Vorlesung Programmierung II: SS 2004

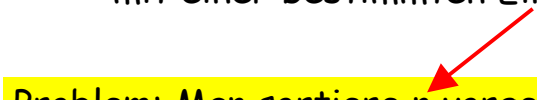
Laufzeitkomplexität und Asymptotische Notation

Prof. Dr. Hans-Peter Lenhof

lenhof@bioinf.uni-sb.de

Laufzeitkomplexität

Gegeben ein Problem
mit einer bestimmten Eingabegröße, z.B.



Problem: Man sortiere n vorgegebene ganze Zahlen.

Zunächst entwickelt man
effiziente Rechenverfahren (Algorithmen)
zur Lösung des Problems.

Dann erst implementiert man einen effizienten
Algorithmus für das Problem.

Der Ursprung des Wortes "Algorithmus"

Historische Herkunft des Begriffs Algorithmus

Das Wort *Algorithmus* geht auf die lateinische Fassung eines arabischen Namens zurück. Dieser gehörte dem Gelehrten Al-Chwarizmi, der seine mathematischen Werke im 9. Jahrhundert am Hofe des Kalifen Al-Ma'mun in Bagdad schrieb. Eines von ihnen hieß *Hisab al-gabr wal-muqabala,* d.h. Rechenverfahren durch Ergänzen und Ausgleichen. Das Wort *Algebra* stammt aus diesem Titel und ist offenbar auch historisch mit dem Lösen von Gleichungen verbunden. In Spanien tauchte der Name des Verfassers rund drei Jahrhunderte später in einer lateinischen Bearbeitung seiner Bücher auf. Diese beginnt mit den Worten:

Dixit Algoritmi ... [Es sprach Algoritmi ...]

Im Laufe der Zeit verband man die daraus entstandene Verballhornung

'Algorithmus, ganz allgemein mit mechanisch ausführbaren Rechenverfahren.

Laufzeitkomplexität

Die Untersuchung der Effizienz von Algorithmen ist eine zentrale Aufgabe der Informatik.

schnelle und stabile Algorithmen

Wir benötigen Verfahren zur Untersuchung der Laufzeiten von Algorithmen.

Laufzeitkomplexität

Die Laufzeit ist in der Regel
abhängig von der Größe der Eingabe .

Die Größe des Problems wird meist
durch eine ganze Zahl n beschrieben.

Wir diskutieren das Laufzeitverhalten
eines Algorithmus als eine Funktion in/von n.

Beispiel: Schuladdition

Problem: Gegeben zwei beliebige positive ganze Zahlen a und b.
Berechne die Summe der Zahlen a und b.

Beispiel für einen einfachen Algorithmus zur Lösung des Problems:

Schuladdition:

```
 1 3 5 6
 6 5 5 6
 0 1 1
-------
 7 9 1 2
```

← Konkrete Instanz des Problems

Eingabegröße: n = max { #Ziffern(a), #Ziffern(b) }

Wie stellen wir große ganze Zahlen dar?

Wir nehmen an, dass die Zahlen a und b beliebig groß (lang) sein können:

→ Darstellung als „int“ oder „long“ nicht möglich.

Sei B >= 2 eine beliebige Basis.

Jede nichtnegative ganze Zahl a lässt sich eindeutig darstellen als

$$\sum_{i=0}^{n-1} a_i B^i =: (a_{n-1} \ldots a_0) \quad \text{mit} \quad 0 \le a_i < B \quad \text{für alle} \quad i \in \{0,1,\ldots,n-1\}$$

Es bietet sich an, B = 2^{31} zu wählen und eine Zahl a = (a_{n-1} ... a_0) als einen Vektor (vector<unsigned int>) von n unsigned int‘s zu speichern.

Beispiel: Schuladdition

Zur Darstellung lange Zahlen implementieren wir die Klasse „Integer“ :

Integer a,b;	deklariert und definiert zwei Objekt der Klasse Integer
a.digits()	gibt die Anzahl der Stellen n von a bzgl. Basis B zurück
b.digits()	gibt die Anzahl der Stellen n von b bzgl. Basis B zurück
a[i]	gibt die i-te Stelle a_i von a zurück (0, falls i >= n)
b[i]	gibt die i-te Stelle b_i von b zurück (0, falls i >= n)

Eine primitive Addition ist die Addition dreier „einstelliger“ Zahlen:

```
unsigned int  primitive_add(unsigned int i , unsigned int j,
                                              unsigned int& carry);
```

Das Resultat einer primitiven Addition kann höchstens zweistellig sein, wobei der Übertrag in der int-Variablen „carry“ gespeichert wird.

Beispiel: Schuladdition


```
Integer Integer::operator+(const Integer& a)
{
    int          n = (a.digits() < this->digits() ? this->digits():a.digits());
    Integer      sum(n+1);              // Variable für die Summe
    unsigned int carry = 0;             // Variable für den Übertrag

    for (int i=0; i<n ; i++)
        {
            sum[i] = primitive_add(a[i],(*this)[i], carry);
        }
    sum[n] = carry;
    return sum;
}
```

```
1 3 5 6
6 5 5 6
0 1 1 0
-------
7 9 1 2
```

Wir versuchen die „Größenordnung“ der Laufzeit zu bestimmen,

d.h., gewisse Konstanten werden vernachlässigt
(genauere Definition folgt).

Beispiel: Schuladdition


```
Integer Integer::operator+(const Integer& a)
{
    int          n = (a.digits() < this->digits() ? this->digits():a.digits());
    Integer      sum(n+1);              // Variable für die Summe
    unsigned int carry = 0;             // Variable für den Übertrag

    for (int i=0; i < n ; i++)
        {
            sum[i] = primitive_add(a[i],(*this)[i], carry);
        }
    sum[n] = carry;
    return sum;
}
```

```
1 3 5 6
6 5 5 6
0 1 1 0
-------
7 9 1 2
```

„Größenordnung“ der Laufzeit wird bestimmt durch (ist proportional zur) Zahl der primitiven Operationen.

Laufzeit $\leq c * \#\text{primitive Operationen} = c * n$

Der Schulmultiplikationsalgorithmus


```
Integer Integer::operator*(const Integer& a)
{
      int  n = this->digits(), m = a.digits();
      Integer  p;

      for (int i = 0; i < m ; i++)
      {     // multipliziere (*this) mit der i-ten Ziffer  von  a
            p = p + (((*this) * a[i]) << i);
       }    // und schiebe das Resultat um i nach links

      return p;
}
```

```
81 * 111
-----------
   81
    81
     81
-----------
   8991
```

Zusätzliche C++-Operatoren für die Klasse „Integer“:

```
                                        //  bei Eingabegröße n  Ziffern(B)
Integer operator+(const Integer& );     //  n primitive Add.
Integer operator*(unsigned int );       //  ≤ n  primitive Add.
Integer operator<<( int /*i */ );       //  keine primitiven Additionen
Integer operator=(const Integer&);      //  keine primitiven Additionen
```

Der Schulmultiplikationsalgorithmus

Wir analysieren die Laufzeit f(n) für den Fall, dass beide Zahlen gleich lang sind, d.h., m = n.

C++-Operatoren für die Klasse „Integer“ mit primitiven Additionen:

```
                                        //  bei Eingabegröße n  Ziffern(B)
Integer operator+(const Integer& );     //  n primitive Add.
Integer operator*(unsigned int );       //  ≤ n  primitive Add.
```

```
for (int i = 0; i < n ; i++)     p = p + (((*this) * a[i]) << i);
```

$$f(n) \le \sum_{i=0}^{n-1}(2n+1+i) = 2n^2 + n + \sum_{i=0}^{n-1} i$$
$$\le 2n^2 + n + \frac{1}{2}n(n-1)$$
$$\le \frac{5}{2}n^2 + \frac{1}{2}n \le 5n^2$$

Laufzeit als eine Funktion der Größe der Eingabe

13

- Primitive Operationen werden definiert (eine präzisere Definition von primitiven Operationen wird folgen).
- Jede primitive Operation wird mit einer Zeiteinheit (1U) bewertet.
- D.h., Unterschiede in den Rechenzeiten bedingt durch unterschiedliche Hardware/Compiler werden nicht berücksichtigt.
- Laufzeit = Zahl der primitiven Operationen
  = Funktion der Eingabegröße n
- Meist kann man nur eine obere Schranke für die Laufzeit angeben (Worst-Case-Analyse) (siehe Schulmultiplikation).
- Diese obere Schranke sollte so “scharf” wie möglich sein.
- “Größenordnung” der Laufzeit?

RAM oder Random Access Machine

14

Wir definieren uns ein einfaches Rechnermodell, dass es uns erlaubt, die Ressourcen, welche von einem Algorithmus benötigt werden, abzuschätzen.

RAM oder Random Access Machine

1. Die RAM hat eine unendliche Folge von Registern oder Speicherzellen.
2. Die folgenden Operationen sind primitive Operationen, d.h., sie können in konstanter Zeit durchgeführt werden:
   - Lese- oder Schreibzugriff auf jedes Register
   - Additionen, Subtraktionen, Multiplikationen, Divisionen von einfachen Datentypen für Zahlen (int, float, usw.)
   - Deklarationen von einfachen Datentypen (short, int, long, float, double, char, bool, usw.)
   - Zuweisungen {=} für einfache Datentypen
   - Sprunganweisungen („goto“), z.B., in if-Anweisung
   - Vergleiche {==, !=, < , > , >= , <= } für einfache Datentypen
   - Boolsche Operationen { &&, || , & , ! , usw. } angewendet auf einfache Datentypen.

RAM oder Random Access Machine

Annahme: Primitive Operationen können auf einer RAM in konstanter Zeit ausgeführt werden.

⇩

Um die Rechenzeit eines Algorithmus für eine konkrete Instanz eines Problems abzuschätzen, müssen wir die Zeiten der primitiven Operationen aufsummieren.

⇩

Die Rechenzeit eines Algorithmus kann als eine Funktion der Eingabegröße (Problemgröße) angegeben werden.

⇩

$$F = \{ f : N \rightarrow R_0^+ \}$$

ist die Menge aller Funktionen, die Rechenzeiten (bzgl. einer Eingabegröße n) darstellen können.

RAM oder Random Access Machine

Wir üben jetzt das Abschätzen von Rechenzeiten

- Jeder Operator (für einen Grunddatentyp) benötigt konstante Zeit.
- Eine Deklaration/Definition eines Grunddatentyps benötigt auch konstante Zeit.
- Reservieren wir Speicherplatz für „größere“ Objekte, z.B. einen Vektor

```
vector<int> v(n);
```

so zählen wir hierfür cn Zeiteinheiten (≈ Größe).

RAM oder Random Access Machine

Beispiel: Skalarprodukt von Vektoren

```
int i, n;                                          // c1          1
float p = 0.0;                                     // c2          1
cin >> n;                                          // c3          1

vector<float> v1(n), v2(n);                        // c4n         1

for (i = 0; i < n ; i++) cin >>  v1[i]  >> v2[i];  // c5          n

for (i=0; i < n ; i++)  p += v1[i] * v2[i];        // c6          n

cout <<  p;                                        // c7          1
```

$$f(n) = (c_4 + c_5 + c_6)n + (c_1 + c_2 + c_3 + c_7)$$

Setze $c = 2\max\{(c_4 + c_5 + c_6), (c_1 + c_2 + c_3 + c_7)\}$

$f(n) \leq cn$

RAM oder Random Access Machine

Beispiel: Insertion-Sort (n = v.size())

```
                                                   // K. /Wie oft?
void  insertionSort(vector<int>& v) {              //  c1   1
  int key, i, j ;                                  //  c2   1

  for (i = 1; i < v.size(); i++) {                 //  c3   n-1
    for (key = v[i], j = i-1; (j >=0)&&(key < v[j]); j--)  v[j+1] = v[j];  //  c4   Z?
    v[j+1] = key;                                  //  c5   n-1
  }
}
```

Im schlimmsten Fall (worst case) gilt:

$$Z = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$

$$f(n) \leq \frac{c_4}{2}n^2 + (c_3 + c_5 - \frac{c_4}{2})n + (c_1 + c_2 - c_5 - c_3)$$

Für eine geeignet gewählte Konstante c gilt: $f(n) \leq cn^2$

Asymptotische Notation

Frage: Gegeben zwei Algorithmen mit Laufzeiten f(n) und g(n).

Welcher Algorithmus ist schneller (effizienter)?

Unser Interesse gilt dem Laufzeitverhalten für große Eingaben n, wenn n sozusagen gegen ∞ geht.

Frage: Wann betrachten wir zwei Algorithmen als „gleich“ effizient?

Man beachte hierbei, dass „Konstanten“ eine untergeordnete Rolle spielen und „vernachlässigt“ werden können.

Asymptotische Notation

1. **Versuch: Definition von „gleich effizient“: Additive Konstante**

   Die Laufzeiten der beiden Algorithmen unterscheiden sich für hinreichend große Eingabegrößen n höchstens um eine „additive“ Konstante c > 0:

$$\| f(n) - g(n) \| \leq c \qquad \forall n \geq n_0$$

Problem mit dieser Definition:

$$f(n) = 2n \qquad g(n) = n$$

Die obigen Laufzeiten sind eigentlich gleich gut (effizient). Nach der obigen Definition sind f(n) und g(n) jedoch nicht in der gleichen „Effizienzklasse“.

Asymptotische Notation

2. Versuch: Definition von „gleich effizient“: Multiplikative Konstante

Die Laufzeiten der beiden Algorithmen unterscheiden sich für hinreichend große Eingabegrößen n höchstens um eine „multiplikative“ Konstante c > 0:

$$\frac{f(n)}{g(n)} \leq c \qquad \forall n \geq n_0$$

Problem mit dieser Definition:

$$f(n) = 10 \qquad g(n) = n^2$$

Die beiden Funktionen sind nicht gleich effizient, erfüllen aber die Bedingungen der obige Definition!

Asymptotische Notation

Definition von „gleich effizient“: Multiplikative Konstante

Für hinreichend große Eingabegrößen n existieren zwei Konstanten $c_1 > 0$ und $c_2 > 0$, so dass gilt:

$$c_1 \leq \frac{f(n)}{g(n)} \leq c_2 \qquad \forall n \geq n_0$$

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall n \geq n_0$$

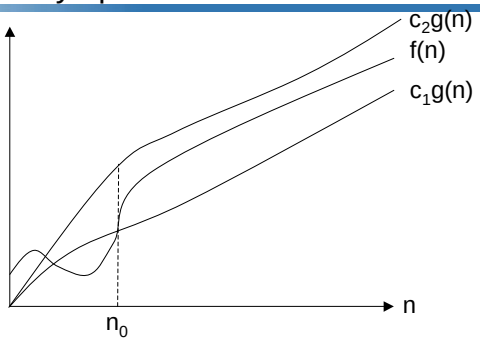
Gegeben eine Funktion (Laufzeit) g(n). Die Menge aller Funktionen, die die gleiche Effizienz wie g(n) besitzen, können wir wie folgt definieren:

(lies: „Theta-von-g“)

$$\Theta(g) = \{ f \in F \mid \exists c_1, c_2 > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

Asymptotische Notation

$f(n) \in \Theta(g)$



(lies: „Theta-von-g“)

$$\Theta(g) = \{ f \in F \mid \exists c_1, c_2 > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

Intuitiv ist f ∈ Θ(g), falls f in der gleichen Größenordnung wie g ist, d.h., f und g sind gleich oder ähnlich effizient

Asymptotische Notation

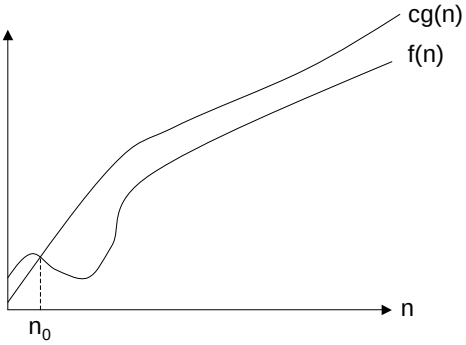
Die asymptotische Notation dient zur Klassifizierung und zum Vergleich von Funktionen nach ihren Zuwachsraten:

Sei $F = \{ f : N \to R_0^+ \}$

$$O(g) = \{ f \in F \mid \exists c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) \leq c g(n) \}$$

(„O-von-g“)

$f(n) \in O(g)$



Asymptotische Notation

Die asymptotische Notation dient zur Klassifizierung und zum Vergleich von Funktionen nach ihren Zuwachsraten:

Sei $F = \{ f : N \to R_0^+ \}$ die Menge aller Funktionen von N nach R_0^+.

Definition [1]: Für g ∈ F ist

$$\Theta(g) = \{ f \in F \mid \exists c_1, c_2 > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

$$O(g) = \{ f \in F \mid \exists c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) \leq cg(n) \}$$

$$\Omega(g) = \{ f \in F \mid \exists c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) \geq cg(n) \}$$

$$o(g) = \{ f \in F \mid \forall c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) < cg(n) \}$$

$$\omega(g) = \{ f \in F \mid \forall c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) > cg(n) \}$$

Asymptotische Notation

Stark vereinfacht können wir das asymptotische Wachstumsverhalten der verschiedenen Funktionenklassen wie folgt interpretieren:

$f \in O(g) \approx f \leq g$ [„f wächst nicht schneller als g“]

$f \in \Omega(g) \approx f \geq g$

$f \in \Theta(g) \approx f = g$

$f \in o(g) \approx f < g$

$f \in \omega(g) \approx f > g$

Beispiel: $g(n) = n^2$

$f(n) = 3n^2+7n+12 \leq 3n^2+7n^2+12n^2 \leq 22n^2 \leq c\, g(n)$

⇨ $f \in O(g)$

$f(n) = 3n^2+7n+12 \geq n^2 = g(n)$

⇨ $f \in \Theta(g)$

Asymptotische Notation

Satz [1]: Transitivität:

$$f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$$

Beweis:

$$f(n) \in O(g(n)) \Leftrightarrow \exists c_f > 0 \;\; \exists n_f > 0 \;\; \forall n \geq n_f \quad f(n) \leq c_f g(n)$$

$$g(n) \in O(h(n)) \Leftrightarrow \exists c_g > 0 \;\; \exists n_g > 0 \;\; \forall n \geq n_g \quad g(n) \leq c_g h(n)$$

Sei $c = c_f * c_g$ und $n_0 = \max\{n_f, n_g\}$ dann gilt für alle $n \geq n_0$

$$f(n) \leq c_f g(n) \leq c_f c_g h(n) \leq ch(n)$$

$$\Rightarrow \exists c > 0 \;\; \exists n_0 > 0 \quad \text{so dass} \quad \forall n \geq n_0 \quad \text{gilt:} \quad f(n) \leq ch(n)$$

$$\Leftrightarrow f(n) \in O(h(n))$$

■

Satz [2]: Reflexivität:

$$f(n) \in O(f(n))$$

Asymptotische Notation

Satz [3]: Symmetrie:

$f(n) \in \Theta(g(n))$ dann und nur dann $g(n) \in \Theta(f(n))$

Beweis: Übung.

Ferner zeige man, dass die Symmetrie-Aussage nicht für „O“ (falls Θ oben durch O ersetzt wird) gilt.

Lemma [1]:

[a] $O(g(n)) + O(h(n)) \subseteq O(\max\{g(n), h(n)\})$

[b] $c * O(g(n)) \subseteq O(g(n))$

[c] $O(g(n))O(h(n)) \subseteq O(g(n)h(n))$

Beweis:

Wir beweisen hier die Aussage [a] von Lemma (3).
Die Beweise von Aussage [b] und [c] sind Übungen.

Die Aussage [a] kann wie folgt umformuliert werden:

nächste Seite

Asymptotische Notation

[a] $O(g)+O(h)\subseteq O(\max\{g,h\})$

$\Leftrightarrow \quad \forall f_g \in O(g) \wedge \forall f_h \in O(h): \qquad f_g(n)+f_h(n)\in O(\max\{g,h\})$

$\forall f_g \in O(g): \exists c_{f_g} > 0 \wedge \exists n_{f_g}$ so dass $\forall n \geq n_{f_g}$ gilt: $f_g(n) \leq c_{f_g} g(n)$

$\forall f_h \in O(h): \exists c_{f_h} > 0 \wedge \exists n_{f_h}$ so dass $\forall n \geq n_{f_h}$ gilt: $f_h(n) \leq c_{f_h} h(n)$

Wir betrachten nun die Summe zweier beliebiger Funktionen f_g und f_h:

$\forall n \geq \max\{n_{f_g}, n_{f_h}\}$ gilt:

$$\begin{aligned} f_g(n)+f_h(n) &\leq c_{f_g} g(n)+c_{f_h} h(n) \leq \max\{c_{f_g}, c_{f_h}\}(g(n)+h(n)) \\ &\leq \max\{c_{f_g}, c_{f_h}\}(2\max\{g(n),h(n)\}) \\ &\leq \max\{2c_{f_g}, 2c_{f_h}\}\max\{g(n),h(n)\} \end{aligned}$$

$\Leftrightarrow \qquad f_g(n)+f_h(n)\in O(\max\{g,h\})$ ■

Asymptotische Notation

- Gegeben ein Algorithmus mit Laufzeit f(n).
- Meist können wir die Laufzeit im Worst-Case nur nach oben durch eine Funktion g(n) abschätzen, d.h., f(n) ∈ O(g(n)).
- Oft ist g(n) ein Polynom in der Variablen n:

$$g(n) = \sum_{i=0}^{k} c_i n^i$$

- Da

$$\lim_{n\to\infty} \frac{g(n)}{n^k} = c_k$$

folgt, dass $g(n)\in\Theta(n^k) \Rightarrow g(n)\in O(n^k)$ und somit

$$f(n)\in O(g(n)) \wedge g(n)\in O(n^k) \xrightarrow{\text{Satz1}} f(n)\in O(n^k)$$

Asymptotische Notation

- Gegeben ein Algorithmus mit Laufzeit f(n).
- Meist können wir die Laufzeit im Worst-Case nur nach oben durch eine Funktion g(n) abschätzen, d.h., $f(n) \in O(g(n))$.
- Ist g(n) die Summe von verschiedenen Funktionen,

$$g(n) = \sum_{i=0}^{k} g_i(n)$$

so bestimmt die am „schnellsten wachsende“ Funktion $g_j(n)$ die Größenordung von g(n) (siehe Lemma [1] a):

$$f(n) \in O(g(n)) \wedge g(n) \in O(g_j(n))$$

Satz [1] $\Rightarrow$ $f(n) \in O(g_j(n))$

Die Anzahl k der Summanden muss hierbei eine Konstante sein und darf nicht von n abhängen.

Asymptotische Notation

Einige vereinfachende Schreibweisen:

$g : N \to R_0^+$

$g(n) = n \Rightarrow O(g(n)) = O(n)$

$g(n) = n^x \Rightarrow O(g(n)) = O(n^x)$

$g(n) = \log(n) \Rightarrow O(g(n)) = O(\log(n))$

$g(n) = n\log(n) \Rightarrow O(g(n)) = O(n\log(n))$ usw.

Anstelle der Schreibweise

$f(n) \in O(g(n))$ findet man in der Literatur oft auch

$f(n) \leq O(g(n))$ oder (ähnlich „schlampig“)

$f(n) = O(g(n))$

Beispiel: $f(n) = 3n^2 + 7n + 12 = O(n^2) = O(n^3)$

$f(n) = 3n^2 + 7n + 12 \in O(n^2) \subset O(n^3)$

Asymptotische Notation

Hinter der Benutzung der asymptotischen Notation steht die implizite Annahme, dass ein Algorithmus immer dann besser als ein anderer Algorithmus ist, wenn die asymptotische Zuwachsrate seiner Laufzeit (bzw. seines Speicherplatzbedarfs) kleiner als die des anderen Algorithmus ist.

Laufzeit von Algorithmus 1:

$$f_1(n) = 10^{100} n \in O(n)$$

Laufzeit von Algorithmus 2:

$$f_2(n) = n^2 \in O(n^2)$$

Für alle n mit $n < 10^{100}$ gilt jedoch:

$$f_1(n) = 10^{100} n \; > n^2 \; = f_2(n)$$

Für alle realistischen Eingabegrößen hat der Algorithmus 2 die bessere Laufzeit.

Vergleiche von Ressourcenanforderungen (Laufzeit und Speicherplatz) mittels asymptotischen Größenordnungen (O-Notation) sind nur unter bestimmten Voraussetzungen sinnvoll („u.a. Voraussetzungen über die Konstanten“).

Asymptotische Notation

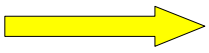
Beispiel: Wir untersuchen einen Algorithmus A und erhalten als Laufzeit

$$f(n) = 3n + 15n\log n + n^2 \in O(n) + O(n\log n) + O(n^2) \subseteq O(n^2)$$

Oft wird dies stark vereinfacht wie folgt geschrieben:

$$3n + 15n\log n + n^2 = O(n) + O(n\log n) + O(n^2) = O(n^2)$$

Diese „Gleichungen“ dürfen nur von links nach rechts gelesen werden!



Man sagt:

„Der Algorithmus hat die Laufzeit (Zeitkomplexität) $O(n^2)$.“

Asymptotische Notation

Die bekannteste Problemlösestrategie der Informatik ist „Divide&Conquer“ („teile und herrsche“).

Gegeben zwei n-stellige Zahlen

$a = (a_{n-1} \cdots a_0)_B$
$b = (b_{n-1} \cdots b_0)_B$

Teile a und b jeweils in zwei Hälften:

$a^{(1)} = (a_{n-1} \cdots a_m)_B \; a^{(0)} = (a_{m-1} \cdots a_0)_B$
$b^{(1)} = (b_{n-1} \cdots b_m)_B \; b^{(0)} = (b_{m-1} \cdots b_0)_B$

wobei $m = \lceil n/2 \rceil$

Dann gilt: $a = a^{(1)}B^m + a^{(0)} \quad b = b^{(1)}B^m + b^{(0)}$

$$ab = (a^{(1)}B^m + a^{(0)})(b^{(1)}B^m + b^{(0)}) = a^{(1)}b^{(1)}B^{2m} + (a^{(0)}b^{(1)} + a^{(1)}b^{(0)})B^m + a^{(0)}b^{(0)}$$

Hieraus folgt:

Die Multiplikation zweier n-stelliger Zahlen lässt sich also auf vier Multiplikation zweier höchstens $\lceil n/2 \rceil$-stelliger Zahlen zurückführen (plus Shift-Operationen).

Asymptotische Notation

$$ab = (a^{(1)}B^m + a^{(0)})(b^{(1)}B^k + b^{(0)}) = a^{(1)}b^{(1)}B^{m+k} + a^{(0)}b^{(1)}B^m + a^{(1)}b^{(0)}B^k + a^{(0)}b^{(0)}$$

```
Integer  operator*(const Integer& a, const Integer& b)  {
        Integer  a0 = a, a1 = 0, b0 = b, b1 = 0;
        int n = a.digits(), l = b.digits();

        // falls a und b einstellig, tut es eine primitive Multiplikation
        if (n == 1  &&  l == 1) return primitive_mult(a[0],b[0]);

        int m = n/2, k = l/2;
        if  ( m > 0 ) {
                a1 = a >> m;            // schiebe a um m Ziffern nach rechts
                a0 = a - (a1 << m);     // die rechtesten m Ziffern von a
        }

        if ( k > 0 ) {
                b1 = b >> k;            // schiebe b um k Ziffern nach rechts
                b0 = b - (b1 << k);     // die rechtesten k Ziffern von b
        }

        //   Fortsetzung auf der nächsten Folie→
```

Asymptotische Notation

$$ab = (a^{(1)}B^m + a^{(0)})(b^{(1)}B^k + b^{(0)}) = a^{(1)}b^{(1)}B^{m+k} + a^{(1)}b^{(0)}B^m + a^{(0)}b^{(1)}B^k + a^{(0)}b^{(0)}$$

```
        Integer  p1, p2, p3, p4;
        p1 = a0*b0;
        p2 = a1*b0;
        p3 = a0*b1;
        p4 = a1*b1;
        return  (p1 + (p2<<m) + (p3<<k) + (p4 << (m+k)));
}
```

Asymptotische Notation

Frage: Gegeben zwei Algorithmen mit Laufzeiten f(n) und g(n).

Welcher Algorithmus ist schneller (effizienter)?

Unser Interesse gilt dem Laufzeitverhalten für große Eingaben n, wenn n sozusagen gegen ∞ geht.

Es bietet sich daher an, folgende Grenzwerte zu betrachten:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \quad \text{bzw.} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Asymptotische Notation

Fall 1:

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=0$$ ⇒ Der Algorithmus mit Laufzeit g(n) ist effizienter!

Fall 2:

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=c \qquad \lim_{n\to\infty}\frac{f(n)}{g(n)}=\frac{1}{c}$$ ⇒ Beide Algorithmen sind „gleich“ effizient (ineffizient)!

D.h., ihre Laufzeiten unterscheiden sich nur um eine Konstante (im Limes).

Fall 3:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$ ⇒ Der Algorithmus mit Laufzeit f(n) ist effizienter!

Asymptotische Notation

Wir betrachten im folgenden die Menge aller Funktionen von N nach R_0^+.

$$F=\{f:N\to R_0^+\}$$

Die Funktionen dieser Menge sollen nach ihrem Wachstumsverhalten klassifiziert werden (in Klassen eingeteilt werden):

Beispiel: Gegeben eine Funktion (Laufzeit) g(n) aus F.

Bestimme die Menge aller Funktionen f(n) aus F, die nicht schneller wachsen als g(n), d.h., f(n) ist effizienter als g(n) oder gleich effizient.

$$O(g)=\left\{f\in F\mid\lim_{n\to\infty}\frac{f(n)}{g(n)}=c_f\wedge c_f\geq 0\right\}$$

Asymptotische Notation

$$O(g) = \left\{ f \in F \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = c_f \wedge c_f \geq 0 \right\}$$

$$= \left\{ f \in F \mid \forall \varepsilon > 0, \exists n_0 \in N : \forall n > n_0 : \left\| \frac{f(n)}{g(n)} - c_f \right\| \leq \varepsilon \right\}$$

$$= \left\{ f \in F \mid \forall \varepsilon > 0, \exists n_0 \in N : \forall n > n_0 : g(n) \left\| \frac{f(n)}{g(n)} - c_f \right\| \leq \varepsilon * g(n) \right\}$$

$$= \left\{ f \in F \mid \forall \varepsilon > 0, \exists n_0 \in N : \forall n > n_0 : \left\| f(n) - c_f * g(n) \right\| \leq \varepsilon * g(n) \right\}$$

$$= \left\{ f \in F \mid \forall \varepsilon > 0, \exists n_0 \in N : \forall n > n_0 : f(n) \leq (\varepsilon + c_f) * g(n) \right\}$$

Setze zum Beispiel $\varepsilon = 1$ und $c = 1 + c_{f,}$, dann folgt:

!

$$O(g) = \left\{ f \in F \mid \exists c > 0 \wedge \exists n_0 \in N : \forall n > n_0 : f(n) \leq c * g(n) \right\}$$