IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(02 – Funktionenklassen)

Prof. Dr. Susanne Albers

Beschreibung und Analyse von Algorithmen

Mathematisches Instrumentarium zur Messung der Komplexität (des Zeit- und Platzbedarfs von Algorithmen):

Groß-O-Kalkül (Landausche Symbole)

Primitive Operationen

- grundlegende Berechnungen, die von einem Algorithmus ausgeführt werden
- ablesbar aus Pseudocode oder Programmstück
- überwiegend unabhängig von einer (imperativen) Programmiersprache
- exakte Definition ist nicht bedeutend
- Beispiele
  - einen Ausdruck auswerten
  - einer Variablen einen Wert zuweisen
  - Indexierung in einem Array
  - Aufrufen einer Methode
  - Verlassen einer Methode

Funktionenklassen

Groß-O-Notation:

Mit O-,Ω - und Θ-Notation sollen obere, untere bzw. genaue Schranken für das Wachstum von Funktionen beschrieben werden.

Idee:

Konstante Faktoren und Summanden dürfen bei der Aufwandsbestimmung vernachlässigt werden.

Gründe:

- man ist an asymptotischem Verhalten für große Eingaben interessiert
- genaue Analyse kann technisch oft sehr aufwändig oder unmöglich sein
- lineare Beschleunigungen sind ohnehin immer möglich (Ersatz von Hardware/Software)

Ziel: Komplexitätsmessungen mit Hilfe von Funktionenklassen. Etwa $O(f)$ sind die Funktionen, die (höchstens) in der Größenordnung von *f* sind.

Die O-Notation

Sei eine Funktion $g : N \rightarrow R^+$ gegeben, dann ist $O(g)$ folgende Menge von Funktionen:

$$O(g) = \{ f : N \rightarrow R^+ \mid \text{es gibt positive Konstanten } c_1, n_0 \text{ mit } 0 \leq f(n) \leq c_1 g(n) \text{ für alle } n \geq n_0 \}$$

Die O -Notation gibt also eine asymptotisch obere Grenze für eine Funktion.

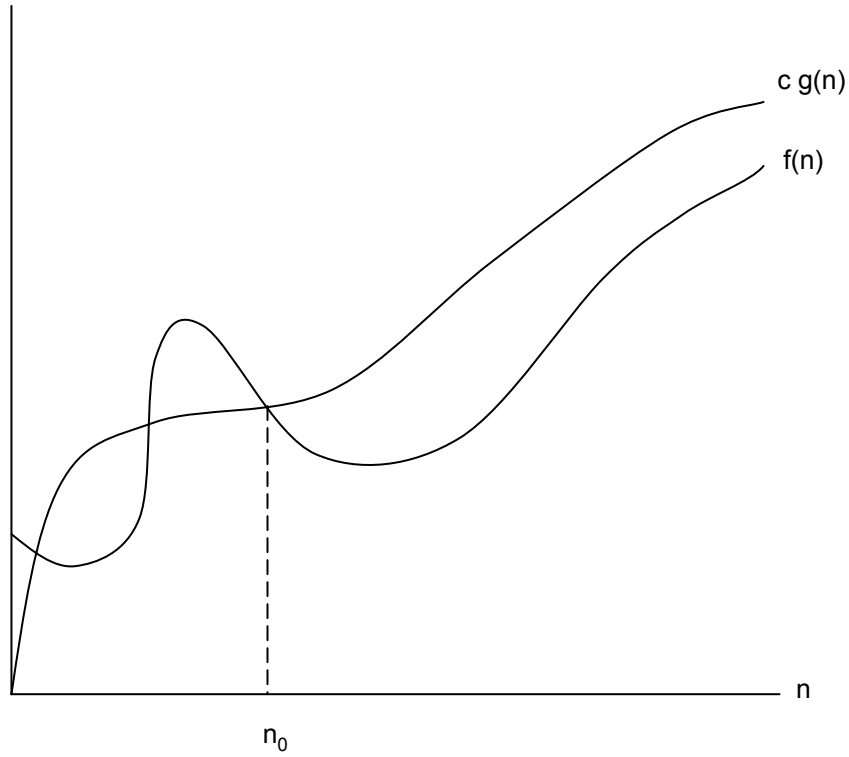
Wenn f zu $O(g)$ gehört, sagt man

f ist Gross – O von g

und schreibt $f(n) \in O(g(n))$ oder auch (mathematisch fragwürdig)
$f(n) = O(g(n))$

Veranschaulichung der O-Notation

Die Funktion f gehört zur Menge $O(g)$, wenn es positive Konstanten c und n_0 gibt, so dass $f(n)$ ab n_0 unter $cg(n)$ liegt.



Die Ω -Notation

Sei eine Funktion $g : N \rightarrow R^+$ gegeben, dann ist $\Omega(g)$ folgende Menge von Funktionen:

$$\Omega(g) = \{ f : N \rightarrow R^+ \mid \text{es gibt positive Konstanten } c_1, n_0 \text{ mit } 0 \leq c_1 g(n) \leq f(n) \text{ für alle } n \geq n_0 \}$$

Die Ω -Notation gibt also eine asymptotisch untere Grenze für eine Funktion.

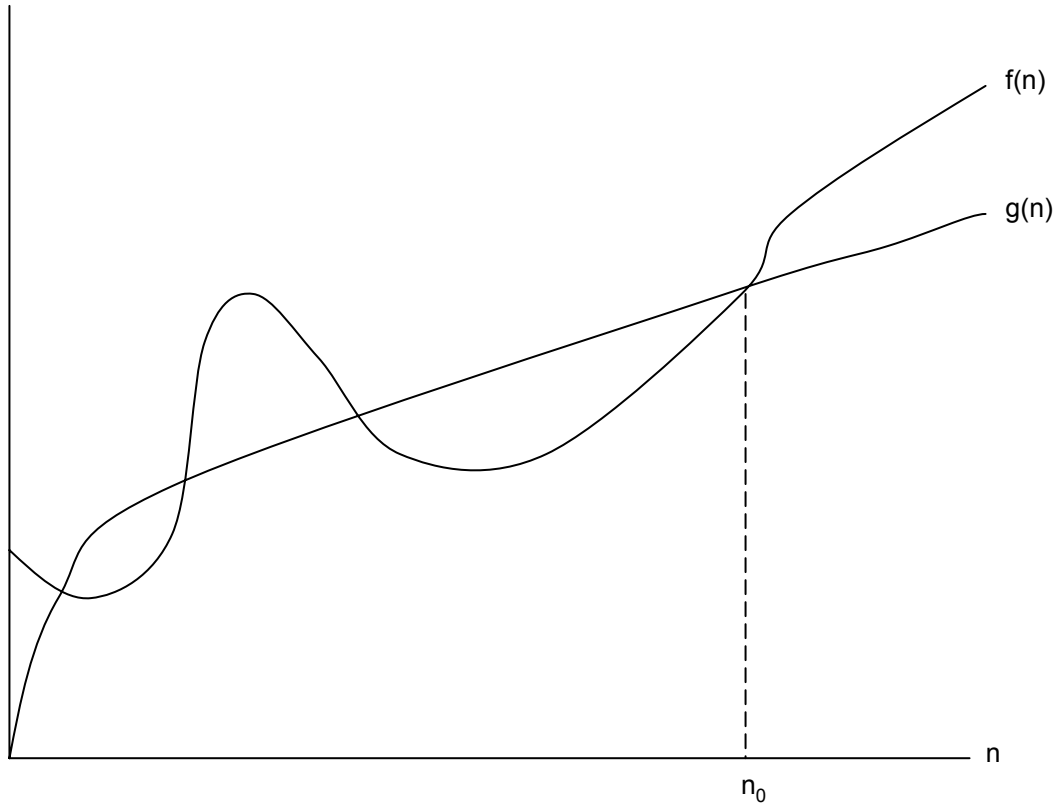
Wenn f zu $\Omega(g)$ gehört, sagt man

f ist Gross-Omega von g

und schreibt $f(n) \in \Omega(g(n))$ oder auch (mathematisch fragwürdig)
$f(n) = \Omega(g(n))$

Veranschaulichung der Ω -Notation

Die Funktion f gehört zur Menge $\Omega(g)$, wenn es positive Konstanten c und n_0 gibt, so dass $f(n)$ ab n_0 oberhalb $cg(n)$ liegt.



Die Θ -Notation

Sei eine Funktion $g : N \rightarrow R^+$ gegeben, dann ist $\Theta(g)$ folgende Menge von Funktionen:

$$\Theta(g) = \{ f : N \rightarrow R^+ \mid \text{es gibt positive Konstanten } c_1, c_2, n_0 \text{ mit}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ für alle } n \geq n_0 \}$$

Die Θ -Notation gibt also eine asymptotisch feste Grenze für eine Funktion.

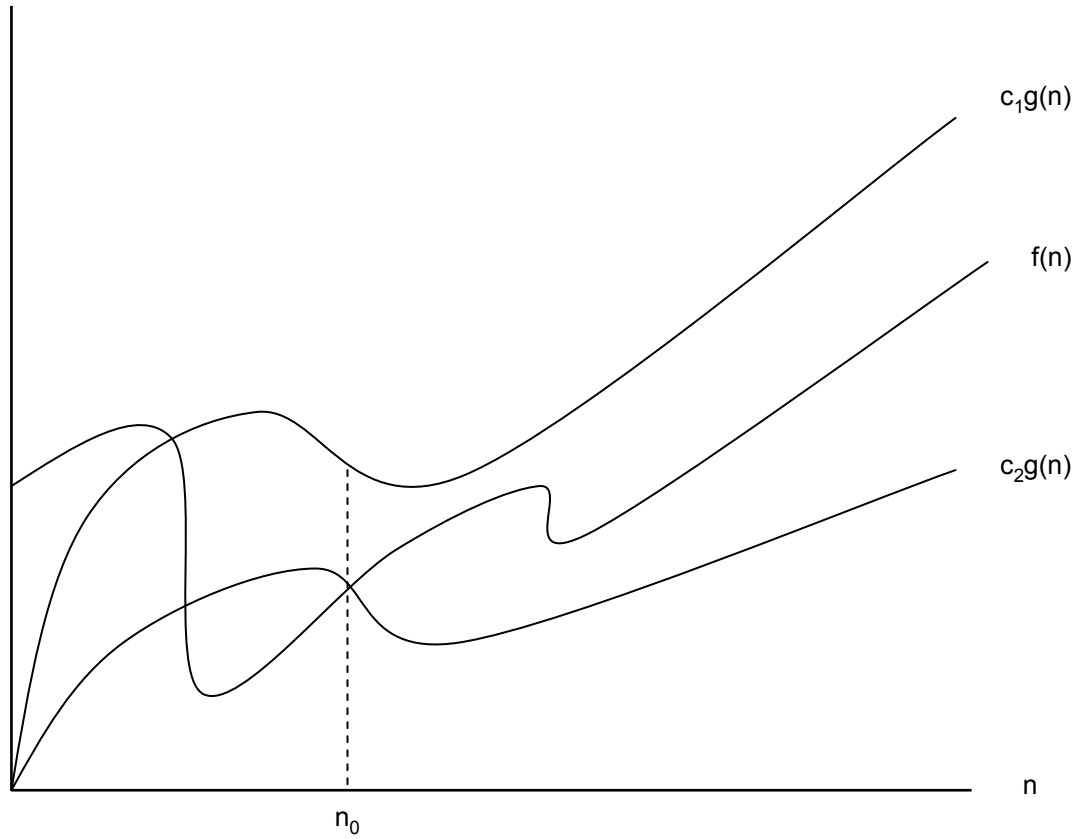
Wenn f zu $\Theta(g)$ gehört, sagt man

f ist Gross-Theta von g

und schreibt $f(n) \in \Theta(g(n))$ oder auch (mathematisch fragwürdig)
$f(n) = \Theta(g(n))$

Veranschaulichung der Θ -Notation

Die Funktion f gehört zur Menge $\Theta(g)$, wenn es positive Konstanten c_1, c_2 und n_0 gibt, so dass $f(n)$ ab n_0 zwischen $c_1 g(n)$ und $c_2 g(n)$ eingepackt werden kann.



Bemerkungen zu den O-Notationen

In manchen Quellen findet man leicht abweichende Definitionen, etwa

$$\mathrm{O}(g) = \{ f : N \to R^{+} \mid \text{es gibt positive Konstanten } a \text{ und } b \text{ mit } f(n) \leq ag(n) + b \text{ für alle } n \}$$

Für die relevantesten Funktionen f (etwa die monoton steigenden f nicht kongruent 0) sind diese Definitionen äquivalent.

Beispiele zur O-Notationen

$f(n) = 2n^2 + 3n + 1 = O(n^2)$

$f(n) = n \log n$ **ist nicht in** $O(n)$

Eigenschaften von Groß-O

- Transitivität: Ist $f(n) = O(g(n))$ und $g(n) = O(h(n))$, dann ist $f(n) = O(h(n))$
- Additivität: Ist $f(n) = O(h(n))$ und $g(n) = O(h(n))$, dann ist $f(n) + g(n) = O(h(n))$
- Für konstantes a ist: $a\, n^k = O(n^k)$
- $n^k = O(n^{k+j})$, für jedes positive j.
- $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n^1 + a_0 = O(n^k)$
- Ist $f(n) = c\, g(n)$ für konstantes c, so ist $f(n) = O(g(n))$
- Für beliebige positive Zahlen $a, b \neq 1$ ist $f(n) = \log_a n = O(\log_b n)$
- Für beliebige positive $a \neq 1$ ist $\log_a n = O(\log_2 n)$

Der O-Kalkül

Einfache Regeln:

$$f = O(f)$$
$$O(O(f)) = O(f)$$
$$k\ O(f) = O(f) \text{ für konstantes } k$$
$$O(f + k) = O(f) \text{ für konstantes } k$$

Additionsregel:

$$O(f) + O(g) = O(\max(f, g))$$

Multiplikationsregel:

$$O(f) * O(g) = O(f * g)$$

Additionsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile hintereinander ausgeführt werden.

Multiplikationsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile ineinander geschachtelt werden.

Beispiele zur Laufzeitabschätzung(1)

Algorithmus *prefixAverages1(X)*

Eingabe: Ein Array X von n Zahlen

Ausgabe: Ein Array A von Zahlen, so dass gilt: A[i] ist das arithmetische Mittel der Zahlen X[0], …, X[i]

Methode:

```
for i = 0 to n-1 do
        a = 0;
        for j = 0 to i do
                a = a + X[j]
        A[i] = a/(i + 1)
return array A
```

Beispiele zur Laufzeitabschätzung(2)

Algorithmus *prefixAverages2(X)*

Eingabe: Ein Array X von n Zahlen

Ausgabe: Ein Array A von Zahlen, so dass gilt: A[i] ist das arithmetische Mittel der Zahlen X[0], …, X[i]

Methode:

```
s = 0;
for i = 0 to n-1 do
        s = s + X[i];
        A[i] = s/(i + 1)
return array A
```

Hierarchie von Größenordnungen

Größenordnung	Name	
$O(1)$	konstante Funktionen	
$O(\log n)$	logarithmische Funktionen	
$O(\log^2 n)$	quadratisch logarithmische Funktionen	
$O(n)$	lineare Funktionen	
$O(n \log n)$	$n \log n$-wachsende Funktionen	
$O(n^2)$	quadratische Funktionen	
$O(n^3)$	kubische Funktionen	
$O(n^k)$	polynomielle Funktionen	k konstant

f heißt polynomiell beschränkt ,wenn es ein Polynom p mit $f = O(p)$ gibt.

f wächst exponentiell , wenn es ein $\varepsilon > 0$ gibt mit $f = \Theta(2^{n^\varepsilon})$.

Skalierbarkeiten

Annahme: 1 Rechenschritt $\cong 0.001$ Sekunden. Maximale Eingabelänge bei gegebener Rechenzeit.

Laufzeit T(n)	1 Sekunde	1 Minute	1 Stunde
n	1000	60000	3600000
$n \log n$	140	4895	204094
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Annahme: Es kann auf einen 10-fach schnelleren Rechner gewechselt werden. Statt eines Problems der Größe p kann in gleicher Zeit dann berechnet werden:

Laufzeit T(n)	neue Problemgröße
n	$10p$
$n \log n$	(fast 10)p $\approx 10 \frac{\ln(p)}{LambertW(10p \ln(p))} p$
n^2	$3.16p$ $\approx \sqrt{10}p$
n^3	$2.15p$ $\approx \sqrt[3]{10}p$
2^n	$3.32 + p$ $\approx \log 10 + p$

Bestimmung des Zeitaufwands

Sei A ein Programmstück, dessen Zeitaufwand cost(A) zu bestimmen ist:

1. A ist einfache Anweisung (read, write, +, -, . . .):
   $\text{cost}(\text{A}) = \text{const} \in \text{O}(1)$

2. A ist Folge von Anweisungen: Additionsregel anwenden

3. A ist if-Anweisung:
   (a) if (cond) B; cost(A) = cost(cond) + cost(B)
   (b) if (cond) B; else C; cost(A) = cost(cond) + max(cost(B), cost(C))

4. A ist eine Schleife (while, for, . . .):

   $$\text{cost}(\text{A}) = \sum_{\text{Umlauf } i} \text{cost}(\text{Anweisungen } i) + \text{cost}(\text{Terminierungsbedingung } i)$$

   oft einfach

   $$\text{cost}(\text{A}) = \#\,\text{Umläufe} * (\text{cost}(\text{Anweisungen}) + \text{cost}(\text{Terminierungsbedingung}))$$

5. A ist Rekursion . . .

Implementation in Java

```
class Mult {

    public static void main ( String [] args ) {
        int x = new Integer (args[0]). intValue();
        int y = new Integer (args[1]). IntValue();
        System.out.println (“Das Produkt von “ +x+ “ und
                            “ +y+ “ ist “ +mult(x,y));

    public static int mult (int x, int y) {
        int z = 0;
        while (y>0)
            if (y % 2 == 0) { y = y / 2; x = x+x ;}
            else { y = y-1; z = z+x; }

        return z;
    }
}
```