II.3.1 Rekursive Algorithmen

Mittwoch, 23. November 2016 09:00

Bislang: Schleifen, um best. Programmabschnitte mehrfach auszuführen (benötigt oftmals Akkumulator-Variablen wie res im Bsp.). iterative Algorithmen

Jetzt: rekursive Algorithmen

Um ein Problem für x zu lösen, führen wir es zurück auf das Problem für einen Wert kleiner als x:

- falls $x > 1$, dann berechne erst die Fak. von $x-1$ und multipliziere das Resultat mit x
- falls $x \leq 1$, dann liefere 1 als Resultat.

Dh: $$fak(x) = \begin{cases} x * fak(x-1), & \text{falls } x > 1 \\ 1, & \text{sonst} \end{cases}$$

Rechenvorschrift greift auf sich selbst zurück: rekursiv.

Bei Rekursion: Problemlösung ist oft völlig analog zur Problemdefinition (=> deklarative

Programmierung)

Ausführg. im Bsp:

$fak(3) = 3 * fak(2)$

$= 3 * 2 * fak(1)$

$= 3 * 2 * 1$

$= 6$

Rekursive Argument muss jeweils "kleiner" werden, damit Rekursion terminiert.
(Vorteil: Verifikation ist meist viel einfacher, da man keine Schleifeninvariante braucht. Stattdessen: Induktion)

Klassifikation rekursiver Methoden:

- lineare / nicht-lineare Rekursion
- direkte / verschränkte Rekursion
- Endrekursion

Lineare Rekursion:
Pro Auswertung des Metho-

denrumpfs wird höchstens ein rekursiver Aufruf erreicht (Bsp. fak)

Bsp. für nicht-lineare Rekursion: Fibonacci-Methode

Fibonacci-Zahlen:

Versuch, Wachstum v. Kaninchen-Population zu beschreiben.

Annahmen:

- Kaninchen werden nach 1 Monat geschlechtsreif und trächtig.
- Jedes Kaninchenpaar bekommt ein männliches + ein weibliches Kind.
- Tragezeit ist 1 Monat
- Kaninchen sind unsterblich

Anzahl der Kaninchenpaare im Monat x =

Anzahl Kaninchenpaare im Monat $x-1$

+

Anzahl Kaninchenpaare im Monat x-2

Jeder Methode führt pro Auswertg. des Methodenrumpfs zu 2 rekursiven Aufrufen.

Dieser Algorithmus ist sehr ineffizient:

$$fib(20) = fib(19) + fib(18)$$
$$= fib(18) + fib(17) + fib(18)$$ ← fib(18) wird 2 mal berechnet
$$= fib(17) + fib(16) + fib(17) + fib(17) + fib(16)$$ ← fib(17) wird 3 mal berechnet
$$= \ldots$$

Exponentieller Aufwand (um fib(n) zu berechnen braucht man etwa 2^n Schritte).

Man kann fib auch rekursiv auf effizientere Weise programmieren (mit linearem Aufwand, so dass die Berechnung von fib(n) nur etwa n Schritte braucht).

Spezialfall der nicht-linearen Rekursion:

geschachtelte Rekursion (nested recursion)

```
public static int f (int x) {
    if (x<1) return 0;
    else     return f(f(x-1));
}
```

↑ terminiert und berechnet immer 0

Direkte/verschränkte Rekursion:

direkte Rekursion:
Methode f ruft selbst wieder f auf (Bsp. fak, fib)

verschränkte Rekursion:
System v. Funktionen, die sich gegenseitig aufrufen, z.B. f ruft g auf, g ruft f auf

Endrekursion (tail recursion)

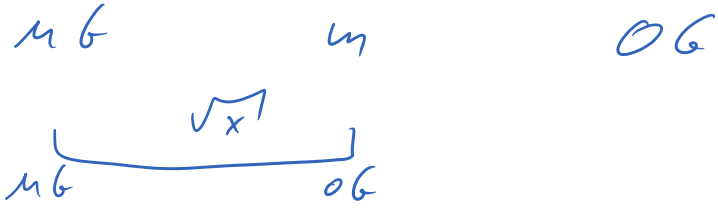
Spezialfall der direkten

Rekursion, bei der der rek. Aufruf nur am Ende des Algorithmus auftreten darf.
D.h.: rekursive Aufrufe dürfen nicht vor anderen Anweisungen oder in Teilausdrücken auftreten.
=> fak ist nicht endrekursiv, denn rek. Aufruf tritt in Teilausdruck auf.
Man muss sich den ursprünglichen Wert von x merken, da man ihn noch nach Ende des rek. Aufrufs braucht.

Bsp. für endrekursiven Alg: sqrt

Aufmf mit sqrt(0, x, x)
↑ UG ↑ OG

Intervallschachtelung:



3. Fall

In Schleifen: lokale Variablen werden in jedem Schleifendurchlauf überschrieben

Bei Rekursion: alte Werte der lokalen Variablen müssen nach dem rek. Aufruf noch zur Verfügung stehen.

Ausnahme ist Endrekursion: Da hier der rek. Aufruf erst zum Schluss kommt, braucht man alte Werte der lok. Variablen nicht mehr.

⇒ Endrekursion kann man direkt in Schleifen überführen

Speicherorganisation bei Rekursion

- Für jeden Methodenaufruf wird ein neuer Speicherrahmen (frame) auf dem Kellerspeicher (stack) angelegt.
- Darin werden alle lokalen Variablen gespeichert (inklusive der formalen Parameter und Variable für Rückgabewert bei nicht-void Methoden)
- Speicherrahmen ≙ Kontext, in dem Methode ausgeführt wird.
- Bei jedem rekursiven Aufruf kommt ein

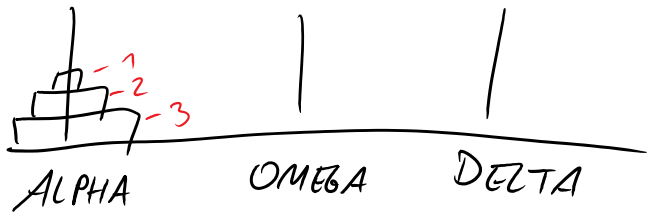
neuer Stack Frame (könnte zu Stack Overflow führen).

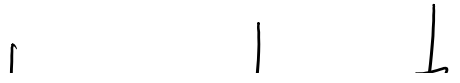
- Rekursion erlaubt aber oft sehr elegante + kurze Programme.

Bsp: Türme von Hanoi

Entwurfstechnik: Divide + Conquer
(Teile + Herrsche)

1. Behandle einfache Fälle (Türme der Höhe h=0)
2. Divide: Teile das Problem in 2 oder mehrere Teilprobleme auf (statt Turm der Höhe h, betrachte Turm der Höhe h-1 und 1)
3. Conquer: Löse Teilprobleme (typischerweise rekursiv)
4. Kombiniere: Setze Teillösungen zusammen

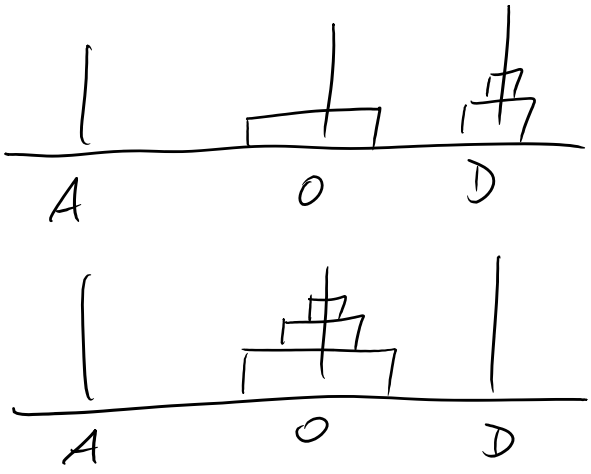




h=3 von ueber nach
bewegeTurm(h, AL, DEL, OM)

① Verschiebe Turm der Höhe h-1 von ALPHA nach DELTA
von nach ueber
bewegeTurm(h-1, AL, OM, DEL)

② Lege unterste Scheibe von ALPHA nach OMEGA



(2) Lege unterste Scheibe von ALPHA nach OMEGA

von nach

druckeZug(h, AL, OM)

(3) Verschiebe Turm der Höhe h-1 von DELTA nach OMEGA

ueber von nach

bewegeTurm(h-1, DEL, AL, OM)