorientierte Sprachen definiert.

Analog ergibt sich die Aufrufreihenfolge, hier als Sequenzdiagram:

Wie man sehen kann, kehrt der Aufruf `fakultaet(4)` des Users erst zurück, nachdem die Rekursionsschritte alle wieder zurückgekehrt sind. Man sagt auch: Die Rekursion "steigt ab", daher auch der Begriff "Rekursionstiefe", im obigen Beispiel 4.

Größter gemeinsamer Teiler (GGT)

Der größte gemeinsame Teiler zweier natürlicher Zahlen kann nach mit dem (einfachen) Euklidschen Algorithmus berechnet werden. Sie kennen vielleicht noch die iterative Formulierung, bei der die beiden Zahlen immer abwechselnd voneinander abgezogen werden, bis

```
int ggT(int a, int b) {
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
```

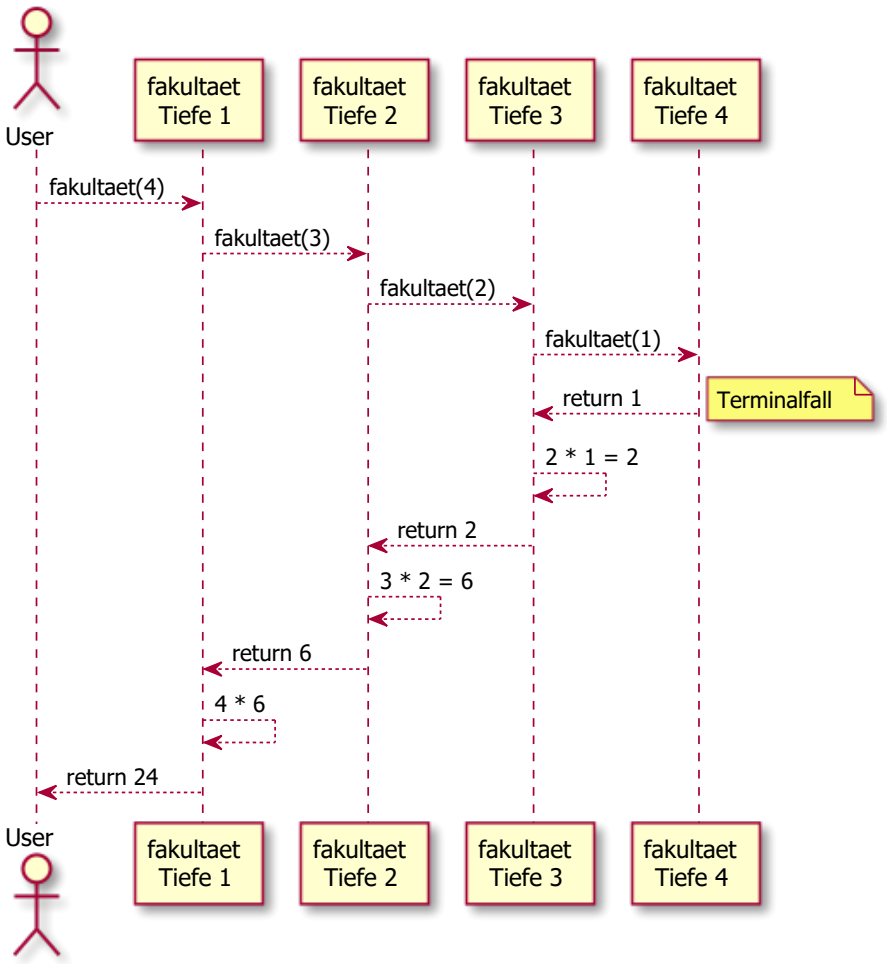

Figure 2: Fakultät

```
    return a;
}
```

Auch diesen Algorithmus kann man rekursiv beschreiben: Sind die beiden Zahlen gleich, so hat man den gemeinsamen Teiler gefunden. Sind sie unterschiedlich, so zieht man die Kleinere von der Größeren ab, und bestimmt hierfür den gemeinsamen Teiler.

```
int ggT(int a, int b) {
    if (a == b)
        return a;
    else if (a < b)
        return ggT(a, b-a);
    else
        return ggT(a-b, b);
}
```

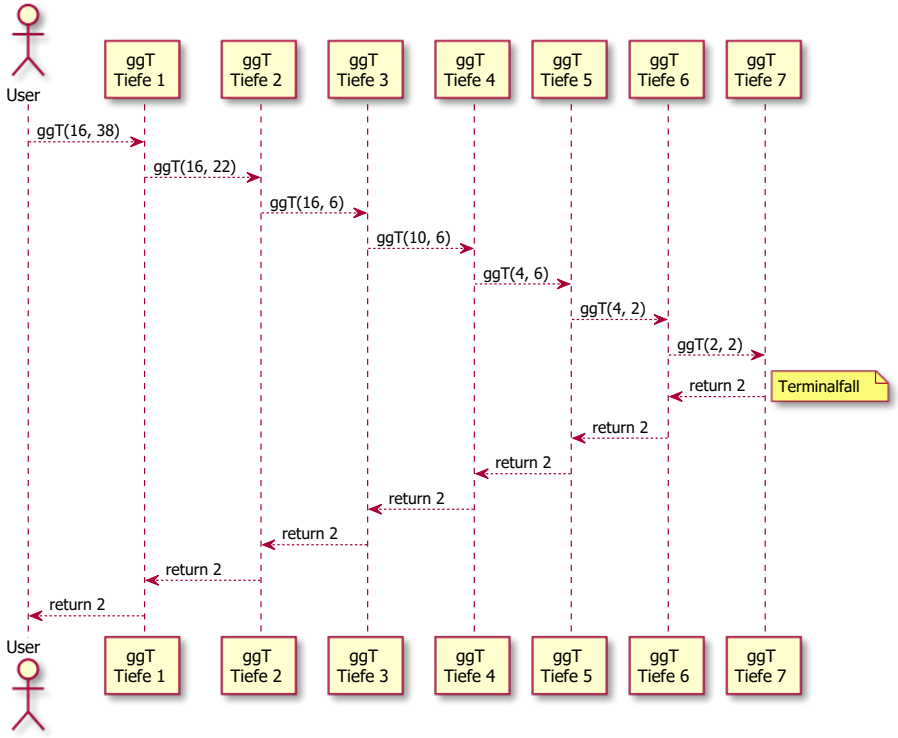

Figure 3: ggT

Fibonacci

Die Fibonacci Zahlenreihe (0, 1, 1, 2, 3, 5, …) hat eine interessante geometrische Eigenschaft. Fügt man Quadrate mit eben diesen Seitenlängen aneinander und zeichnet man jeweils einen Viertelkreis darin, so erhält man eine Annäherung der goldenen Spirale.

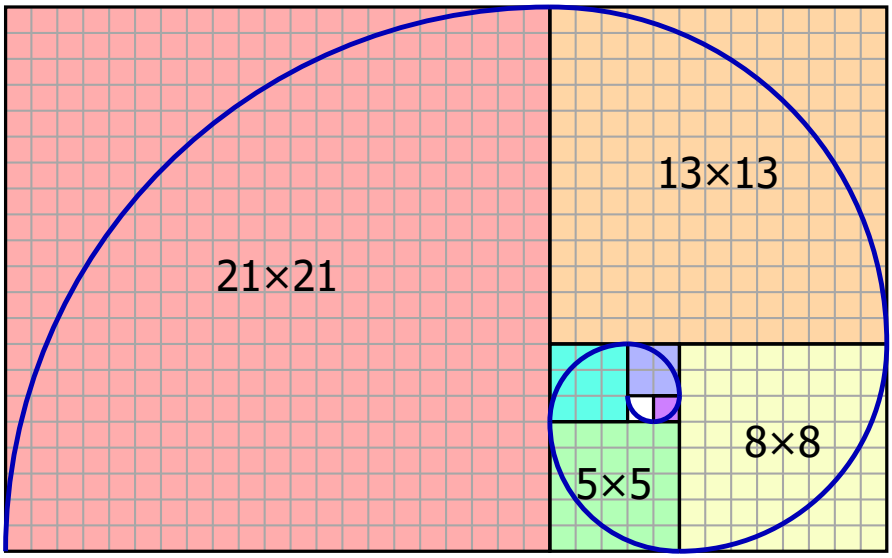

Figure 4: Fibonacci

Quelle: Wikipedia

Die Fibonacci Reihe ist ebenso rekursiv definiert: eine Zahl ist die Summe ihrer beiden Vorgänger.

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

Analog dazu die Java Implementierung:

```
class Fibonacci {
    static int fib(int n) {
        if (n == 0)
            return 0;
        else if (n == 1)
            return 1;
        else
```

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

Diese einfache Implementierung hat aber einen Nachteil: Im Rekursionsfall wird die Methode gleich **zwei Mal** aufgerufen. Das hat zur Folge, dass die Anzahl der rekursiven Aufrufe exponenziell steigt:

```
fib(5) =>
 |    \
 |     \
fib(4) + fib(3) =>
 |    \    \-----+--------
 |     \          \       \
fib(3) + fib(2) + fib(2) + fib(1) =>
 |    \    \        \
 |     \    \        \------------+----------
 |      \    \---+---------       \         \
 |       \        \        \        \         \
fib(2) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1) =>
 |    \
 |     \
fib(1) + fib(0) + ...
```

Allein ein Aufruf von `fib(70)` benötigt bereits mehrere Sekunden bis Minuten zur Berechnung. Dabei fällt auf, dass gewisse Aufrufe *mehrfach* vorkommen (z.B. `fib(3)`), also im Endeffekt Arbeit unnötig mehrfach verrichtet wird.

Um diesen Mehraufwand zu vermeiden kann man Caches verwenden, z.B. realisiert als Map: Argument auf Ergebnis:

```
class Fibonacci {
    static private Map<Integer, Integer> cache = new HashMap<>();

    static int fibCached(int n) {
        if (n == 0)
            return 0;
        else if (n == 1)
            return 1;
        // bereits ausgerechnet?
        else if (cache.containsKey(n))
            return cache.get(n);
        else {
            int a = fibCached(n-1);
            int b = fibCached(n-2);

            // Cache pflegen
            if (!cache.containsKey(n-1))
```

```
                cache.put(n-1, a);
            if (!cache.containsKey(n-2))
                cache.put(n-2, b);

            return a + b;
        }
    }
}
```

Und schon terminiert auch ein `fibCached(70)` rasend schnell.

Eine weitere Optimierung der obigen Rekursion wäre die Vorschrift genauer zu betrachten: $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$. Ein Wert hängt also immer genau von seinen zwei Vorgängern ab. Diese kann man nun auch als Argumente in einer Hilfsfunktion "mitschleifen".

```
class Fibonacci {
    static int fibBesser(int n) {
        return fibHilf(n, 0, 1);  // initialisiere Terminalfälle
    }

    private static int fibHilf(int n, int a, int b) {
        if (n == 0)
            return a;
        else if (n == 1)
            return b;
        else
            return fibHilf(n-1, b, a+b);  // angepasste Parameter!
    }
}
```

Rekursion auf Datenstrukturen

Nun sind Rekursionen für mathematische Probleme zwar interessant, aber in der angewandten Informatik selten praxisrelevant. Wir wollen daher exemplarisch ein paar Probleme herausnehmen, welche durchaus konkreten Bezug zur Praxis haben. Dort ist die rekursive Formulierung oft knapper und einfacher als eine Iterative.

Palindrom

Wir beginnen mit dem bereits bekannten Palindromproblem: Ist ein Wort (oder Satz) vorwärts wie rückwärts gelesen das selbe?

Wir kennen die iterative Variante, welche die Buchstaben von hinten wie von vorn der Reihe nach auf Gleichheit prüft.

```
class Palindrom {
    static boolean istPalindrom(String s) {
        for (int i = 0; i < s.length()/2; i++)
            if (s.charAt(i) != s.charAt(s.length()-1-i))
                return false;
        return true;
    }
}
```

Eine rekursive Variante wäre:

```
class Palindrom {
    static boolean istPalindrom(String s) {
        if (s.length() < 2)
            return true; // Leer und ein Zeichen sind immer Palindrom
        else if (s.charAt(0) != s.charAt(s.length() - 1))
            return false;  // Oops.
        else
            // angenommen erster und letzter passen, was ist mit dem Rest?
            return istPalindrom(s.substring(1, s.length() - 1));
    }
}
```

Hierbei unterscheidet man zunächst zwei Terminalfälle: Ist ein String kürzer als zwei Zeichen, so ist es ein Palindrom; passt erster und letzter Buchstabe nicht, so ist es kein Palindrom. Hingegen beim letzten `else` Fall ist bereits klar, dass erster und letzter Buchstabe gleich sind, verbleibt nur noch der Teilstring von 1 bis zum vorletzten Buchstaben zu verifizieren.

Binäre Suche im Array

Ein eher algorithmisches Problem ist die Suche, ob und welche Position ein Wert in einem Array hat. Ist das Array bereits sortiert, so kann eine *binäre Suche* durchgeführt werden, bei der man für einen Bereich des Arrays zunächst das Element in der Mitte vergleicht, und bei Ungleichheit entweder in der linken oder rechten Arrayhälfte weiter sucht.

```
class Suchen {
    static <T extends Comparable<T>> int binarySearch(T[] sorted, T value) {
        // Hilfsmethode: Durchsuche von Anfang bis Ende
        return binarySearchHelp(sorted, value, 0, sorted.length);
    }

    private static <T extends Comparable<T>> int binarySearchHelp(
        T[] sorted, T value, int from, int to) {

        // schon zu weit gelaufen?
        if (from >= to)
```

```
            return -1;

        // Offset für Pivot finden, Wert vergleichen
        int p = from + (to - from) / 2;
        int c = value.compareTo(sorted[p]);  // Vorsicht beim Index!

        // Treffer?
        if (c == 0)
            return p;
        if (c < 0)
            // links weitersuchen
            return binarySearchHelp(sorted, value, from, p);
        else
            // rechts weitersuchen
            return binarySearchHelp(sorted, value, p+1, to);
    }
}
```

Man sieht: Eine iterative Lösung wäre hier sehr ähnlich:

```
class Suchen {
    private static <T extends Comparable<T>> int binarySearchIt(T[] sorted, T value) {

        int from = 0;
        int to = sorted.length;
        while (from < to) {
            int p = from + (to - from) / 2;
            int c = value.compareTo(sorted[p]);

            // "Terminalfall"
            if (c == 0)
                return p;

            // "Rekursionsfall"
            if (c < 0)
                to = p;  // nur noch bis links vom Pivot
            else
                from = p + 1;  // nur noch ab rechts vom Pivot
        }

        return -1;
    }
}
```

Obwohl diese Version durchaus sehr übersichtlich ist, so ist die Rekursive semantisch gesehen etwas deutlicher.

Rekursion für Listen

Eine verkettete Liste ist, vom UML Diagram ausgehend, i.d.R. eine "rekursive Struktur": Eine Liste hat entweder ein erstes Element oder es hat keins; jedes Element wiederum hat ein optionales Nachfolgeelement.

Möchte man nun die Größe (`size`) der Liste bestimmen, so muss man wieder Terminal- und Rekursionsfälle betrachten.

1. Eine Liste welche kein erstes Element hat ist leer.
2. Gibt es ein erstes Element, so kann man dieses Fragen wie lang es denn ist.
3. Ein Element ist in jedem Fall mind. 1 lang; gibt es einen `next` Nachfolger, so muss man dazu noch die Länge des Nachfolgers addieren.

```
class Liste<T> {
    Element first;

    public int size() {
        if (first == null)  // Terminalfall 1
            return 0;
        else
            return first.size();  // Hilfsmethode!
    }

    class Element {
        T value;
        Element next;
        int size() {
            if (next == null)  // Terminalfall 3a
                return 1;
            else
                return 1 + next.size();
        }
    }

    // ...
}
```

Rekursion für Bäume

Das obige Beispiel (Liste) ist zugegebener Maßen nicht umbedingt sehr intuitiv. Klarer sollte der Vorteil von Rekursion sein, wenn man auf baumartigen Strukturen arbeitet. Hier können wir z.B. die Größe (`size`) rekursiv definieren:

1. Terminalfall: Gibt es keinen Wurzelknoten, so ist der Baum leer.
2. Rekursionsfall: Gibt es einen Wurzelknoten, so ist die Baumgröße mind. 1 (Terminalfall), sowie zusätzlich die Größe des linken und rechten Teil-

baums (Rekursion, sofern vorhanden).

```
public class Baum<T extends Comparable<T>> {
    class Element {
        T value;
        Element left, right;
        Element(T value) { this.value = value; }
        int size() {
            return 1 +
                (left == null ? 0 : left.size()) +
                (right == null ? 0 : right.size());
        }
    }

    Element root;

    int size() {
        if (root == null) {
            return 0;
        } else {
            return root.size();
        }
    }
}
```

Man sieht, dass diese Definition bzw. Implementierung deutlich intuitiver ist, als iterativ mit Agenda.

Kochrezept für rekursive Funktionen

Wie geht man also im Allgemeinen mit Rekursion um, wie erstellt man eine rekursive Lösung?

1. Terminalfälle bestimmen. Wann ist die Lösung trivial?
2. Rekursionsfälle bestimmen. Wie kann ich das Problem auf ein kleineres runterbrechen?
3. Rekursion zusammensetzen: Brauche ich eine Hilfsmethode, wie muss die Signatur aussehen, wie müssen die Argumente beim rekursiven Aufruf verändert werden?

Bei der Implementierung ist das Schema dann entsprechend ähnlich:

```
// kein valides Java...
int rekursiv(...) {
    if (Terminalfall) {
        return /* fester Wert */
    } else {
```

```
        // Rekursionsfall: mind. 1x rekursiv aufrufen!
        return rekursiv(/* veränderte Argumente*/);
    }
}
```

Arten der Rekursion

Wir haben oben nun eine Reihe von verschiedenen Rekursionsarten gesehen.

- Lineare Rekursion: genau ein rekursiver Aufruf, z.B. Fakultät.
- Repetetive Rekursion (Rumpfrekursion, engl. *tail recursion*): Spezialfall der linearen Rekursion, bei der der rekursive Aufruf die letzte Rechenanweisung ist. Diese Rumpfrekursionen können direkt in eine iterative Schleife umgewandelt werden (und umgekehrt). Beispiel: verbesserte Implementierung der Fibonacci Funktion.
- Kaskadenartige Rekursion: in einem Zweig der Fallunterscheidung treten *mehrere* rekursive Aufrufe auf, was ein lawinenartiges Anwachsen der Funktionsaufrufe mit sich bringt. Beispiel: einfache Implementierung der Fibonacci Funktion.
- Verschränkte Rekursion: Eine Methode `f()` ruft eine Methode `g()`, die wiederum `f()` aufruft.

Zusammenfassung

- Eine **rekursive Methode** ist eine Methode, die sich selbst wieder aufruft; charakteristisch sind die Abwesenheit von `for` und `while`, sowie klare `if-else` Anweisungen, welche Terminal- von Rekursionsfall unterscheiden.
- Bei **kaskadenartigen** Rekursionen, also mehr als ein rekursiver Aufruf pro Durchlauf, können je nach Problemstellung Caches die Berechnung enorm effizienter gestalten.
- **Repetitive Rekursion** ist wünschenswert, da diese effektiv als `for` bzw. `while` Schleife realisiert werden könnten.
- Für obige braucht man oft Variablen, welche die Zwischenergebnisse im rekursiven Aufruf codieren.