Grundlagen der Programmierung (Vorlesung 17)

Ralf Möller, FH-Wedel

- Vorige Vorlesung
  - Blöcke, Funktionen, Prozeduren
  - Auswertestrategien, Rekursion
- Inhalt dieser Vorlesung
  - (Asymptotische) Komplexität von Algorithmen
- Lernziele
  - Grundlagen der Analyse von Algorithmen

Danksagung

- Die Präsentation wurde aus der Vorlesung "Einführung in die Informatik 1" von der Universität - Gesamthochschule Siegen übernommen:
- http://www.informatik.uni-siegen.de/~inf/EI1/Skript/

Aufwand von Algorithmen

Jede Operation eines Programms verursacht einen gewissen *Aufwand*:

- *Algorithmen* verbrauchen *Rechenzeit*,
- *Datenstrukturen* verbrauchen *Speicher*.

Der Verbrauch dieser Ressourcen soll möglichst *minimal* sein, d.h. es stellen sich folgende Fragen:

- *Wieviel* Ressourcen verbraucht ein *gegebener* Lösungsansatz?
- Was ist der *minimale* Aufwand zur Lösung des Problems?

  Dazu ist die Frage zu präzisieren: Ist die Rede vom *besten*, dem *durchschnittlichen* oder dem *aufwendigsten* Fall?

Entwurfsüberlegungen

Folgende Fragen sollte man sich stellen, bevor man *Datenstrukturen* und *Algorithmen* entwickelt bzw. verwendet:

- ❑ Für welchen *Zweck* soll der Einsatz erfolgen?
- ❑ Welche Operationen werden *unbedingt* benötigt, welche hingegen nicht?
- ❑ Welche Operationen werden *am häufigsten* verwendet? Welche sind *typisch*?
- ❑ Mit welchem *Aufwand* ist bei den einzelnen Operationen zu rechnen?

Wir wollen nun die letzte Frage näher betrachten, da sie für die Entscheidung für oder gegen bestimmte Datenstrukturen oder Algorithmen eine große Rolle spielen kann.

Begriffe

Folgende Begriffe werden bei der Aufwandsuntersuchung benutzt:

- *Problemumfang* n: meist der *Wert* oder die *Stellenanzahl* der Eingabe
- *Zeitaufwand*: Abstraktion von konkreten Zeiten zu Größenordnungen
- *Platzbedarf*: konkrete Berechnung des Speicherbedarfs des Algorithmus
- *günstigster, durchschnittlicher* und *ungünstigster* Fall, z.B. bei *Sortierung*.
- *Komplexität* eines **Algorithmus**: *asymptotischer* Aufwand (für $n \to \infty$) der *Implementierung*.
- *Komplexität* eines **Problems**: *minimale* Komplexität eines Algorithmus zur Lösung des Problems

Motivation der Komplexitätsbetrachtung

Die *Komplexität* untersucht also folgende Fragen:

- ❑ Wieviel *Rechenzeit* wird für die Ausführung bei gegebener Eingabe in etwa benötigt?
- ❑ Wie ändert sich die Rechenzeit (größenordnungsmäßig) bei *Vergrößerung der Eingabe*?
- ❑ Wieviel *Speicher* wird für die Ausführung (in Abhängigkeit von der Eingabe) benötigt?
- ❑ Ist die Rechenzeit des Algorithmus noch *zumutbar* für „große“ Eingaben?

Motivation

Die Komplexität mißt **nicht** konkrete Zeiten („`1456 ms`“), da diese u.a. abhängen von

- ❑ *Prozessortyp* und *Taktrate*,
- ❑ *Betriebssystem*, *Arbeits-* und ggf. *virtuellem Speicher* (Größe und Zugriffszeit),
- ❑ Entwicklungs- und *Laufzeitumgebung*,
- ❑ *Compiler-* oder *Interpreterversion*, *Übersetzungsparameter* und *Optimierung*,
- ❑ *Buslast* zum Zeitpunkt der Ausführung
- ❑ sowie *Speicherfragmentierung*.

Meßergebnisse in Sekunden etc. sind daher

- ❑ nur in Einzelfällen exakt *reproduzierbar*,
- ❑ *sehr schwer* auf andere Situation (gemäß der obigen Liste) *übertragbar*
- ❑ und daher von sehr *geringer Aussagefähigkeit*.

Nutzen

Wozu wird die Komplexität benötigt?

- ❑ Vergleichbarkeit von Algorithmen anhand ihres *Aufwands*
- ❑ Beurteilung, ob ein gegebener Algorithmus für ein Problem *praktisch verwendbar* ist
- ❑ Prognose, wie sich die Laufzeit *im Verhältnis zur Problemgröße* ändert
  Ist der Algorithmus nur bis zu einer *bestimmten Problemgröße* akzeptabel?

Im folgenden werden nur Funktionen auf *natürlichen Zahlen* betrachtet. Die *Problemgröße* ist in der Regel der *Wert* des Parameters.

Elementare Kosten

Zur Bestimmung der Komplexität wird zunächst die *Anzahl der Rechenschritte* bestimmt.

Dabei werden folgende Faktoren verwendet (etwas vereinfacht):

Operation	**Anzahl Rechenschritte**
elementare Arithmetik, Vergleich, Wertzuweisung	1
Ein- und Ausgabe	1
Funktionsaufrufe	*Komplexität der Funktion*
logische Ausdrücke	*gemäß den obigen Faktoren*
Fallunterscheidung	Komplexität des *logischen Ausdrucks* + *Maximum* der Rechenschritte beider Zweige
zusammengesetzte Anweisung	*Summe der Kosten der einzelnen Befehle*
Schleifen	Falls m Durchläufe: *Initialisierung* + $m\times$ *Komplexität des Schleifenkörpers* + $m\times$ *Komplexität des Weiterzählens*

Tabelle 22: Faktoren für die Komplexität

Beispiel 1

Die Anzahl der Rechenschritte beträgt:

$$1+1+1+(n-2)\cdot(2+2+1+1)+1$$
$$=4+6\cdot(n-2)=6\cdot n-8$$

Die Anzahl der Rechenschritte hängt also linear von n ab.

```
is-prim(n : N0) : B
    if n < 2                                  {1}
      then false
      else
        begin
            var prim : B;
                i : N0;
            prim, i := true, 2;               {1+1}
            while prim ∧ (i < n) do           {(n-2)*(2}
              if (n mod i) = 0                 {+2}
                then prim := false            {+1}
              end if;
              i := i + 1                      {+1)}
            end while;
            prim                              {1}
        end
    end if
```

O-Notation

Wie bereits erwähnt, interessiert nur das *grundsätzliche* Verhalten des Algorithmus für *größer werdende* Eingaben.

Hierzu gibt es die „**O-Notation**“, die eine Einteilung der Algorithmen in *Größenordnungen* erlaubt. Dabei wird von *unwesentlichen* Konstanten abstrahiert.

- Sei $f(n)$ die Anzahl auszuführender Rechenschritte für den Algorithmus in Abhängigkeit von der Problemgröße n
- $g(n)$ sei eine von n abhängige Funktion
- Man schreibt $f(n) \leq g(n) := \exists c.\, (\forall n.\, (f(n) \leq c \cdot g(n)))$
  In unserem Beispiel mit $f(n) = 6n - 8$ kann man also beispielsweise $c = 6$ und $g(n) = n$ wählen, denn $\forall n.(6n - 8 \leq 6n)$
- Definition der **O-Notation**:

$$f(n) \in O(g(n)) := \exists c_0.\, (\exists N_0.\, (\forall n.\, (n > N_0 \rightarrow f(n) \leq c_0 \cdot g(n))))$$

  Gesprochen „ab einem Wert N_0 liegt die Komplexität von f unter der c_0-fachen Komplexität von g“ oder verkürzt: „f liegt in der gleichen Größenordnung wie g“.

Rechenbeispiele

- f(n) = $3n - 1 \in O(n)$, denn $\forall n.\, (n > 0 \rightarrow 3n - 1 \leq 3n)$ mit $c_0 = 3, N_0 = 0, g(n) = n$
- f(n) = $4n + 7 \in O(n)$, denn $\forall n.\, (n > 6 \rightarrow 4n + 7 \leq 5n)$ mit $c_0 = 5, N_0 = 6, g(n) = n$
- f(n) = $n^2+n-1 \in O(n^2)$: $\forall n.\, \left(n > 0 \rightarrow n^2 + n - 1 \leq 2n^2\right)$ mit $c_0 = 2, N_0 = 0, g(n) = n^2$

Beachten Sie:

- Es kommt *nicht* auf den *kleinstmöglichen* **Startwert** N_0 an. Statt $N_0 = 6$ im zweiten Beispiel hätte man auch $N_0 = 100$ wählen können.
- Es kommt *nicht* auf die *kleinstmögliche* **Konstante** c_0 an. Im zweiten Beispiel hätte man statt $c_0 = 5$ genausogut $c_0 = 20$ wählen können.

  Die Konstante c_0 ist nur ein Multiplikator für die Laufzeit, die unabhängig von der Problemgröße ist und daher für die Einteilung in Klassen vernachlässigt werden kann.
- Unter mehreren möglichen Komplexitätsklassen ist stets die *kleinste* zu wählen.

  So gilt zwar rechnerisch $3n - 1 \leq 20n^7$ und somit $3n - 1 \in O(n^7)$, diese Aussage ist aber *nicht konstruktiv* und daher zu unterlassen.
- **Faustregel:** der *höchste Exponent* bzw. die *höchste Basis* (2^n, 3^n etc.) dominiert.

Weitere Symbole

Die „**O-Notation**“ geht auf den Zahlentheoretiker *Edmund Landau* (1877-1938) zurück, das „**O**“ wird daher auch als „Landau’sches Symbol“ bezeichnet.

$O(g(n))$ bezeichnet dabei eine ganze *Klasse* von Funktionen; so liegen in $O(n)$ *alle* Funktionen, die *maximal* linear mit n wachsen.

Es gibt insgesamt drei verschiedene Symbole:

- $f(n) \in O(g(n))$: „$f(n)$ wächst *höchstens* so schnell wie $g(n)$“ (**obere** Schranke)
- $f(n) \in \Omega(g(n))$: „$f(n)$ wächst *mindestens* so schnell wie $g(n)$“ (**untere** Schranke)
- $f(n) \in \Theta(g(n))$: „$f(n)$ wächst *genauso* schnell wie $g(n)$“ (**exakte** Schranke)

Weitere Symbole

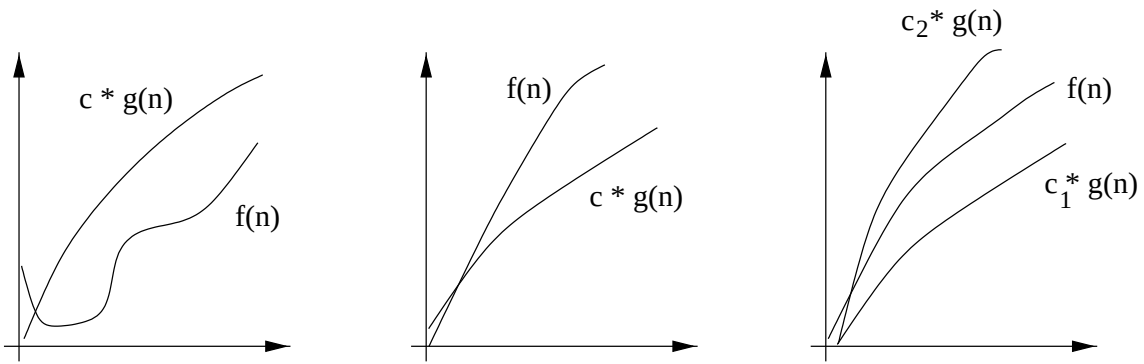


Abbildung 19: Schema von O, Ω und Θ

Beispiele zu den Komplexitätsklassen

$O(1)$	Tritt ein, wenn das Programm nur einmal linear durchlaufen wird, ohne von der Problemgröße abzuhängen (d.h. auch ohne Schleifen über n) ☞ *„konstante Laufzeit“*
$O(\log n)$	Die Rechenzeit wächst logarithmisch zur Problemgröße. Häufig bei Zerlegung von Problemen in Teilprobleme und Berechnung *eines* der Teilprobleme *Beispiel*: *Binäre Suche*
$O(n)$	Lineare Zunahme der Rechenzeit mit n. Typischer Vertreter sind Schleifen von $i \ldots n$ ($i < n$) sowie alle Operationen, die jedes Element genau k-mal referenzieren für ein *von* n *unabhängiges* k. *Beispiel*: Invertieren eines Bildes; *Sequentielle Suche*
$O(n \log n)$	Linear logarithmische Zunahme der Rechenzeit. Typisch bei Zerlegung des Problems in Teilprobleme mit Bearbeitung *aller* Teilprobleme. *Beispiel*: Bessere Sortierverfahren, etwa *Quicksort*

Die folgende Tabelle enthält die *wichtigsten Komplexitätsklassen*:

Klasse	**Leseweise**
$O(1)$	Die Rechenzeit ist unabhängig von der Problemgröße.
$O(\log n)$	Die Rechenzeit wächst *logarithmisch* (zur Basis 2) mit der Problemgröße.
$O(n)$	Die Rechenzeit wächst *linear* mit der Problemgröße.
$O(n \cdot \log n)$	Die Rechenzeit wächst *linear logarithmisch* mit der Problemgröße.
$O(n^2)$	Die Rechenzeit wächst *quadratisch* mit der Problemgröße.
$O(n^3)$	Die Rechenzeit wächst *kubisch* mit der Problemgröße.
$O(2^n)$	Die Rechenzeit wächst *exponentiell (zur Basis 2)* mit der Problemgröße.

Tabelle 23: Die wichtigsten Komplexitätsklassen

Verhalten Grundfunktionen

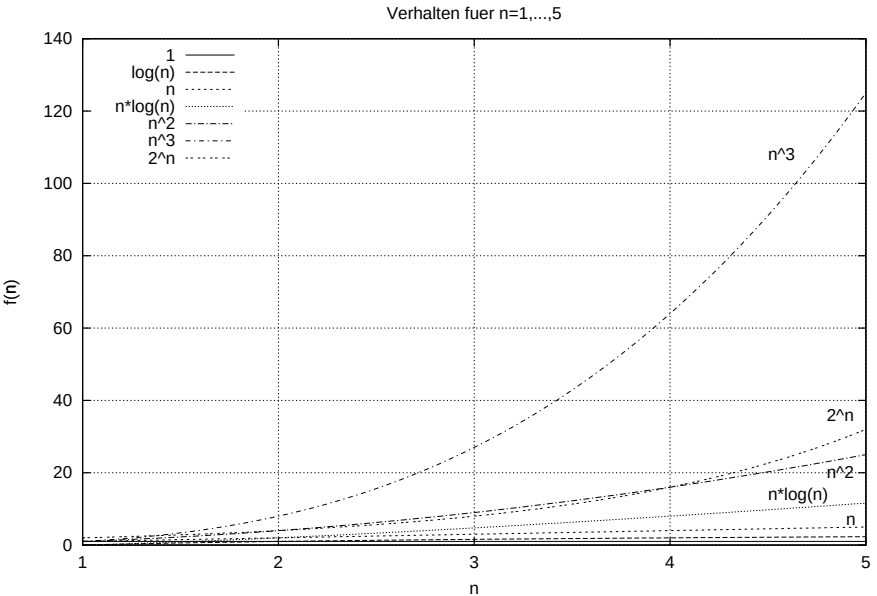


Abbildung 20: Verhalten der Funktionen für n=1,. . .,5

Beispiele zu den Komplexitätsklassen

$O(n^2)$	Quadratische Zunahme der Rechenzeit mit n. Typisch bei verschachtelten Schleifen. *Beispiel*: Paarweiser Vergleich aller Elemente bei „naheliegendem“ Sortierverfahren.
$O(n^3)$	Kubische Zunahme der Rechenzeit mit n. Typisch bei dreifach verschachtelten Schleifen. *Beispiel*: Matrixmultiplikation mit $c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$
$O(2^n)$	Exponentielle Zunahme der Rechenzeit (zur Basis 2). Typisch bei der erschöpfenden Suche in allen Kombinationen von Paaren oder Teilmengen.
$O(n!)$	Zunahme gemäß Fakultätsfunktion. Tritt bei der Bildung aller Permutationen auf.

Auf den beiden nächsten Folien sind Diagramme für das Wachstum dieser Funktionen angegeben.

Verhalten Grundfunktionen

Beispiel für einen 500 Mhz-Rechner mit Rechenzeit $2ns$ pro Taktzyklus:

	Wert von n						
$O(\ldots)$	2	4	8	16	32	64	128
$O(\log_2 n)$	$2ns$	$4ns$	$6ns$	$8ns$	$10ns$	$12ns$	$14ns$
$O(n)$	$4ns$	$8ns$	$16ns$	$32ns$	$64ns$	$128ns$	$256ns$
$O(n \log_2 n)$	$4ns$	$16ns$	$48ns$	$128ns$	$320ns$	$768ns$	$1792ns$
$O(n^2)$	$8ns$	$32ns$	$128ns$	$512ns$	$2\mu s$	$8\mu s$	$32\mu s$
$O(n^3)$	$16ns$	$128ns$	$1\mu s$	$8\mu s$	$65\mu s$	$524\mu s$	$4ms$
$O(2^n)$	$8ns$	$32ns$	$512ns$	$131\mu s$	$8.59s$	$1169a$	$2 \cdot 10^{22}a$
$O(3^n)$	$18ns$	$162ns$	$13\mu s$	$86ms$	$42.89d$	$2 \cdot 10^{14}a$	$7.5 \cdot 10^{44}a$
$O(n!)$	$4ns$	$48ns$	$81\mu s$	$11.6h$	$1.67 \cdot 10^{28}a$	$1.9 \cdot 10^{74}a$	$2 \cdot 10^{214}a$

Tabelle 24: Berechnungsdauer von n Rechenschritten nach Komplexitätsklasse

Deutlich zu erkennen ist das geringe Wachstum (Addition) bei $\log n$, das ebenfalls geringe Wachstum von $n \log n$ und das starke Zunehmen von 2^n.

Aufgrund der Berechnung von $\log n$ (zur Basis 2) wurden nur Zweierpotenzen verwendet.

Verhalten Grundfunktionen

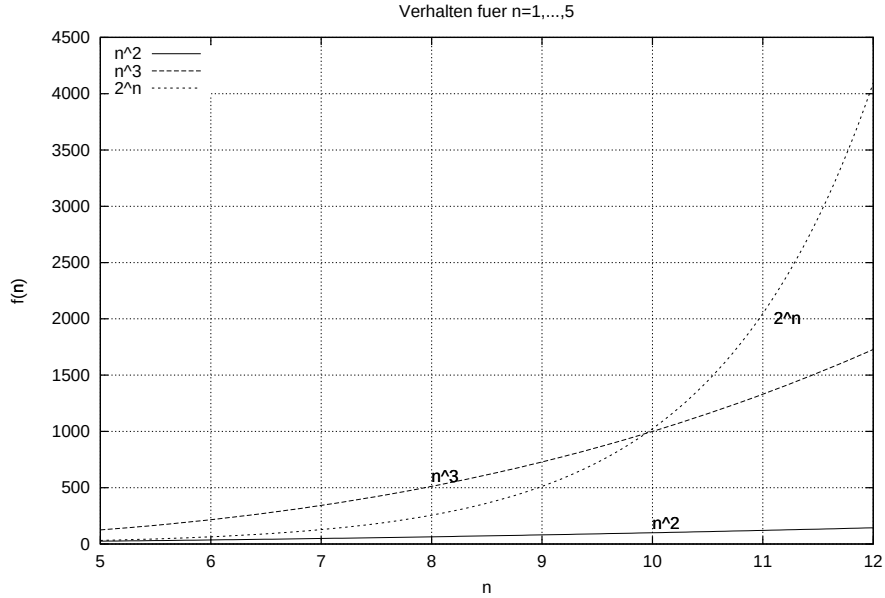


Abbildung 21: Verhalten der Funktionen für n=5,. . .,12

Average, Best und Worst Case

Die Komplexitätstheorie unterscheidet weiter in folgende Kategorien:

- *worst case*-Komplexität („**worst**“)
  Gibt die Komplexität des *rechenintensivsten* Falls an.
- *average case*-Komplexität („**avg**“)
  Dies gibt die Komplexität des *durchschnittlichen* Falls an.
- *best case*-Komplexität („**best**“)
  Beschreibt die Komplexität des *günstigsten* Falls.

Naheliegenderweise gilt

$$\textbf{best} \leq \textbf{avg} \leq \textbf{worst}$$

Für einige Algorithmen fallen die Komplexitätsklassen zusammen, so daß z.B. „**avg**“ und „**worst**“-case identisch sind, z.B. bei dem Sortierverfahren *Sortieren durch Auswahl*.

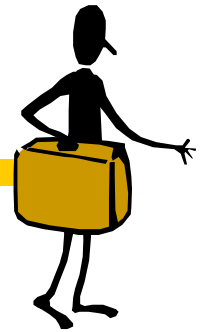
Die Bestimmung des tatsächlichen „durchschnittlichen“ Falls ist in der Regel nicht trivial.

Die übliche Kenngröße eines Algorithmus ist der *worst case* - es interessiert vor allem, womit *im schlimmsten Fall* zu rechnen ist.

Ergebnis zu den Grundfunktionen

- ❑ Solange n noch sehr klein ist, verhält sich 2^n sogar günstiger als etwa n^3 (bis $n = 9$)
- ❑ Das schnelle Wachstum der Funktionen n^3 und 2^n ist deutlich zu erkennen.
- ❑ Die Funktionen lassen sich in der in der Tabelle 23 angegebenen Reihenfolge anordnen.
- ❑ Günstige Komplexitäten liegen bis $O(n \log n)$
- ❑ $O(n^2)$ ist ebenfalls noch mit akzeptablem Wachstum verbunden
- ❑ $O(n^3)$ ist nur für relativ geringe n akzeptabel.
- ❑ Exponentielles Wachstum *vervielfacht* die Laufzeit bei Hinzunahme *eines* Elements
  - ☞ nicht mehr akzeptable Laufzeiten.

Zusammenfassung, Kernpunkte



- (Asymptotische) Komplexität von Algorithmen

Was kommt beim nächsten Mal?

- Datenstrukturen
- Algorithmen und deren Analyse