Informatik I WS 07/08

Tutorium 24

07.02.08

Bastian Molkenthin

E-Mail: infotut@sunshine2k.de

Web: http://infotut.sunshine2k.de

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Rekursion

- Rekursion = Aufrufen einer Methode durch sich selbst
- Wichtig: Abbruchbedingung die immer erreicht wird, sonst Endlosschleife!

```
void doSomething (int x) {
    doSomething(x-1);
}
```

- Jedes rekursive Problem kann auch iterativ gelöst werden
- Vorteile Rekursion:
  - Probleme oft elegant formulierbar und einfach zu implementieren
- Nachteile Rekursion:
  - Langsamer als iterative Variante
  - brauchen mehr Speicher

Aufgabe Rekursion

Schreibe 2 Methoden, die 2^n (n >= 0) berechnet. Einmal als iterative und einmal als rekursive Methode! Ohne Verwendung von pow! ($2^n = 2 * 2^{n-1}$ und $2^0 = 1$)

Rekursiv:

```
public static int exp2recursive(int n) {
    if (n == 0) return 1;
    else
        return exp2recursive(n-1)*2;
}
```

Iterativ:

```
public static int exp2(int n) {
    int result = 1;
    while (n >= 1) {
        result *= 2;
        n--;
    }
    return result;
}
```

Aufgabe Rekursion (2)

Was berechnet folgende Funktion?

```
public static int foo(int a, int b)
{
    if (b == 0) return a;
    else
        return foo(b, a % b);
}
```

Berechnet den größten gemeinsamen Teiler von a und b

Gib die Methode *foo* iterativ an!

```
public static int gcd(int a, int b)
{
    int rest = 0;
    while (b != 0) {
        rest = a % b;
        a = b;
        b = rest;
    }
    return a;
}
```

Schleifeninvariante

- Eine Schleifeninvariante ist eine Bedingung, die in jedem Schleifendurchlauf an einer bestimmten Position gültig ist

- Hilft bei der Verifizierung von Programmen ("Tut das Programm wirklich das, was es soll?")

- Um eine gegebene Schleifeninvariante zu beweisen muss man zeigen, dass die Bedingung an einer bestimmten Stelle in **jedem** Schleifendurchlauf gilt

- Anwendung der vollständigen Induktion
  (⇨ hier Induktion über die Anzahl der Schleifendurchläufe)

Schleifeninvariante - Beispiel

```
public static int myexp(int n)
{
    int r = 1;
    int i = 0;
    while (i < n) {        /*1*/
        r *= 2;            /*2*/
        i++;               /*3*/
    }
    return r;
}
```

Zeige durch geeignete Zusicherungen an den Stellen /*1*/, /*2*/ und /*3*/ dass $r = 2^i$ eine Schleifeninvariante für die while-Schleife ist. (n >= 1)!

- **Induktionsanfang:**

  Zu Beginn des 1. Schleifendurchlaufs gilt:

  $r_1 = 1 = 2^{i1}$ und $i_1 = 0$ ✓

- **Induktionsannahme:**

  Beim n-ten Durchlauf gelte bei Schleifenbeginn (/*1*/): $r_n = 2^i$

Schleifeninvariante - Beispiel

- **Induktionsschluss:** n ➡ n +1

Es gelten folgende Zusicherungen:

/*1*/ : $r_n = 2^{i_n}$

/*2*/ : $r_{n+1} = r_n * 2$

/*3*/ : $i_{n+1} = i_n + 1$

```
public static int myexp(int n)
{
    int r = 1;
    int i = 1;
    while (i <= n) {        /*1*/
        r *= 2;             /*2*/
        i++;                /*3*/
    }
    return r;
}
```

Also gilt: $r_{n+1} = r_n * 2 = 2^{i_n} * 2 = 2^{i_n+1} = 2^{i_{n+1}}$

Die Schleifeninvariante gilt also am Ende des n-ten Durchlauf und damit auch am Anfang des n + 1-ten Durchlauf. Durch die Induktion ist nun gezeigt, dass $r = 2^i$ am Anfang und am Ende jedes Schleifendurchlaufes gilt.
Damit ist bewiesen, dass es eine Schleifeninvariante ist.

Frage: Terminiert die while-Schleife?

O-Kalkül

- Definition?

$$\boldsymbol{O}\big(\boldsymbol{g}(\boldsymbol{n})\big) = \{\, \boldsymbol{f}(\boldsymbol{n}) \mid \exists\, \boldsymbol{n_0} \in \mathbb{N}, \exists\, \boldsymbol{c} \in \mathbb{R}\, \forall\, \boldsymbol{n} \geq \boldsymbol{n_0} : \boldsymbol{0} \leq \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c} * \boldsymbol{g}(\boldsymbol{n}) \}$$

- Also…
  - O(g(n)) enthält alle Funktionen, die asymptotisch höchstens so schnell wachsen wie g(n).
  - t ∈ O(f) : t wächst höchstens so schnell wie f.

- Wir kümmern uns nicht…

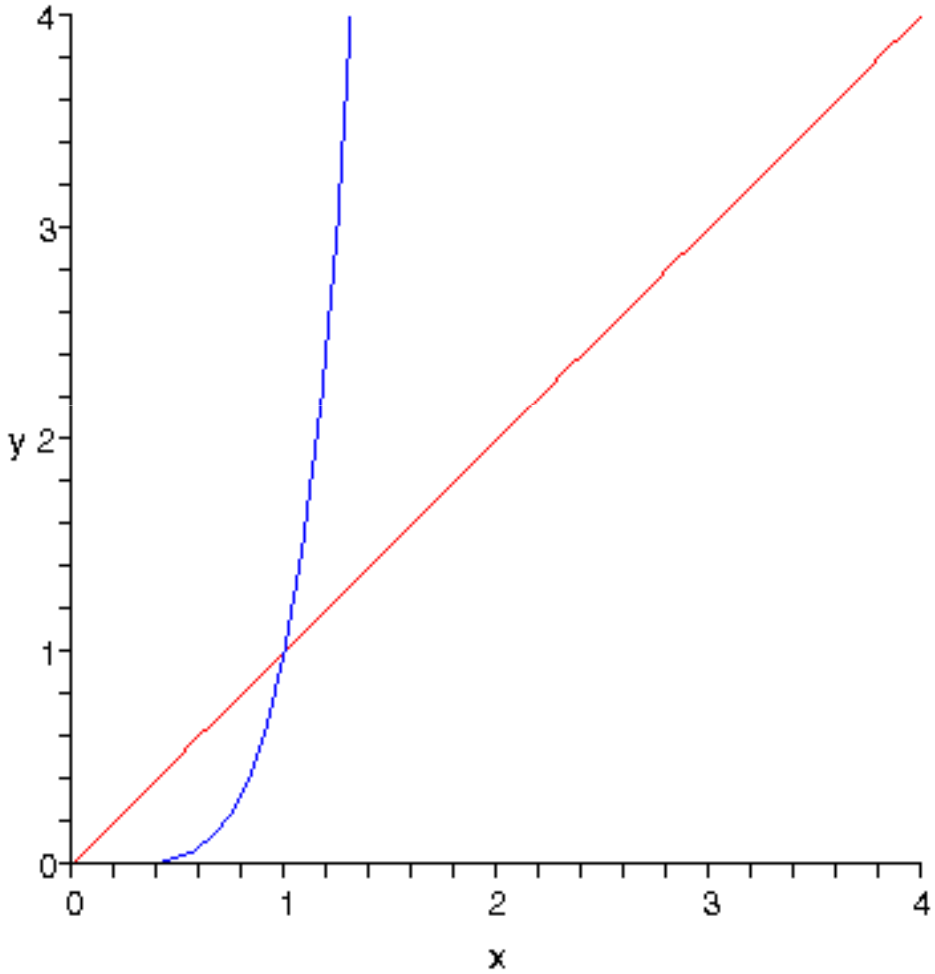
  … was am Anfang passiert : $\exists\, \boldsymbol{n_0} \in \mathbb{N}, \forall\, \boldsymbol{n} \geq \boldsymbol{n_0}$

  … um konstante Faktoren: $\exists\, \boldsymbol{c} \in \mathbb{R} \ldots\, \boldsymbol{c} * \boldsymbol{g}(\boldsymbol{n})$

Komplexitätsklassen

Beispiel	Komplexität
$t \in \mathcal{O}(1)$	Konstant
$t \in \mathcal{O}(\log n)$	Logarithmisch
$t \in \mathcal{O}(\sqrt{n})$	Wurzelfunktion
$t \in \mathcal{O}(n)$	Linear
$t \in \mathcal{O}(n \log n)$	n · Logarithmisch
$t \in \mathcal{O}(n^2)$	Quadratisch
$t \in \mathcal{O}(c^n)$	Exponentiell
$t \in \mathcal{O}(n!)$	Fakultät
$t \in \mathcal{O}(n^n)$	Superexponentiell
$t \in \mathcal{O}(n^{n^{n\cdots}})$	Hyperexponentiell

O-Kalkül



O-Kalkül - Aufgabe

Setzen Sie ein Kreuz, falls der betreffende Algorithmus die jeweilige Aufwands-Komplexität besitzt. Setzen Sie ein o, falls das nicht der Fall ist.

	$O(n^{5/2})$	$O(n*\log(n))$	$O(\frac{1}{2}\ n)$	$O(\log(n))$
Binäre Suche	x	x	x	x
Quicksort	x	o	o	o
Floyd-Warshall	o	o	o	o
4 + 7	x	x	x	x

Bemerkungen:

- Binäre Suche hat einen Aufwand von O(log(n)).
- Quicksort: Average case ist O(nlogn), aber worst case $O(n^2)$.
- Floyd-Warshall:

```
static void warshall(boolean[][] s) {
    for(int i = 0 ; i < s.length; i ++)
        s[i][i] = true;
    for(int k = 0 ; k < s.length; k ++)
        for(int i = 0 ; i < s.length; i++)
            for(int j = 0 ; j < s[0].length ; j++)
                s[i][j] = s[i][j] || (s[i][k] && s[k][j]);}
```

$O(n^3)$

Diese Aufgabe ist der inoffiziellen Informatik-I Probeklausur 2008 der Universität Karlsruhe entnommen.

Wissenfragen

EBNF ..

- … hilft Syntax zu formalisieren
- … Regeln werden mit einem Punkt abgeschlossen
- … Nichtterminalzeichen werden in Anführungszeichen gesetzt

Der von-Neumansche Flaschenhals …

- ... resultiert aus physikalischen Gegebenheiten
- ... spielt bei modernen Rechner keine Rolle mehr
- ... kann nur abgeschwächt und nicht gelöst werden

Ein ungerichteter Graph …

- ... lässt sich leicht in einen gerichteten Graphen umformen
- ... hat nur Knoten mit Eingangsgrad = Ausgangsgrad.

Eine Halbgruppe...

- ... ist abgeschlossen
- ... ist assoziativ
- ... besitzt ein Neutrales Element

Wissenfragen

EBNF ..

- … hilft Syntax zu formalisieren ✓
- … Regeln werden mit einem Punkt abgeschlossen ✓
- … Nichtterminalzeichen werden in Anführungszeichen gesetzt X

Der von-Neumansche Flaschenhals …

- ... resultiert aus physikalischen Gegebenheiten ✓
- ... spielt bei modernen Rechner keine Rolle mehr X
- ... kann nur abgeschwächt und nicht gelöst werden ✓

Ein ungerichteter Graph …

- ... lässt sich leicht in einen gerichteten Graphen umformen ✓
- ... hat nur Knoten mit Eingangsgrad = Ausgangsgrad. ✓

Eine Halbgruppe...

- ... ist abgeschlossen ✓
- ... ist assoziativ ✓
- ... besitzt ein Neutrales Element X

Das wars für heute…

Fragen ???

Viel Spaß mit dem Übungsblatt!