II.3.1 Rekursive Algorithmen

Dienstag, 14. November 2017 09:30

Bisher: iterative Algorithmen
(d.h.: mehrfaches Durchlaufen v. Prog-Abschnitten in Schleifen)
Benötigt oft Akkumulator-Variablen (z.B. res).

Jetzt: rekursive Algorithmen
d.h.: führe das Problem für x zurück auf das Problem für einen kleineren Wert als x:

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

Methode fak ruft sich selbst (mit einem kleineren Argument) wieder auf.

- Bei Rekursion ist die Lösung oft vollkommen analog zur Problembeschreibung
(führt zur deklarativen Programmierung => fkt. + log. Prog.)

Bsp: fak(4)

= 4 * fak(3)

$= 4 * 3 * fak(2)$

$= 4 * 3 * 2 * fak(1)$

$= 4 * 3 * 2 * 1$

$= 24$

- Damit rek. Alg. terminieren muss das Arg. im rek. Aufruf "kleiner" werden.

Verschiedene Arten von Rekursion

- Lineare / nicht-lineare Rekursion

  lineare Rekursion: pro Methodenrumpf-Ausführung gibt es höchstens einen rek. Aufruf (Bsp: fak)

  Bsp. für nicht-lineare Rekursion: Fibonacci-Algorithmus.

Fibonacci-Zahlen zur Vorhersage v. Kaninchen-Populationen:

Annahmen:

- Kaninchen werden nach 1 Monat geschlechtsreif + trächtig.
- Tragezeit 1 Monat
- Jedes Kaninchenpaar bekommt

je ein männliches und ein weibliches Kind.

- Kaninchen sterben nicht

$fib(x)$: Anzahl der Kaninchenpaare im x-ten Monat

Dann gilt:

$$fib(x) = fib(x-1) + fib(x-2), \quad \text{für } x \geq 2$$

$$fib(1) = 1$$

Nicht-lineare Rekursion: Aufruf kann zu 2 rek. Aufrufen führen.

Aber: Algorithmus ist sehr ineffizient.

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

$fib(19)$ wird 1 mal ber.

$fib(18)$ wird 2 mal berechnet

$fib(17)$ wird 3 mal berechnet

$fib(16)$ wird 5 mal berechnet

$fib(15)$ wird 8 mal ber.

Weitere Klassifikation v. Rekursion:

- direkte / verschränkte Rekursion

direkt: Methode f ruft wieder f auf

(Bsp: fak, fib)

verschränkt: z.B. Methode f ruft g auf, g ruft f auf.

Bsp: even + odd

- Weitere Klassifikation

Endrekursion (tail recursion): Rekursion darf nur am Ende des Alg. auftreten, d.h. nicht in Teilausdrücken u. nicht vor weiteren Anweisungen des Alg.

Bsp: fak ist nicht endrekursiv, denn nach dem rek. Aufruf findet noch eine Multiplikation statt (rek. Aufruf tritt in Teilausdruck auf)

Hingegen: sqrt ist endrekursiv

Ist die rekursive Version d. Alg, den wir schon iterativ betrachtet hatten.

Ziel: Berechne $\sqrt{x}$
durch Intervallschachtelung.

- Starte mit $[u6, o6]$.
- Setze m auf die Mitte
- Rek. Aufruf mit
  $[u6, m]$ oder
  $[m, o6]$.

- Endrekursion: Nach dem rekursiven Aufruf werden die alten Werte der lokalen Variablen nicht mehr benötigt.
  ⇒ endrekursive Methoden lassen leicht in Schleifen umformen.
- Im allgemeinen ist diese Übersetzung nicht möglich, da man nach Beendigung des rek. Aufrufs noch die Werte der Variablen brauchen könnte, die sie vor dem rek. Aufruf hatten.
  ⇒ Speicherorganisation bei Rekursion
- Speicherrahmen für Methodenaufruf enthält Speicher für jede lokale Variable (inklusive der formalen Parameter und des Resultats)

(Potentielle Gefahr des Überlaufs des Kellerspeichers - stack overflow).

Vorteil v. Rekursion: erlaubt oftmals eine sehr kurze + elegante Programmierung

Bsp: Türme von Hanoi

Technik zum Entwurf des Programms: Divide + Conquer
(Teile + Herrsche)

1. Behandle einfache Fälle (z.B. Turm der Höhe h=0)
2. Divide: Teile das Problem in 2 oder mehr Teilprobleme auf (z.B. in die Probleme je einen Turm der Höhe h-1 und der Höhe 1 zu verschieben).
3. Conquer: Löse die Teilprobleme (typischerweise rekursiv)
4. Kombiniere: Setze Teillösungen zur Gesamtlösung zusammen.