Rekursive Funktionen

Was sind rekursive Funktionen?

Rekursive Funktionen sind Funktionen, die sich auf sich selbst beziehen bzw. Funktion, die sich selbst aufrufen.

Wozu braucht man rekursive Funktionen?

Rekursive Funktionen braucht man, um typisch rekursive Probleme möglichst einfach lösen zu können. (Ein sehr schlauer Satz – ich weiß *g*; Erklärung folgt…)

Gibt es Alternativen zu rekursiven Funktionen?

Na logisch! Jedes typisch rekursive Problem ist immer auch iterativ lösbar. Mathematiker haben das sogar bewiesen.

Was ist eine iterative Lösung im Vergleich zur rekursiven Lösung?

Kurzer Einschub:

> Nach dem Holländer Edsger Wybe Dijkstra kann (und soll) man jedes Problem der Programmierung mithilfe der drei Strukturmodelle
> - Sequenz (= Befehlsabfolge)
> - Alternation (= Verzweigung, if-else…)
>
> und
> - Iteration (= Wiederholungsanweisung, Schleife, while…, for…)
>
> lösen. Leider konnte er das nicht beweisen. Bisher konnte aber auch noch niemand das Gegenteil beweisen. Wir müssen demzufolge davon ausgehen, dass der Holländer Recht hat.

Eine iterative Lösung erfolgt also mithilfe von Schleifen, wohingegen bei der rekursiven Lösung charakteristisch ist, dass eine Funktion sich selbst aufruft.

Wenn man rekursive Probleme auch mit einfachen Schleifen lösen kann, warum brauche ich dann den rekursiven Sch…?

Dazu möchte ich noch einmal auf meinen schlauen Satz von weiter oben zurückgreifen: *„Rekursive Funktionen braucht man, um typisch rekursive Probleme möglichst einfach lösen zu können."*
Von diesem Satz kann man nun auch die Antwort der Frage ableiten, nämlich dass typisch rekursive Probleme auf iterative Weise nicht einfachst-möglich lösbar sind. Eine iterative Lösung von rekursiven Problemen kann mitunter abartig kompliziert werden.

Zu abstrakt?

Dann ein Beispiel:

Hier ein einfaches typisch rekursives Problem aus der Mathematik: die Fakultät.
Nur zur Erinnerung: die Fakultät von 5 ist 5 * 4 * 3 * 2 * 1
Oder anders formuliert: die Fakultät von 5 ist 5 * Fakultät(4)
Die allgemeine Funktionsdefinition lautet: Fakultät(x) = x * Fakultät(x – 1)

Letztere Definition ist rekursiv, weil der Funktionsterm, also „x * Fakultät(x – 1)", einen Aufruf der Funktion selbst beinhaltet.

Dabei ist festzustellen, dass der innere Funktionsaufruf nicht denselben Parameterwert enthält als die Funktion selbst. Hier ist außen der Parameterwert „x“ und innen ist der Parameterwert „x – 1“. Beide Funktionswerte unterscheiden sich also. Dieser Fakt ist sehr wichtig! Würden beide Parameterwerte identisch sein, so käme dies einer „Endlosschleife“ gleich. Eine solche, beispielsweise in Java programmierte, Funktion käme nie zu einem Ende. Das heißt, ein Ende gibt es schon – wenn auch ein Unschönes. Rekursive Funktionen laufen nie ewig. Spätestens, wenn der Speicher droht „überzulaufen“, bricht das Betriebssystem den ursächlichen Prozess ab oder aber der Rechner stürzt ab; Dazu später mehr. Noch einmal: Innere rekursive Funktionsaufrufe müssen mit abgewandelten Funktionsparametern stattfinden. Ich betone das nur deshalb so deutlich, weil rekursive Funktionen – im Gegensatz zu unserem nahezu „trivialen“ Fakultäts-Beispiel – generell dazu tendieren kompliziert zu sein. Deshalb sollte man auf solche Grundregeln achten, damit man zumindest gegen die gröbsten Fehlerquellen ausreichend gewappnet ist.

Es gibt noch einen zweiten Aspekt, auf den man beim Entwurf rekursiver Funktionen, meiner Meinung nach, dringend achten sollte. Dazu werfen wir ein weiteres Mal einen Blick auf unsere – bisherige – allgemeine Funktionsdefinition zur Lösung der Fakultät:

Zur Erinnerung: Fakultät(x) = x * Fakultät(x – 1)

An dieser Definition fehlt nämlich noch etwas. So wie sie hier dasteht ist sie noch ziemlich falsch. So falsch, dass sie problemlos jedes Computerprogramm zum Absturz zwingen würde. Man findet das Problem indem man sie einmal exemplarisch berechnet.

Beispielsweise für x = 2:

Fakultät(2) = 2 * Fakultät(1)
Fakultät(1) = 1 * Fakultät(0)
Fakultät(0) = 0 * Fakultät(-1)
Fakultät(-1) = -1 * Fakultät(-2)
Fakultät(-2) = -2 * Fakultät(-3)
Fakultät(-3) = -3 * Fakultät(-4)
…

Problem erkannt? – Es fehlt die Abbruchbedingung! Die Funktion läuft ewig – fast ewig… Deshalb aufpassen: Jede rekursive Funktion braucht immer eine Abbruchbedingung. Es folgt die korrigierte Version der Funktion zur Berechnung der Fakultät:

$$\text{Fakultät}(x) = \begin{cases} x * \text{Fakultät}(x-1) & \text{für } x > 0 \\ 1 & \text{für } x = 0 \end{cases}$$

Was fehlt jetzt noch? – Natürlich, die Definitionsmenge!

D = N

Für x sind also alle natürlichen Zahlen inklusive der 0 erlaubt. Die Fakultät von 0 gibt nämlich per Definition den Wert 1. Sich beim Entwurf einer Funktion über die Definitionsmenge Gedanken zu machen und im Programm die Parameterwerte der Funktion ggf. entsprechend zu prüfen ist zwar eine nicht zu verachtende Angelegenheit, aber keine Besonderheit von rekursiven Funktionen. Das gilt generell bei allen Funktionen. Deshalb werde ich diesen Punkt der Einfachheit halber im Folgenden weitgehend vernachlässigen.

Also noch mal: Worauf ist beim Design rekursiver Funktionen zu achten?

- In der Gesamtheit müssen sich die Parameterwerte bei rekursiven Funktionsaufrufen von den „äußeren“ Parameterwerten unterscheiden.
- Eine Abbruchbedingung muss generell gegeben sein.
- (Die Definitionsmenge ist festzulegen und ggf. abzuprüfen)

(Um der Korrektheit Folge zu leisten: Der erste Aspekt ist dann nicht richtig, wenn der Abbruch der Funktion von globalen Variablen abhängig ist, die in der Funktion bzw. durch die Funktion dementsprechend abgeändert werden. Dies wäre allerdings ein schlechtes Design und sollte tunlichst vermieden werden.)

Können wir jetzt endlich mal anhand eines anschaulichen Beispiels klären, warum man rekursive Funktionen braucht, wenn alle Probleme auch iterativ lösbar sind?

Natürlich! Ich weiß schon, ich bin jetzt etwas abgeschweift, aber wir mussten einfach zunächst die Grundlagen der rekursiven Funktionen klären, um diese Sache angehen zu können.

Also, bleiben wir bei obigem Beispiel der Fakultätsberechnung und vergleichen hierzu sowohl eine rekursive als auch eine iterative Implementierung in Java.

Zunächst die rekursive Funktion:

```
public static int fakultaet(int zahl) {

      if (zahl == 0) return 1;
      else return zahl * fakultaet(zahl – 1);
}
```

Und hier das iterative Pendant:

```
public static int fakultaet(int zahl) {

      int fak = 1;

      for (int i = 2; i <= fak; fak++) {
            fak = fak * i;
      }
      return fak;
}
```

Weiter oben habe ich behauptet, dass typisch rekursive Probleme mit rekursiven Funktionen einfacher zu lösen sind, als auf rein iterativem Wege. Im vorliegenden Beispiel spricht dafür, dass die zweite Funktion in jedem Fall ein bisschen länger als die Erste ist. Außerdem benötigt sie zwei Hilfsvariablen (fak und i) – die rekursive Funktion nicht. Viel komplizierter scheint sie also offensichtlich nicht zu sein. Einige würden sagen, dass sie sogar einfacher ist. Das liegt vor allem daran, dass sich das Problem Fakultätsberechnung alles in Allem als zu einfach darstellt. Dennoch sei darauf hingewiesen, dass die Rekursive, tendenziell zumindest, die intuitivere Lösung ist und das Problem selbst besser fasst.

Wesentlich krassere Unterschiede zwischen iterativer und rekursiver Lösung gäbe es bei komplexeren rekursiven Problemen. Darauf, hier Solche exemplarisch darzustellen, möchte ich aber gerne verzichten. Wir wollen es ja nicht gleich übertreiben! – ich bin dafür, dass Sie mir das einfach so glauben, schließlich habe ich ja schon gezeigt, dass selbst die rekursive Funktion unseres Fakultätsbeispiels einfacher ist, sprich mit weniger Ecken und Kanten ausfällt (ohne Hilfsvariablen), als das iterative Gegenstück.

Und warum bringt eine rekursive Funktion ohne Abbruchbedingung einen Speicherüberlauf?

Das liegt am Stack. Wir sollten zunächst klären, was ein Stack ist.

Na gut, also was ist ein Stack?

Also schön, wenn Sie das so sehr interessiert *g*: Ein Stack, auch häufig als Stapel oder Keller bezeichnet, ist eine spezielle Datenstruktur, die in der Informatik sehr oft ihre Anwendung findet. Sein

Prinzip wurde bereits in den 1950er Jahren von dem, in Regensburg geborenen, Informatikpionier Friedrich Ludwig Bauer entwickelt.

Und so funktioniert der Spaß: Man kann sich einen Stack als einen Stapel Bücher vorstellen, welcher auf einem Tisch steht. Man kann lediglich ein Buch oben drauflegen oder von oben ein Buch herunternehmen. Aus der Mitte ein Buch herausziehen oder gar das unterste Buch herausnehmen, während sich noch weitere Bücher auf dem Stapel befinden, geht nicht; – Immer nur oben ein Buch drauflegen oder das oberste Buch herunternehmen. Dieses Prinzip nennt man auch „LIFO", also „last in, first out". Das Buch, das zuletzt auf den Stapel gelegt wurde („last in") muss als erstes Buch wieder heruntergenommen werden („first out") (– sofern man sich nicht dazu entschließt, stattdessen weitere Bücher auf den Stapel zu legen).

Jenes „Kellerungsprinzip" – so nannte Bauer den Stack – findet, wie schon erwähnt, in der Informatik rege Anwendung. So verwenden die allermeisten Programmiersprachen (außer COBOL vielleicht) einen so genannten „Funktionsstack". Und der funktioniert so: Wird eine Funktion geöffnet, so bedeutet das, dass sie „auf den Stapel gelegt" wird. Sobald sie geschlossen wird, wird sie von Selbigem wieder heruntergenommen.

Dabei bedeutet „eine Funktion auf den Stapel legen", dass man Informationen zur Funktion in einem bestimmten Bereich im Speicher, dem sog. „Stack", hinterlegt. Meist befindet sich dieser Speicherbereich, statt im normalen Hauptspeicher, gleich im superschnellen Cache-Speicher des Prozessors. Dieser ist etwa um den Faktor 1000 schneller. An Informationen werden i.d.R. hinterlegt: Funktionsparameter, lokale Variablen und eine Rücksprungadresse, die angibt, wo es im Programm nach Abarbeitung dieser Funktion weitergeht.

Warum also bei einer „Endlos-Rekursion" ein Speicherüberlauf?

Hierzu noch einmal unsere rekursive Funktion zur Fakultätsberechnung:

```
public static int fakultaet(int zahl) {

      if (zahl == 0) return 1;
      else return zahl * fakultaet(zahl – 1);
}
```

Errechnen wir beispielsweise die Fakultät von 3, so passiert auf dem Stack folgendes:

Lege fakultaet(3) auf den Stack
Lege fakultaet(2) auf den Stack
Lege fakultaet(1) auf den Stack
Lege fakultaet(0) auf den Stack
Entferne fakultaet(0) vom Stack
Entferne fakultaet(1) vom Stack
Entferne fakultaet(2) vom Stack
Entferne fakultaet(3) vom Stack

Und fakultaet(3) gibt uns schließlich das gesuchte Ergebnis, nämlich den Wert 6 zurück. Wenn wir aus unserer Funktion die Abbruchbedingung entfernen, haben wir (ausnahmsweise bewusst) eine Art Endlosschleife kreiert. Sie sieht dann so aus:

```
public static int fakultaet(int zahl) {

      return zahl * fakultaet(zahl – 1);
}
```

Jetzt werden ständig neue Instanzen der Funktion „fakultaet" auf den Stack gelegt, wobei der entsprechende Parameterwert jeweils um den Wert 1 verringert ist. Das passiert so lange, bis auf dem Stack kein Platz mehr ist. Dann wird üblicherweise ein „stack overflow" bzw. ein „Stapelüberlauf" gemeldet und der ursächliche Prozess wird daraufhin „gekillt" bzw. „abgebrochen".

Bei unserer iterativen Funktion kann ein solcher Speicherüberlauf nicht passieren. Zur Erinnerung – so sieht sie aus:

```
public static int fakultaet(int zahl) {

      int fak = 1;

      for (int i = 2; i <= fak; fak++) {
            fak = fak * i;
      }
      return fak;
}
```

Auf folgende Weise könnten wir eine Endlosschleife provozieren:

```
public static int fakultaet(int zahl) {

      int fak = 1;

      for (int i = 2; ; fak++) {
            fak = fak * i;
      }
      return fak;
}
```

Unsere for-Schleife findet jetzt zwar kein Ende mehr, aber ein Speicherüberlauf wird nicht erzeugt. Es wird ja auch nicht pro Schleifendurchgang neuer Speicher reserviert. Der Prozess wird vom Betriebssystem nicht abgebrochen. Wenn man ihn nicht händisch abbricht, läuft er mindestens bis zum nächsten Stromausfall – mit USV sogar noch länger.

Welcher der beiden Fälle nun der „bessere“ ist, ob rekursive oder iterative Endlosschleife, sei dahingestellt. Beide Zustände sind sicherlich denkbar ungünstig. Eine geeignete Abbruchbedingung einzubauen ist deshalb sehr zu empfehlen.

Gibt es Argumente für eine iterative Lösung eines rekursiven Problems?

Geklärt ist bereits, dass ein rekursives Problem in aller Regel, außer vielleicht bei nahezu trivialen Aufgabenstellungen (siehe Fakultätsberechnung), für den Programmierer mithilfe rekursiver Funktionen einfacher lösbar ist.

Es gibt meiner Meinung nach nur einen Aspekt, der dann trotzdem für eine iterative Lösung spricht und zwar das Performance-Argument. Die ständige Beanspruchung des Stacks, das ständige Anlegen neuer Funktionsinstanzen beansprucht den Prozessor ziemlich. Ein Funktionsaufruf an sich ist für einen Rechner relativ stressig. So kann man bei sehr zeitkritischen Anwendungen versuchen, durch einen iterativen Ansatz die Performance zu verbessern.

Gibt es sonst noch etwas über rekursive Funktionen zu wissen?

Das beantwortet folgende rekursive Funktion (oder auch nicht):

```
public static String GibtEsNochWasZuWissen() {

      System.out.println("Keine Ahnung – ich frag mal nach.");
      return GibtEsNochWasZuWissen();
}
```