11. Rekursion, Komplexität von Algorithmen

Teil 1

Java-Beispiele:

Power1.java
Hanoi.java

Schwerpunkte

- Wiederholung von Anweisungen:
  durch Iteration und Rekursion
- Anwendungsfälle der Rekursion
- Rekursiv definierte Funktionen
- Rekursive Problemlösungen
- Verarbeitung rekursiver Daten
- Überführung: Rekursion → Iteration
- Vor- und Nachteile der Rekursion
- Komplexitätstheorie:
  Komplexität von Algorithmen

Überblick

Wiederholung von Abarbeitungsschritten

Iteration: Zyklen **(while, for ...)**

1 * 2 * 3 * 4 * ... * n

Rekursion:

- Problemlösung durch "Selbstanwendung“
- Funktion wird durch sich selbst definiert

Eine Methode heißt r e k u r s i v :
ruft sich (direkt oder indirekt) selbst auf

```
static int power (int k, int n) {
   if (n == 0)
        return 1;
   else
        return k * power (k , n - 1);
}
```

direkt

```
int f1 (int n) {
   ... f2 (n-2) ..
}
int f2 (int n) {
   ... f1 (n-3)
}
```

indirekt

Anwendung der Rekursion

- Rekursiv definierte Funktionen
  - **Fibonacci-Funktion**
  - **Fakultät, Potenz**
  - **...**
- Natürliche rekursive Problemlösungen
  - **Sortierverfahren**
  - **Türme von Hanoi**
  - **...**
- Rekursiver Aufbau der zu verarbeitenden Daten
  - **Programme (EBNF)-> Compiler**
  - **Bäume und Listen**

Rekursiv definierte Funktionen

Rekursiv definierte Funktionen (1)

Beispiel: Fakultät

fakultät(n) = 1 * 2 ... * n

Anfang: fakultät(1) = 1

Schritt: fakultät(n + 1) = (n + 1) * fakultät(n)

Elementarer Fall

Allgemeines Problem **zurückgeführt auf** **einfacheres Problem**

derselben Problemklasse

Beispiel: Potenz

pot(k , n) = k * k * ... * k

Anfang: pot (k , 0) = 1

Schritt: pot (k , n + 1) = k * pot (k , n)

Rekursiv definierte Funktionen (2)

Beispiel: Summenformel

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

Anfang: sum(1) = 1

Schritt: sum(n + 1) = sum(n) + n + 1

Beispiel: Fibonacci - Funktion

Anfang: fib(0) = 1 fib(1) = 1

Schritt: fib(n + 1) = fib(n) + fib(n - 1)

Rekursive Methode: power()

```
static int power (int k, int n) {
   if (n == 0)
      return 1;
   else
      return k * power (k, n - 1);
}
```

→ Wie in iterativer Lösung: Anzahl Multiplikationen = n

→ Effektivere Lösung (später): Anzahl ca. $\log_2 n$

Aufruf von power()

```
static int power (int k, int n) {
   if (n == 0)
        return 1;
   else
        return k * power (k , n - 1);
}
```

```
                                             81
power ( 3 , 4 )
    3 * power ( 3 , 3 )
        3 * power ( 3 , 2 )                    27
            3 * power ( 3 , 1 )                  9
                3 * power ( 3 , 0 )                3
                    Anfang: = 1                    1
```

- 4 Multiplikationen
- 5 Methodenaufrufe gleichzeitig aktiv (Zeit + Speicher)
  ➔ **Compilerbau:** vor jedem Aufruf muss Kontext (Aufrufposition) zwischengespeichert werden, vgl. 2 hoch 1 Mio

Effizientere Implementation von power()?

pot(k , n) = k * k * ... * k

Anfang: pot (k , 0) = 1

Schritt: pot (k , n + 1) = k * pot (k , n)

pot(2, 1000) = 2 * 2 * 2 * 2 ... * 2

Statt 1000 Multiplikationen nur (ca.) 10?

Effizientere Implementation von power()

$k^n = (k^{n/2})^2$;falls n gerade

$k^n = (k^{(n-1)/2})^2 * k$;sonst

In $3^8 * 3^8$ wird 3^8 nur einmal berechnet

Beispiel:

$3^{16} = (3^8)^2 = 3^8 * 3^8$
$3^8 = (3^4)^2 = 3^4 * 3^4$
$3^4 = (3^2)^2 = 3^2 * 3^2$
$3^2 = 3 * 3$

Anzahl der Multiplikationen:

$\log_2 16 = 4$

nicht 16

Beispiel:

$3^{15} = (3^7)^2 * 3 = 3^7 * 3^7 * 3$
$3^7 = (3^3)^2 * 3 = 3^3 * 3^3 * 3$
$3^3 = (3^1)^2 * 3 = 3^1 * 3^1 * 3$

Anzahl der Multiplikationen: 6

Effizientere Implementation von power()

$k^n = (k^{n/2})^2$;falls n gerade

$k^n = (k^{(n-1)/2})^2 * k$;sonst

Power1.java

```
static int power1 (int k, int n) {

   if (n == 0)
       return 1;
   else {
       int t = power1(k, n/2);

       if ((n % 2) == 0)
          return t * t;
       else
          return k * t * t;
   }
}
```

Obige Formel wird auch für ungerade Zahlen umgesetzt – Warum?

n/2 ist ganzzahlige Division, z.B.
8/2 = 4
7/2 = 3

power() oder power1()?

```
static int power (int k, int n) {
   if (n == 0)
        return 1;
   else
        return k * power (k , n - 1);
}
```

```
static int power1 (int k, int n) {

   if (n == 0)
        return 1;
   else {
        int t = power1(k, n/2);

        if ((n % 2) == 0)
           return t * t;
        else
           return k * t * t;
   }
}
```

Lesbare Programme oder Effizienz durch trickreiche Algorithmen?

Effizienzvergleich: Anzahl Multiplikationen für k^n

→ power: n

→ power1: ca. $\log_2 n + 2$

(exakt: falls n Zweierpotenz)

+2, da 3^1 zu weiteren Aufrufen $3^0 * 3^0$ führt

d.h. völlig andere Komplexitätsklassen

Beispiele:

n	8	1024	1023	1025	999999
power	8	1024	1023	1025	999999
power1	5	12	20	13	32

z.B. int k=2, n=1024 → berechneter Wert von k^n ?

Anm: bei n = 1024, k > 1 Überlauf (max n = 30)
int: Werte bis maximal 2^{31}-1 → Klasse BigInteger nutzen