II.3.1 Rekursive Algorithmen

Freitag, 9. November 2018 13:30

Bislang: iterative Programme, die mit Hilfe von Schleifen best. Prog.-Teile mehrfach durchlaufen. Benötigen oft Akkumulator-Variable (z.B. res), um schrittweise das Ergebnis zu berechnen.

Jetzt: rekursive Algorithmen z.B. führe das Problem für x auf das Problem für einen kleineren Wert als x zurück. Wird in Mathematik oft zur Definition benutzt:

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

fak ist rekursiv definiert, d.h. "fak" tritt in ihrer eigenen Def. auf, aber für einen kleineren Wert.

$$\begin{aligned} fak(3) &= 3 * fak(2) \\ &= 3 * 2 * fak(1) \\ &= 3 * 2 * 1 \end{aligned}$$

Die Beschreibung/Deklaration des Problems ergibt direkt das Programm, das das Problem

Lisp ⇒ deklaratives Programmieren.

(⇒ funkt. Programmieren, später)

Klassifikation rekursiver Methoden

- linear / nicht-linear:

  linear: höchstens 1 rekursiver Aufruf pro Methodenrumpf-Ausführung (z.B. fak)

  nicht-linear: Bsp fib

Fibonacci-Zahlen (13. Jh.)

Vorhersage der Entwicklung von Kaninchen-Populationen

Annahmen:

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

fib(x) = Anzahl der Kaninchenpaare im x-ten Monat.

fib(x) = 0 für x < 1

fib(x) = 1 bei x = 1

$fib(x) = \begin{cases} \text{bei } x \geq 2 \\ fib(x-1) + fib(x-2) \end{cases}$

Bsp hat nicht-lineare Rekursion und ist sehr ineffizient:

$fib(20) = fib(19) + fib(18)$
$= fib(18) + fib(17) + fib(18)$
$= fib(17) + fib(16) + fib(17) + fib(17) + fib(16)$

fib(18) wird 2-mal berechnet
fib(17) wird 3-mal ber.
fib(16) wird 5-mal ber.
fib(15) wird 8-mal ber.
⋮ ↑
Fibonacci-Zahlen

Dieser Algorithmus hat exponentiellen Aufwand (die Berechnung von fib(n) dauert etwa 2^n Schritte).
Man kann fib auch mit linearem Aufwand programmieren (d.h. Ber. von fib(n) braucht dann etwa n Schritte).

weitere Klassifikation d. Rekursion

- direkte / verschränkte Rekursion

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

verschränkte Rekursion:
Methoden, die sich gegenseitig aufrufen, z.B.

f ruft g auf, g ruft f auf.

- Endrekursion:
  Spezialfall der direkten Rekursion, bei dem rek. Aufrufe nur am Ende des Alg. auftreten
  (d.h: Keine Rekursion in Teilausdrücken, keine Rekursion vor weiteren Anweisungen der Methode)
  Bsp. fak ist nicht endrekursiv, da man nach dem rek. Aufruf das Ergebnis noch mit x multiplizieren muss. ⇒ x muss danach noch verfügbar sein.

Bsp für endrek. Alg:

Sqrt berechnet $\sqrt{x}$ durch Intervallschachtelung.

Aufruf mit sqrt(0, x, x).
(0 ↑ UG, x ↑ OG)

Setzt m auf die Mitte des Intervalls [ug, og].

Falls x in der unteren Hälfte liegt ⇒ rek. Aufruf mit [ug, m].

Sonst ⇒ rek. Aufruf mit [m, og].

In Schleifen:
lokale Variablen können in jedem Schleifendurchlauf überschrieben werden.

Bei Rekursion:
Methode wird mit neuen Argumenten aufgerufen (≙ Überschreiben der lokalen Variablen), aber nach Beendigung des rek. Aufrufs müssen die alten Werte der lokalen Var. noch zur Verfügung stehen.
Ausnahme: Endrekursion
⇒ endrekursive Methoden können direkt in iterative Methoden übersetzt werden.

Speicherorganisation bei Rekursion

- bei jedem Methodenaufruf wird auf Stack ein neuer Speicherrahmen angelegt
- Speicherrahmen (frame) enthält Speicherplatz für alle lokalen Variablen d. Methode (inklusive formale Parameter und Resultat bei nicht-void

Methoden)

- rek. Aufrufe führen gleichen Prog-text jeweils in neuem Kontext aus.
- Wenn bei zu vielen rek. Aufrufen der Stack nicht ausreicht ⇒ Stack Overflow

Nachteil d. Rekursion:
Verbraucht ggf. mehr Speicher

Vorteil d. Rekursion:
Programme lassen sich oft sehr klar + einfach formulieren.

Bsp: Türme von Hanoi

Entwurfstechnik:

Divide and Conquer
(Teile und herrsche)

1. Behandle einfache Fälle
(Bsp: Turm hat Höhe h=0.)

2. Divide: Teile nicht-einfache Fälle in 2 oder mehrere Teilprobleme auf.
(Bsp: Wenn h>0, dann löse die Teilprobleme für Türme der Höhe h-1 und Türme der Höhe 1)

3. Conquer: Löse die Teilprobleme

4. Kombiniere die Teillösungen zur Gesamtlösung.