Grundlagen von Datenstrukturen und Algorithmen

Vorlesung Programmierung II: SS 2003

Asymptotische Notation

Prof. Hans-Peter Lenhof

lenhof@bioinf.uni-sb.de

Übungsleiter: Andreas Hildebrandt

anhi@bioinf.uni-sb.de

Arithmetik großer Zahlen

Schuladdition von Zahlen:

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

Konkrete Instanz des Problems

Wir benötigen eine abstrakte bzw. allgemeine Formulierung des Problems:

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

Algorithmus: Ein effektives Verfahren, mit dem man quasi „mechanisch“ beliebige Instanzen eines vorgegebenes Problem lösen kann, nennt man einen Algorithmus.

„Mechanisch ausführbares Rechenverfahren“

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.

Bevor man ein vorgegebenes Problem mittels Algorithmen auf einem Rechner lösen kann, muss man die Daten der realen Objekte des Problems mittels geeigneter Datenstrukturen (virtueller Repräsentation (Objekte)) im Rechner speichern.

Wir nehmen an, dass die Zahlen a und b beliebig groß (lang) sein können und dass sie nicht mit dem Datentyp „int“ (mit 32 Bit [<= 2^{32}]) oder „long“ (mit 64 Bit [<= 2^{64}]) repräsentiert werden können.

Bem.: Für eine vorgegebene Basis B >= 2, lässt sich jede nichtnegative Zahl a eindeutig schreiben als

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

Für die Repräsentation beliebig großer Zahlen im Rechner bietet es sich an, B = 2^{31} zu wählen und eine Zahl a = (a_{n-1} ... a_0) als einen Vektor (vector<int>) von n int‘s zu speichern.

Arithmetik großer Zahlen

Wir speichern lange Zahlen als Objekte der Klasse „Integer“ :

Integer a	deklariert und definiert ein Objekt a aus der Klasse Integer
Integer b	deklariert und definiert ein Objekt b aus 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 als int zurück (0, falls i >= n)
b[i]	gibt die i-te Stelle b_i von b als int zurück (0, falls i >= n)

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

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

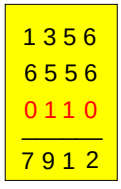
Das Resultat einer primitiven Addition kann höchstens zweistellig sein, wobei der Übertrag in der int-Variablen „carry“ gespeichert wird.

Auf der folgenden Seite geben wir die C++-Synthax (C) des Schuladditions-algorithmus an.

Arithmetik großer Zahlen

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

  for (int i=0; i<n ; i++)
  {
    sum[i] = primitive_add(a[i],(*this)[i], carry);  // primitive Add. mit 2-stelligem
                                                     // Resultat.
  }
  sum[n] = carry;
  return sum;
}
```



Lemma (1): Um zwei n-stellige Zahlen zu addieren benötigt der Algorithmus Schuladdition n primitive Additionen (Operationen).

Die Zahl der Stellen (Größe der Eingabe) n ist hier ein Maß für die Größe des Problems.

Arithmetik großer Zahlen

Der Schulmultiplikationsalgorithmus

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

Zusätzliche C++-Operatoren (Operator Overloading)

- Integer operator+(const Integer& a); } n primitive Additionen
- Integer operator*(int d); } n primitive (Add. + Mult.)
- Integer operator<<(int i); } n+i primitive Zuweisungen (Kopien)

```
Integer Integer::operator*(const Integer& a)
{
  int  n = this->digits(), m = a.digits();
  Integer  p(n*m);                    // das wird das Produkt
  for (int i = 0; i < m ; i++)
  {
     p = p + (((*this) * a[i]) << i);  // multipliziere (*this) mit der i-ten Ziffer  von a
  }                                    // und schiebe das Resultat um i nach links
  return p;
}
```

Arithmetik großer Zahlen

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

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

$$\sum_{i=0}^{n-1} 4n+2i+2 = 4n^2+2n+2\sum_{i=0}^{n-1} i = 4n^2+2n+n(n-1) = 5n^2+n$$

Lemma (2): Um zwei n-stellige Zahlen miteinander zu multiplizieren, benötigt der Algorithmus Schulmultiplikation $5n^2+n$ primitive Operationen.

Beweis: Siehe oben. ■

Die O-Notation oder asymptotische Notation

Im vorhergehenden Abschnitt haben wir die folgenden Begriffe eingeführt:

- (formale) Problemdefinition
- (konkrete) Instanz eines Problems
- Algorithmus
- Eingabegröße oder Größe der Instanz (n)
- Primitive Operationen (in unserem Beispiel: Add. + Mult. + Zuweisungen)

Ein einfaches Rechnermodell:

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, Sprunganweisungen („goto“), Vergleiche {==, !=, < , > , >= , <= }, boolsche Operationen { &&, || , & , | , usw. }

   Später werden wir noch weitere primit. Operationen kennen lernen!

Die O-Notation oder asymptotische Notation

Annahme:

Für jede primitive Operation benötigt ein Rechner (RAM) eine Zeiteinheit (1U).

⇩

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

⇩

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

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

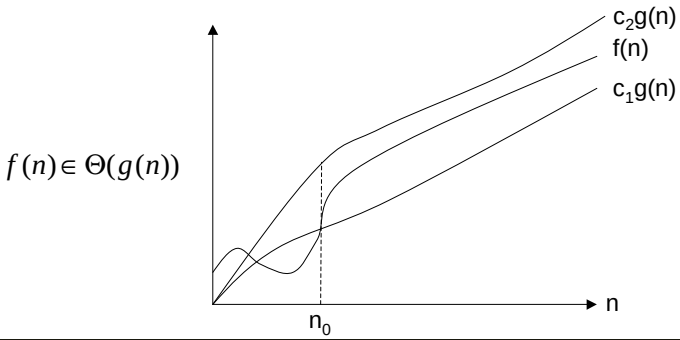
Sei $F = \{ f : N \to R_0^+ \}$ die Menge aller Funktionen von N nach R_0^+.

Definition [1]: Für $g \in F$ ist (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 $f \in \Theta(g)$, falls f in der gleichen „Größenordnung“ wie g ist.

Die O-Notation oder 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 (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∈O(g), falls f in der gleichen „Größenordnung“ wie g ist.

Die O-Notation oder asymptotische Notation

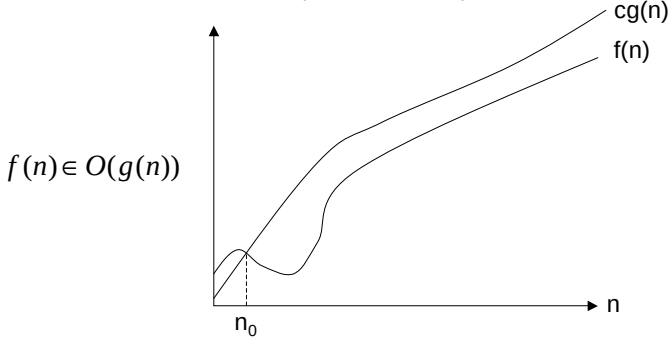
Die asymptotische Notation dient zur Klassifizierung und zum Vergleich von Funktionen nach ihren Zuwachsraten:

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

Definition [1]: Für g∈F ist

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

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



Die O-Notation oder 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 $g \in F$ ist

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

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

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

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

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

Die O-Notation oder 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^+.

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 [1]: $g(n) = n^2$

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

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

$f(n) = 3n^2+7n+12 \geq n^2 = g(n) \Rightarrow f \in \Omega(g) \Rightarrow f \in \Theta(g)$

Die O-Notation oder 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: $f(n) = 3n^2 + 7n + 12 = O(n^2) = O(n^3)$

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

Die O-Notation oder 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“).

Die O-Notation oder asymptotische Notation

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 $n \geq n_0$

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

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

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

Reflexivität:

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

Die O-Notation oder asymptotische Notation

Symmetrie:

$$f(n) \in \Theta(g(n)) \quad \text{dann und nur dann} \quad g(n) \in \Theta(f(n))$$

Beweis: Übung.

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

Lemma (3):

[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

Die O-Notation oder asymptotische Notation

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

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

$\forall f_g \in O(g): \exists c_{f_g} > 0 \wedge \exists n_{f_g}$ 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:

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

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

Die O-Notation oder asymptotische Notation

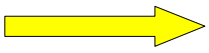
Beispiel: Wir untersuchen einen Algorithmus A und erhalten als Laufzeit

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

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) $O(n^2)$.“

Ein rekursiver Multiplikationsalgorithmus

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).

Ein rekursiver Multiplikationsalgorithmus

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

```
Integer recursive_mult(const Integer& a, const Integer& b)  {
  Integer  a0, a1, b0, b1;
  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
  }
  Integer  p1 = recursive_mult(a0,b0);
  Integer  p2 = recursive_mult(a1,b0);
  Integer  p3 = recursive_mult(a0,b1);
  Integer  p4 = recursive_mult(a1,b1);
  return  (p1 + (p2<<m) + (p3<<k) + (p4 << (m+k)));
}
```