1. Kapitel: Rekursion: Theorie

Grundidee: *"Löse ein Problem, indem du es auf ein oder mehrere* ***gleichartige, aber "kleinere"*** *Teilprobleme zurückführst, bis das Problem so klein geworden ist, daß die Lösung trivial ist."*

Rekursion im Alltag: "Bild im Bild im Bild ...", "Russische Puppen", ...

Im Code:

- Eine rekursive Funktion ruft sich ***selbst*** auf, entweder ***direkt*** (`func1` ruft `func1` auf) oder ***indirekt*** (`func1` ruft `func2` auf, `func2` ruft `func3` auf, ... , `funcx` ruft wieder `func1` auf).
- Jede rekursive Funktion muß mindestens einen ***nichtrekursiven Zweig*** (d. h. einen `if`-Zweig, der ohne rekursiven Aufruf ein Ergebnis liefert) enthalten. Dieser stellt das Ende der Rekursion dar.
- Jeder rekursive Aufruf innerhalb der rekursiven Funktion muß dem Rekursionsende ***näher sein*** als der aktuelle Aufruf (d. h. mit einer kleineren Datenstruktur, einer kleineren Zahl, oder weniger noch zu findenden Lösungselementen erfolgen).

Sind die letzten beiden Punkte nicht erfüllt, kommt es zur ***Endlosrekursion*** (ähnlich einer Endlosschleife): Es erfolgen unendlich viele geschachtelte Aufrufe der Funktion (bis das Programm wegen Stacküberlauf abbricht), ohne daß jemals ein Ergebnis zurückgegeben wird.

Bei der Ausführung:

- Es werden der Reihe nach ineinander geschachtelt mehrere Aufrufe derselben Funktion gestartet.
- Jeder Aufruf der Funktion bekommt unabhängig von den bereits laufenden Aufrufen der Funktion seine eigenen lokalen Variablen (Ausnahme: lokale `static`-Variablen) und seine eigenen Argumente.
  Globale Variablen und Variablen, auf die nur ein Pointer übergeben wurde, gibt es hingegen nur einmal für alle Aufrufe gemeinsam.
- Die Aufrufe kehren in umgekehrter Reihenfolge des Aufrufs wieder zurück.

Die Verarbeitung von rekursiven Aufrufen ist der Hauptgrund, warum moderne Programmiersprachen lokale Variablen, Argumente und Returnadresse dynamisch auf einem ***Stack*** ablegen und nicht auf fix zugeteilten Speicherplätzen.

Zur Theorie: Für eine universelle Programmiersprache (d. h. eine Programmiersprache, mit der man alles berechnen kann, was theoretisch berechenbar ist) genügen `if`, Rekursion, und "normale" Variablen. Man braucht keine Schleifen: Alle Schleifen lassen sich durch Rekursionen ersetzen.

Bietet eine Programmiersprache oder ein Rechenautomat keine Rekursion, sondern nur `if` und Schleifen, braucht man zusätzlich potentiell unendlich große Arrays, um universell zu sein.

Die rekursive Definition: Viele Datenobjekte, die sinnvollerweise rekursiv verarbeitet werden, sind auch rekursiv definiert: Ein solches Datenobjekt ist entweder elementar (entspricht dem Ende der Rekursion), oder aus gleichartigen, aber kleineren Unterobjekten zusammengesetzt (entspricht dem rekursiven Zweig).

Beispiele:

- Ein Binärbaum ist entweder leer, oder er besteht aus einem Knoten, an dem links und rechts jeweils wieder ein Binärbaum dranhängt.
- Ein arithmetischer Ausdruck besteht (vereinfacht) entweder aus einer Zahl oder einem arithmetischen Ausdruck in Klammern oder zwei oder mehreren arithmetischen Teilausdrücken mit je einem Operator dazwischen (in dieser Art sind auch Programmiersprachen definiert, siehe später).

Rekursive versus iterative Programme:

- *Jede* rekursive Funktion läßt sich (sogar automatisch) in ein Programm umformen, das nur Schleifen statt rekursiven Aufrufen enthält und die Rekursion durch eine händisch ausprogrammierte stack-artige Datenverwaltung (in einem Array oder mit verketteten Listen) ersetzt.

  Das resultierende Programm braucht im Wesentlichen gleich viel Speicherplatz und gleich viel Rechenzeit wie das rekursive Original, ist aber wesentlich länger und viel schwerer zu verstehen (abschreckendes Beispiel: Iterativer Quicksort). Man bleibt daher besser bei der rekursiven Variante!

- Eine ***tail-rekursive*** Funktion ist eine Funktion, die in jedem Zweig maximal einen rekursiven Aufruf enthält, und den als letzten Schritt der Berechnung: Der Returnwert des rekursiven Aufrufs wird direkt und unverändert als Returnwert des aktuellen Aufrufs zurückgegeben, d. h. im `return` steht nur der rekursive Aufruf und sonst nichts. Der ggT nach Euklid (siehe unten) ist ein Beispiel für eine tail-rekursive Funktion.

  Jede tail-rekursive Funktion läßt sich in ein Programm mit einer Schleife und *ohne* zusätzliches Array oder irgendwelche dynamischen Datenstrukturen umformen. Ebenso läßt sich jede Schleife in eine tail-rekursive Funktion verwandeln.

  In einigen Fällen gelingt es durch Einführen eines zusätzlichen Parameters, in dem das Zwischenergebnis mitgeführt wird, eine nicht tail-rekursive in eine tail-rekursive Funktion (und dann ev. in eine Schleife) zu verwandeln (wie z. B. bei der Potenz oder Fakultät).

  Das umgeformte Programm ist normalerweise um einen konstanten Faktor schneller als das rekursive, und es braucht vor allem — unabhängig vom Eingabewert — nur konstant viel Speicherplatz, während der Speicherbedarf der rekursiven Variante mit der Eingabegröße wächst (im Normalfall n Stack-Frames mit Argumenten, lokalen Variablen und Returnadresse für eine Eingabe der Größe n).

  In diesen Fällen sollte man also die nichtrekursive Variante verwenden!

- Besonders ineffizient sind manche rekursive Funktionen, die das Ergebnis für eine Zahl n aus zwei oder mehr rekursiven Aufrufen für kleinere Zahlen berechnen, wie die Fibonaccizahlen oder die Binomialkoeffizienten (siehe unten): Sie brauchen für eine Eingabe n größenordnungsmäßig 2^n rekursive Aufrufe!

  Wenn man ein Array der Größe n (bzw. $n * n$ bei den Binomialkoeffizienten) nimmt und mit einer Schleife "von unten nach oben" (von 1 aufwärts statt wie die rekursive Funktion von n abwärts) jedes Element aus den schon ermittelten Elementen berechnet, kommt man hingegen mit n statt 2^n Schritten aus (das ist ein *gigantischer* Laufzeitunterschied!)!

  Die Idee läßt sich verallgemeinern (*"memorizing"*): Wenn

  * die Rekursion über eine ganze Zahl n läuft,
  * die rekursive Funktion möglicherweise mehrmals für das gleiche n aufgerufen wird,
  * und jeder Aufruf für ein und dasselbe n auch ein und denselben Wert liefert (d. h. nicht von irgendwelchen globalen Variablen oder anderen Argumenten abhängt),

dann ist es sinnvoll,

* beim ersten Aufruf das Ergebnis rekursiv zu berechnen und in einem globalen oder statischen Array abzuspeichern
* und bei jedem weiteren Aufruf für dasselbe n sofort den gespeicherten Wert zurückzugeben!

2. Kapitel: Rekursion: Beispiele

Rekursiv definierte mathematische Funktionen:

Viele mathematische Funktionen über natürliche Zahlen lassen sich rekursiv definieren[1]:

- Die Fakultät $n!$ ist 1 für $n = 0$ und $n * (n-1)!$ sonst.
- Die Fibonaccizahl f_n ("theoretische Vermehrung der Kaninchen") ist 0 für $n = 0$, 1 für $n = 1$, und $f_{n-2} + f_{n-1}$ sonst.
- Der Binomialkoeffizient $\begin{pmatrix} n \\ k \end{pmatrix}$ (Werte des Pascal'schen Dreiecks) ist definiert als 0 für $k > n$, 1 für $k = 0$ oder $k = n$, und $\begin{pmatrix} n-1 \\ k-1 \end{pmatrix} + \begin{pmatrix} n-1 \\ k \end{pmatrix}$ für $0 < k < n$.

Enthält die Definition einen rekursiven Aufruf, so sind schlimmstenfalls n rekursive Aufrufe nötig (also akzeptabel), enthält sie derer zwei, braucht man in etwa 2^n Aufrufe (also für nichttriviale n sehr langsam!).

Auch der größte gemeinsame Teiler von a und b läßt sich rekursiv definieren und berechnen (*rekursiver Euklid'scher Algorithmus*):

- Ist $b = 0$, so ist das Ergebnis a.
- Sonst ist das Ergebnis gleich dem größten gemeinsamen Teiler von b und (a mod b).

Rekursiv definierte Datenstrukturen:

Der Algorithmus folgt meist der Definition der Datenstruktur. Beispiele:

Suchen in einem binären Baum:

```
tree *search(int n, tree *b) {
  if (b == NULL) return NULL;
  if (n == b->value) return b;
  if (n < b->value) return search(n, b->left);
  else return search(n, b->right);
}
```

(das Durchwandern eines Baumes z. B. zum Ausdrucken erfolgt analog)

Auswerten arithmetischer Ausdrücke: (vereinfacht)

- Werte Zahl oder Klammerausdruck aus: Liefert Zwischenergebnis.
- Solange ein Operator folgt:
  * Werte Zahl oder Klammerausdruck dahinter aus.
  * Berechne neues Zwischenergebnis.
- Returniere das Zwischenergebnis.

---

[1] Im Prinzip sind ja auch die natürlichen Zahlen selbst rekursiv definiert: Eine natürliche Zahl ist entweder 0 oder der Nachfolger einer natürlichen Zahl.

Beachte die indirekte Rekursion:
"Arithmetischer Ausdruck" ruft "Zahl oder Klammerausdruck" auf und umgekehrt.

"Divide and Conquer": "Teile und herrsche"

Allgemeine Vorgangsweise:

- Ist das Problem klein genug, berechne die Lösung direkt.
- Sonst:
  - Zerlege das Problem in zwei oder mehrere kleinere Teile.
  - Löse jeden Teil für sich.
  - Berechne die Lösung aus den Teillösungen.

Beispiele:

- Quicksort und Mergesort.
- Rekursive Multiplikation langer Zahlen (vier Multiplikationen halb so langer Zahlen).

Komplexität:

- Bei einer Zerlegung in zwei halb so große Probleme, von denen nur *eines* gelöst werden muss (z. B. binäres Suchen), braucht man $\log(n)$ Aufrufe (das ist normalerweise sehr gut!).
- Bei einer Zerlegung in *zwei* halb so große Probleme braucht man $n * \log(n)$ Aufrufe (das ist normalerweise gut!).
- Teilt man *nicht in der Mitte*, braucht man bei zwei Teilproblemen schlimmstenfalls (bei Teilung in $n-1$ und 1, beispielsweise beim ungünstigsten Fall des Quicksort) in etwa n^2 Schritte (deutlich schlechter!).
- Bei *vier* Teilproblemen halber Größe braucht man n^2 Schritte[2], bei acht n^3.

Backtracking, Branch-and-Bound, Alpha-Beta-Search: "Probieren & rückgängig machen"

Backtracking ist eine Methode, um Such- und Optimierungsprobleme zu lösen, deren Lösung sich aus mehreren Einzelschritten oder Einzelentscheidungen (z. B. Spielzügen) zusammensetzt. Jede Ebene der rekursiven Aufrufe behandelt dabei einen solchen Einzelschritt der Gesamtlösung: Der äußerste Aufruf den ersten, der innerste den letzten.

Backtracking findet alle Lösungen durch systematisches Durchprobieren, verfolgt aber im Unterschied zum Durchprobieren aller Möglichkeiten sinnlose Teillösungen nicht weiter: Stehen beispielsweise beim 8-Damen-Problem schon die erste und zweite Dame auf Kriegsfuß, wird gar nicht mehr versucht, eine Position für die dritte zu finden: Alle Möglichkeiten, die mit dieser Position für Dame eins und zwei beginnen, werden daher gleich gar nicht berechnet.

Beispiele: 8-Damen-Problem, Rösslsprung, Weg im Labyrinth, Landkarten-Färbe-Problem, optimal Geld wechseln, optimal Rucksack packen, ...

Solche Probleme lassen sich als Baum darstellen, die Rekursion durchwandert den Baum komplett: Jeder Knoten des Baumes entspricht einem rekursiven Aufruf, die Wurzel dem äußersten. Jede Ebene des Baumes entspricht einer Ebene der Rekursion, jeder Sohn eines Knotens einer Zugmöglichkeit in diesem Knoten. Blätter auf untester Ebene sind Lösungen, Blätter darüber sind tote Zweige (ohne zulässige Möglichkeiten).

Allgemeine Vorgehensweise:

---

[2] Die rekursive Multiplikation bringt also gegenüber der normalen ziffernweisen Methode, die auch n^2 Einzelmultiplikationen braucht, nichts. Es gibt aber einen Trick, mit drei statt vier halb so langen Multiplikationen auszukommen, und dann braucht man nur mehr $n^{1.585}$ Einzelmultiplikationen.

Mache i-ten Schritt:
`for` *alle Möglichkeiten für i-ten Schritt*
`if` *Möglichkeit ist zulässig*
Speichere die Möglichkeit als i-ten Schritt in der Lösung
`if` *Ziel erreicht*
`then` *Drucke Lösung*
`else` *Mache (i+1)-ten Schritt*
Lösche i-ten Schritt wieder aus der Lösung

Komplexität: Gibt es x Möglichkeiten für jeden Einzelschritt und besteht die Lösung aus n Einzelschritten, so beträgt der Rechenaufwand im schlimmsten Fall x^n (also *sehr* viel!). Sehr viele der Möglichkeiten fallen zwar normalerweise von vornherein weg, aber der Aufwand bleibt expotentiell (1 Schritt mehr $\Longrightarrow$ doppelte, dreifache, ... Rechenzeit!).

Wenn man nur die erste Lösung sucht, kann man das Programm im ersten Aufruf von *Drucke Lösung* beenden.

Wenn man alle Lösungen berechnet, kann man natürlich auch die beste ermitteln: Man merkt sich in *Drucke Lösung* die beste bisher gefundene Lösung in einer globalen Variable und vergleicht jede neu gefundene damit. Nach der Rückkehr aus dem äußersten rekursiven Aufruf gibt man die gespeicherte Lösung aus.

Beispiele: Rucksack-Problem, Traveling Salesman, ...

Mit zwei Tricks lassen sich solche Verfahren wesentlich verbessern:

- Man betrachtet die Lösungsmöglichkeiten für jeden Einzelschritt möglichst in jener Reihenfolge, die die vermutlich guten Lösungen zuerst liefert.
- Branch-and-Bound: Man scheidet nicht nur unmögliche Zweige (z. B. "Rucksack übervoll") von vornherein aus, sondern verfolgt auch jene Teillösungen nicht weiter, die kein besseres Ergebnis als das bisher beste gefundene mehr liefern können (z. B. "Bester derzeitiger Rucksackwert ist mit derzeitigem Inhalt und noch verbleibenden Gegenständen nicht mehr erreichbar").

Folgendes "Kochrezept" gilt für die Lösung von Such- und Optimierungsproblemen mittels Backtrack:

1. *Was ist ein Schritt meiner Lösung?*

   Beim 8-Damen-Problem: Setze eine Dame
   Beim Labyrinth: Ziehe ein Feld
   Beim Rucksack packen: Entscheide über einen Gegenstand

   Jede Ebene des rekursiven Aufrufs macht genau *einen* Schritt. Die rekursive Funktion enthält daher ***nie*** eine Schleife über mehrere Schritte!

   Läßt sich die Lösungssuche nicht in Teilschritte zerlegen, ist Backtracking keine geeignete Lösungsmethode!

2. *Was sind die Möglichkeiten für einen Schritt?*

   Beim 8-Damen-Problem: Die 8 Spalten
   Beim Labyrinth: Die 4 Richtungen
   Beim Rucksack packen: Mitnehmen oder zu Hause lassen

   Pro Möglichkeit wird *ein* rekursiver Aufruf gemacht. Es gibt daher in der Funktion entweder nacheinander so viele rekursive Aufrufe wie Möglichkeiten (Beispiel: Labyrinth), oder

es steht *ein* rekursiver Aufruf innerhalb einer Schleife über alle Möglichkeiten (Beispiel: 8-Damen-Problem).

3. *Welche Möglichkeiten sind zulässig?*

   Beim 8-Damen-Problem: Keine andere Dame in gleicher Spalte und Diagonale
   Beim Labyrinth: Feld ist frei und noch nicht begangen
   Beim Rucksack packen: Zu Hause lassen geht immer, mitnehmen je nach Gewicht

   Diese Abfrage steht entweder *unmittelbar vor* dem rekursiven Aufruf (die Funktion wird nur für zulässige Möglichkeiten aufgerufen), oder gleich *am Beginn* der rekursiven Funktion (die Funktion wird immer aufgerufen, kehrt aber für unzulässige Möglichkeiten sofort wieder zurück).

4. *Was muß ich vor dem rekursiven Aufruf speichern und nachher eventuell wieder löschen?*

   Vor der Betrachtung der Möglichkeiten für den nächsten Schritt im rekursiven Aufruf muß die gewählte Möglichkeit für den aktuellen Schritt gespeichert werden (üblicherweise werden dazu globale Variablen verwendet), *nach* der rekursiven Betrachtung aller Möglichkeiten des nächsten Schrittes muß der aktuelle Schritt wieder rückgängig gemacht werden (falls er nicht durch die nächste Möglichkeit ohnehin sofort überschrieben wird).

5. *Welche Argumente und welchen Returnwert braucht die Funktion, und wie sieht der rekursive Aufruf aus?*

   Jeder rekursive Aufruf kümmert sich um die *restlichen* Schritte zum Ziel.

6. *Was ist das Ziel bzw. wann bin ich fertig?*

   Beim 8-Damen-Problem: Alle Damen gesetzt
   Beim Labyrinth: Zielpunkt erreicht
   Beim Rucksack packen: Über alle Gegenstände entschieden

   Diese Abfrage steht entweder *am Beginn* der rekursiven Funktion, bevor man die Möglichkeiten eines neuen Schrittes analysiert (man muß gar keinen Schritt mehr machen und gibt stattdessen die Lösung aus), oder *unmittelbar vor* dem rekursiven Aufruf (man ruft die Lösungsausgabe auf, anstatt rekursiv weiter zu suchen).

   Die Ausgabe oder Speicherung der Lösung ist der *nichtrekursive* Zweig der rekursiven Funktion.

7. *Zur Kontrolle: Komme ich dem Ziel mit jedem Aufruf näher?*

   Beim 8-Damen-Problem: Pro Aufruf eine Dame mehr gesetzt
   Beim Labyrinth: Pro Aufruf ein freies Feld weniger
   Beim Rucksack packen: Pro Aufruf ein zu entscheidender Gegenstand weniger

8. *Wie sieht der Anfangs-Aufruf für den ersten Schritt aus?*

Ein weiterer Sonderfall ist der Alpha-Beta-Algorithmus für Spiele, bei denen zwei Spieler abwechselnd ziehen, beispielsweise für Schachprogramme:

- Normalerweise begrenzt man hier die Rekursionstiefe, auch ohne eine komplette Lösung (Endposition des Spiels, d. h. Sieg, Patt oder Niederlage) gefunden zu haben, d. h. man rechnet maximal eine bestimmte Anzahl von Zügen voraus.
- Ist eine Endposition oder die maximale Rekursionstiefe erreicht, wird diese bewertet (z. B. mit einer Zahl zwischen -1 für Niederlage und 1 für Sieg).

- Die mittleren Ebenen der Rekursion berechnen ihren Wert für Einzelschritte, bei denen der Computer zieht, als Maximum aller Werte der rekursiven Aufrufe (weil er sich ja den Zug aussuchen wird, der ihm die besten Gewinn-Chancen verspricht), und als Minimum bei jenen Schritten, die der Gegner macht (weil der ja vermutlich den Zug wählt, der die schlechtesten Siegeschancen für den Computer bringt).
- Der oberste Aufruf wählt jene Zugmöglichkeit aus, deren rekursiver Aufruf den besten Wert geliefert hat.
- Die Alpha-Beta-Methode besteht wieder darin, Züge, die nichts am optimalen Ergebnis ändern, möglichst bald zu erkennen und nicht weiter zu verfolgen, um Geschwindigkeit zu gewinnen:
  * Hat die beste bisher gefundene Möglichkeit für einen Zug, den der Computer macht, den Wert α, und sinkt für eine andere Möglichkeit desselben Computer-Zugs das Minimum beim Probieren der Möglichkeiten für den darauffolgenden Zug des Gegners unter α, so braucht man diese Möglichkeit des Computer-Zugs nicht weiter analysieren: Sie ist sicher schlechter als die beste bisher gefundene.
  * Umgekehrt gilt das gleiche: Steht das aktuelle Minimum beim Analysieren der Zugmöglichkeiten eines Zuges des Gegners auf β, und liefert für eine bestimmte Zugmöglichkeit des Gegners irgendein darauffolgender Zug des Computers einen besseren Wert als β, so braucht man diese Zugmöglichkeit des Gegners nicht länger anschauen (weil sie der Gegner nicht wählen wird) und kann sofort die nächste Zugmöglichkeit des Gegners analysieren.

Andere rekursive Problemlösungen:

Ein klassischer rekursiver Algorithmus ist *"Towers of Hanoi"*:

Problem: Auf einem Stab stecken übereinander n Scheiben mit verschiedenem Durchmesser, sortiert nach Durchmesser (die kleinste oben). Diese sollen auf einen anderen Stab umgeschlichtet werden. Die Scheiben können dazu zwischendurch auf einem dritten Stab abgelegt werden. Es darf immer nur eine Scheibe bewegt werden, und es darf nie eine größere auf eine kleinere Scheibe gelegt werden.

Lösung:

- Transferiere die oberen $n-1$ Scheiben auf den dritten Stab. Verwende dabei den Zielstab als Zwischenablage.
- Bewege die jetzt frei liegende unterste Scheibe auf den ebenfalls freien Zielstab.
- Transferiere die oberen $n-1$ Scheiben vom dritten Stab auf den Zielstab. Verwende dabei den ursprünglichen Stab als Zwischenablage.

Oder in Pseudo-Code:

```
void move(int n, pile from, pile to, pile using) {
  if (n == 0) return;
  move(n - 1, from, using, to);
  move_one(from, to);
  move(n - 1, using, to, from);
}
```

Für n Scheiben braucht man $2^n - 1$ Züge (wie bei jeder Rekursion mit zwei Aufrufen der Größe $n-1$)!