Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Komplexität und Effizienz

Was ist Effizienz, was ist Komplexität, wie sind sie definiert?

Komplexität und Effizienz

Was ist Effizienz, was ist Komplexität, wie sind sie definiert?

Wie wird Komplexität/Effizienz berechnet?

Komplexität und Effizienz

Was ist Effizienz, was ist Komplexität, wie sind sie definiert?

Wie wird Komplexität/Effizienz berechnet?

Wie wird Komplexität/Effizienz gemessen?

Komplexität und Effizienz

Was ist Effizienz, was ist Komplexität, wie sind sie definiert?

Wie wird Komplexität/Effizienz berechnet?

Wie wird Komplexität/Effizienz gemessen?

Was ist praktisch relevant?

Einstieg

Für die algorithmische Lösung eines gegebenen Problems ist es unerlässlich, dass der gefundene Algorithmus das Problem korrekt löst. Darüber hinaus ist es natürlich wünschenswert, dass er dies mit möglichst geringem Aufwand tut. Manchmal ist dies auch unerlässlich – etwa beim automatischen Abschalten eines Kernkraftwerkes im Falle einer Havarie.

- Die Theorie der Komplexität von Algorithmen beschäftigt sich damit, gegebene Algorithmen hinsichtlich ihres Aufwandes abzuschätzen und – darüber hinaus – für gegebene Problemklassen zu bestimmen, mit welchem Mindestaufwand Probleme dieser Klasse gelöst werden können.

Einstieg

Für die algorithmische Lösung eines gegebenen Problems ist es unerlässlich, dass der gefundene Algorithmus das Problem korrekt löst. Darüber hinaus ist es natürlich wünschenswert, dass er dies mit möglichst geringem Aufwand tut. Manchmal ist dies auch unerlässlich – etwa beim automatischen Abschalten eines Kernkraftwerkes im Falle einer Havarie.

- Die Theorie der Komplexität von Algorithmen beschäftigt sich damit, gegebene Algorithmen hinsichtlich ihres Aufwandes abzuschätzen und – darüber hinaus – für gegebene Problemklassen zu bestimmen, mit welchem Mindestaufwand Probleme dieser Klasse gelöst werden können.

Wir haben bereits Beispiele aus der Numerik kennen gelernt, wo unterschiedliche Algorithmen das gleiche Problem mit unterschiedlicher Effizienz lösen. Das Ergebnis war (fast) immer gleich (aber in der Numerik nicht selbstverständlich).

Ein Beispiel: Suche in linearen Arrays

Die Bestimmung des Aufwandes wollen wir zunächst an einem einfachen Beispiel verdeutlichen: der sequenziellen **Suche** in Folgen. Gegeben seien hierzu

i=IndexOf($b \in \{a_1,..., a_n\}$)

- eine Zahl $n \geq 0$,
- n Zahlen **A**=$a_1,..., a_n$, die alle verschieden sind,
- eine Zahl b.

Gesucht wird der Index $i = 1, 2,..., n$, so dass b = a_i ist, sofern ein solcher Index existiert. Sonst soll $i = -1$ ausgegeben werden.

Ein Beispiel: Suche in linearen Listen

Eine sehr einfache Lösung (Implementierung) dieses Suchproblems ist die folgende (wobei standardmäßig $a_{n+1} = 0$ gesetzt sei) und mit i=0 als programmatischen Startindex:

```
a=[1,2,5,6,7]
b=5

i=0
n=length(a)
while (i<n && b != a[i]) {
  i=i+1
}
if (i==n) i=-1; // nicht gefunden
```

Alg. 1. Suche in linearer Array Liste

Am Ende hat i den gesuchten Ausgabewert (oder -1 wenn nicht gefunden).

Analyse

Der Aufwand der Suche, d.h. die Anzahl der Schritte, hängt von der Eingabe ab, d.h. von n, $\mathbf{A}=a_1,...,a_n$ und b. Es gilt:

1. Bei erfolgreicher Suche, wenn $b = a_i$ ist, werden $S = i$ Schritte benötigt.
2. Bei erfolgloser Suche werden $S = n + 1$ Schritte benötigt.

Die erste Aussage hängt noch von zu vielen Parametern ab, um aufschlussreich zu sein. Man ist bemüht, globalere Aussagen zu finden, die nur von einer einfachen Größe abhängen. Hier bietet sich die Länge n der Eingabeliste an und man beschränkt sich auf Fragen der Art:

A. Wie groß ist S für ein gegebenes n im **schlechtesten Fall**?
B. Wie groß ist S für ein gegebenes n im **Mittel**?

Wir analysieren nun den Fall der erfolgreichen Suche:

Analyse

A. Im schlechtesten Fall wird *b* erst im letzten Schritt gefunden, d.h. $b = a_n$. Also gilt: $S = n$ im schlechtesten Fall.

B. Um eine mittlere Anzahl von Schritten zu berechnen, muss man Annahmen über die Häufigkeit machen, mit der – bei wiederholter Verwendung des Algorithmus mit verschiedenen Eingaben – *b* an erster, zweiter, …, letzter Stelle gefunden wird.
   - Wird b häufiger am Anfang der Liste gefunden, so ist die mittlere Anzahl von Suchschritten sicherlich kleiner, als wenn *b* häufiger am Ende der Liste gefunden wird.
   - Als einfaches Beispiel nehmen wir die Gleichverteilung an. Läuft der Algorithmus N-mal, N ≫ 1, so wird *b* gleich häufig an erster, zweiter,.., letzter Stelle gefunden, also jeweils *N*/*n*-mal.

Dann werden für alle *N* Suchvorgänge **insgesamt** *M* Schritte benötigt, im Mittel *S*=*M*/*N*:

$$M = \frac{N}{n}1 + \frac{N}{n}2 + .. + \frac{N}{n}n = \frac{N}{n}(1 + 2 + .. + n) = N\left(\frac{n+1}{2}\right)$$

Analyse

Also im Mittel bei Gleichverteilung:

$$S = \frac{n+1}{2}$$

Unser Algorithmus ist für das gestellte Problem keineswegs der schnellste, sondern eher einer der langsameren. Es gibt Algorithmen, die die Suche in Listen (oder Tabellen) mit *n* Einträgen unter speziellen Voraussetzungen i.W. mit einer konstanten, nicht von *n* abhängigen Anzahl von Schritten lösen!

- Die besten Suchverfahren für direkten Zugriff, die auch Ordnung der Schlüssel zulassen, benötigen eine logarithmische Anzahl von Schritten, d.h. $S = a \cdot \log2(b \cdot n) + c$, wobei *a*, *b*, *c* Konstanten sind.

Asymptotische Analyse

Meist geht es bei der Analyse der Komplexität von Algorithmen bzw. Problemklassen darum, als Maß für den Aufwand eine Funktion $F(n)$ zu finden.

$F(n) = a$ bedeutet: Bei einem Problem der Größe n ist der Aufwand a.

- Die Problemgröße n bezeichnet dabei in der Regel ein Maß für den Umfang einer Eingabe, z.B. die Anzahl der Elemente in einer Eingabeliste oder die Größe eines bestimmten Eingabewertes.
- Der Aufwand a ist in der Regel ein grobes Maß für die Rechenzeit, jedoch ist auch der benötigte Speicherplatz zuweilen Gegenstand der Analyse.
- Die Rechenzeit wird meist dadurch abgeschätzt, dass man zählt, wie häufig eine bestimmte Art von Operation ausgeführt wird, z.B. Speicherzugriffe, arithmetische Operationen.
  - Auf diese Weise erhält man ein maschinenunabhängiges Maß für die Rechenzeit, die mit einer **abstrakten Maschine** maschinenunabhängig bestimmt werden kann ⇒ **Continuum VM in Java2JS**!

Asymptotische Analyse

Die Aufwandsfunktion *F* lässt sich in den wenigsten Fällen exakt bestimmen.
Vorherrschende Analysemethoden sind wie bereits eingeführt:

- Abschätzungen des Aufwandes im schlechtesten Fall
- Abschätzungen des Aufwandes im Mittel
- Abschätzungen des Aufwandes im besten Fall (Business best hope)

Selbst hierfür lassen sich im Allgemeinen keine exakten Angaben machen. Man beschränkt sich dann auf approximiertes Rechnen in Größenordnungen.

Rechnen in Größenordnungen

Das folgende Beispiel illustriert die Schwierigkeit von exakten Analysen, insbesondere bei Abschätzungen im Mittel, bereits in recht einfachen Fällen.

Rechnen in Größenordnungen

- Gegeben: $n \geq 0$, $\boldsymbol{A}=\{a_1,..,a_n, a_i \in \mathbb{N}\}$
- Gesucht: Der Index *i* der (ersten) größten Zahl

```
a=[5,2,1,7,6]

n=length(a)
amax=a[0],imax=0
for (i=1;i<n;i++) {
  if (amax<a[i]) { amax=a[i]; imax=i }
}
```

Alg. 2. Maximumssuche in linearer Array Liste

Rechnen in Größenordnungen

Analyse: Wie oft wird die Anweisung {*} im **Mittel** ausgeführt, abhängig von *n*? (Worst case wäre *n*-1, best case 1).

- Diese gesuchte mittlere Anzahl sei T_n. Offenbar gilt: $1 \leq T_n \leq n$.
- Beobachtung: {*} wird genau dann ausgeführt, wenn a_i das größte der Elemente $\{a_1,..,a_i\}$ ist.
- Annahme (Gleichverteilung): Für jedes $i = 1,..,n$ hat jedes der Elemente $a_1,..,a_n$ die gleiche Chance, das größte zu sein.
  - Das heißt, dass bei *N* Durchläufen durch den Algorithmus, $N \gg 1$, insgesamt *N*/*n*-mal die Anweisung {*} ausgeführt wird für $i = 1,..,n$. Das heißt:
  - imax := 1 wird *N*-mal, d.h. immer, ausgeführt
  - imax := 2 wird *N*/2-mal ausgeführt
  - etc.

Rechnen in Größenordnungen

Die gesamte Ausführungszeit ist dann $N*T_n$:

$$NT_n = N + \frac{N}{2} + \frac{N}{3} + .. + \frac{N}{n} = N\left(1 + \frac{1}{2} + .. + \frac{1}{n}\right) = NH_n$$

H_n ist die harmonische Zahl mit der **Näherung** $H_n \approx ln(n)+\gamma$

Da der Aufwand schon vom Ansatz her ohnehin nur grob abgeschätzt werden kann, macht eine Feinheit wie $+\gamma$ im obigen Beispiel nicht viel Sinn. Interessant ist lediglich, dass T_n logarithmisch von n abhängt, nicht so sehr, wie dieser Zusammenhang im Detail aussieht.

- Man schreibt dafür die O-Notation mit der weiteren Approximation $ln(n) \approx log(n)$:

$$T_n = O(\log n),$$

D.h. T_n ist von der Ordnung log n, wobei multiplikative und additive Konstanten sowie die Basis des Logarithmus unspezifiziert bleiben.

Die O-Notation

Die O-Notation lässt sich mathematisch exakt definieren. Seien $f, g : \mathbb{N} \rightarrow \mathbb{N}$ Funktionen, dann gilt:

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

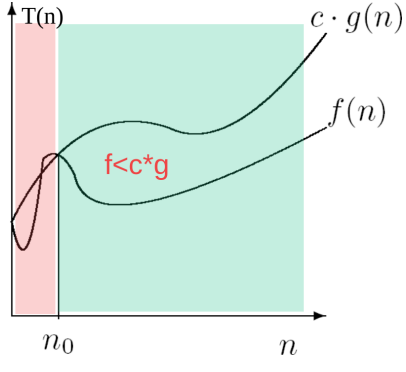
Was bedeutet diese Relation?

Das heißt, dass $f(n)/g(n)$ für genügend große $n > n_0$ durch eine Konstante c beschränkt ist. Anschaulich bedeutet dies, dass f nicht stärker wächst als g.

- Diese Begriffsbildung wendet man bei der Analyse von Algorithmen an, um Aufwandsfunktionen f durch Angabe einer einfachen Vergleichsfunktion g abzuschätzen, so dass $f(n) = O(g(n))$ gilt, also das Wachstum von f durch das von g beschränkt ist (Oberschranke).

Die O-Notation

Eigentlich ist die Angabe $f = O(g)$ im mathematischen Sinn nicht exakt. Vielmehr müsste man $f \in O(g)$ schreiben, um auszudrücken, dass f zur Klasse der Funktionen gehört, deren Wachstum asymptotisch durch das Wachstum von g beschränkt ist. Die Schreibweise mit dem »=« hat sich allerdings allgemein eingebürgert, so dass wir sie auch verwenden.



[addp]

Abb. 1. Asymptotische obere Schranke: O-Notation

Die O-Notation

Da die $O(g)$-Notation eine obere Grenze für eine Klasse von Funktionen definiert, kann die Funktion g jeweils vereinfacht werden, indem konstante Faktoren weggelassen werden sowie nur der **höchste Exponent berücksichtigt** wird.

Beispiele:

$$T_n = n^2 + 3n - 3 < O\left(n^2\right) = c \cdot n^2$$

$$T_n = \ln n + \gamma < O(\log n) = c \cdot \log\left(n\right)$$

- Diese Ungleichung wird für $n \geq n_0$ erfüllt, wenn man beispielsweise $c = 2$ und $n_0 = 1$ einsetzt (erster Fall).
- Für kleine n kann es aber signifikante Abweichungen geben (siehe Übungsaufgaben) von der tatsächlichen Rechenzeit geben!

Warum?

Die O-Notation

Komplexitätsklassen

Aufwand	Klasse
O(1)	Konstanter Aufwand (unabhängig von n)
O(log n)	Logarithmischer Aufwand
O(n)	Linearer Aufwand
O(n log n)	Linearer Aufwand
$O(n^2)$	Quadratischer Aufwand
$O(n^3)$	Kubischer Aufwand
$O(n^k)$	Polynomieller Aufwand für $k \geq 0$
$O(k^n)$	Exponentieller Aufwand

Tab. 1. Typische Komplexitätsklassen aufsteigend sortiert

Komplexitätsklassen

$f(n)$	$n=2$	$2^4=16$	$2^8=256$	$2^{10}=1024$	$2^{20}=1048576$
$\mathrm{ld}n$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot \mathrm{ld}n$	2	64	2048	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$
n^3	8	4096	16777200	$\approx 10^9$	$\approx 10^{18}$
2^n	4	65536	$\approx 10^{77}$	$\approx 10^{308}$	$\approx 10^{315653}$

[addp]

Abb. 2. Wachstum für ausgewählte Komplexitätsklassen

Die Polynomiellen Klassen sind auch noch für großes *n* mit "schnelleren" Rechnern (z.B. durch Parallelisierung) in akzeptabler Zeit lösbar, wo hingegen exponentielle Laufzeit auch nicht mehr durch schnellere Rechner oder Parallelisierung lösbar sind.

Komplexitätsklassen

Recht illustrativ ist es auch, sich vor Augen zu führen, welche Größenordnung ein Problem haben kann, wenn man die Zeit durch eine maximale Dauer *T* begrenzt. Wir nehmen dazu an, dass ein Schritt des Aufwandes genau eine Mikrosekunde (1 µs) dauert. *G* sei dabei das größte lösbare Problem in der Zeit *T*. Den Aufwand $g(n) = log\ n$ lassen wir weg, da man hier praktisch unbegrenzt große Probleme in vernünftiger Zeit bearbeiten kann.

G	T = 1 Min.	1 Std.	1 Tag	1 Woche	1 Jahr
n	$6 \cdot 10^7$	$3,6 \cdot 10^9$	$8,6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3 \cdot 10^{13}$
n^2	7750	$6 \cdot 10^4$	$2.9 \cdot 10^5$	$7.8 \cdot 10^5$	$5,6 \cdot 10^6$
n^3	391	1530	4420	8450	31600
2^n	25	31	36	39	44

[addp]

Abb. 3. Zeitaufwand für einige Problemgrößen

Problemklassen

Aufwand	Problem
O(1)	Einfache Suchverfahren mit Hash Tabellen
O(log n)	Allgemeine Suchverfahren für Tabellen (Baum-Suchverfahren)
O(n)	sequenzielle Suche, Suche in Texten, syntaktische Analyse von Programmen, einfache Vektoroperationen
O(n log n)	Sortieren, Suche
$O(n^2)$	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume), Multiplikation Matrix-Vektor (einfach)
$O(n^3)$	Matrizen-Multiplikation (einfach)
$O(2^n)$	viele Optimierungsprobleme (z.B. optimale Schaltwerke), automatisches Beweisen (im Prädikatenkalkül 1. Stufe)

Tab. 2. Typische Problemklassen

NP versa P-vollständig

P-vollständig Gruppe

Der Algorithmus löst maximal in polynomieller Laufzeit (aber beliebiger Polynomgrad)

NP

Der Algorithmus löst maximal in exponentieller Laufzeit (aber beliebige Basis), eventuell auch in P Laufzeit.

- Offensichtlich gehört jedes Problem aus P auch NP an, d.h., es gilt $P \subseteq NP$.
- Es gibt eine Reihe von Problemen, von denen man weiß, dass sie NP angehören, aber nicht, ob sie auch P angehören.
  - Das bedeutet, dass sie mithilfe eines nichtdeterministischen Algorithmus gelöst werden können, aber es noch nicht gelungen ist, einen effizienten deterministischen Algorithmus zu finden.

Es ist – streng genommen – ein offenes Problem, ob sich die exponentiellen Probleme nicht doch mit polynomialem Aufwand lösen lassen, d.h., ob P=NP gilt. Es wäre jedoch eine große Überraschung, wenn sich dies herausstellen sollte. Man kann allerdings beweisen, dass es eine Klasse von verwandten Problemen aus NP gibt, die folgende Eigenschaft aufweisen: Falls eines dieser Probleme in polynomialer Zeit mit einem deterministischen Algorithmus gelöst werden könnte, so ist dies für alle Probleme aus NP möglich (demzufolge wäre P=NP). Man bezeichnet Probleme mit dieser Eigenschaft als NP-vollständig. Ein bekanntes Problem aus dieser Klasse ist das Problem des Handlungsreisenden.

NP versa P-vollständig

Aber:

Diese Annahmen und Aussagen gelten nur für "handelsübliche" Rechnermodelle, also der Turing- oder Registermaschine (von-Neumann Rechner). Wie sich das bei völlig andersartigen Rechnermodellen (z.B. Quantencomputer) entwickelt ist noch unklar.

Analyse von Algorithmen

Wie kann nun die Laufzeitkomplexität eines Algorithmus bestimmt werden? Die Probleme einer exakten Analyse haben wir bereits diskutiert. Die Bestimmung der Größenordnung ist dagegen wesentlich einfacher, da sich hierfür einige einfache Regeln angeben lassen:

1. Zählschleifen

```
for(i=a;i<b;i=i+c) { * }
```

Def. 1. Laufzeit := max. Laufzeit der inneren Anweisung · Anzahl der Iterationen (b-a)/c

2. Geschachtelte Zählschleifen

```
for(i=a1;i<b1;i=i+c1) for(j=a2;j<b2;j=j+c2) { * }
```

Def. 2. Laufzeit := max. Laufzeit der inneren Anweisung · Produkt der Durchläufe aller Schleifen, d.h. $\Pi_i(b_i-a_i)/c_i$

Analyse von Algorithmen

3. Nacheinanderausführung

```
for(i=a;i<b;i=i+c) { * } ; for(j=a2;j<b2;j=j+c2) { * }
```

Def. 3. Bei einer Sequenz von Anweisungen werden die Laufzeiten zunächst addiert. Da konstante Faktoren weggelassen werden können und nur die jeweils höchsten Exponenten berücksichtigt werden, wird für die Gesamtlaufzeit nur die maximale Laufzeitkomponente verwendet.

4. Bedingte Verzweigungen

```
if (cond) { A1 } else { A2 }
```

Def. 4. Laufzeit := Aufwand für Test + max(Aufwand für A1, Aufwand für A2)

5. Funktionsaufrufe (Rekursion)

```
function foo(x) { return terminatecond(x)?C:F(foo(G(x))) }
```

Def. 5. Laufzeit := max. Laufzeit der inneren Anweisung · Anzahl der Iterationen

Landau Symbole

But wait, there is more...

[Wikipedia]

Notation	Anschauliche Bedeutung (immer ohne Berücksichtigung von Konstanten)
$f = o(g)$ oder $f \in o(g)$	f wächst langsamer als g
$f = O(g)$ oder $f \in \mathcal{O}(g)$	f wächst höchstens genauso schnell wie g
$f \in \Theta(g)$	f wächst genauso schnell wie g
$f = \Omega(g)$	f wächst nicht immer langsamer als g (Zahlentheorie)
$f \in \Omega(g)$	f wächst mindestens genauso schnell wie g (Komplexitätstheorie)
$f \in \omega(g)$	f wächst schneller als g

Abb. 4. Es gibt neben der O-Notation noch weitere Funktionen die unterschiedliche Schranken definieren

Little versa Big O

Big-O- und Little-O-Notationen haben sehr ähnliche Definitionen, und ihr Unterschied liegt darin, wie streng sie in Bezug auf die von ihnen dargestellte Obergrenze sind.

Big O

Engerer Abstand zwischen f und g

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

Little o

Großzügigerer Abstand zwischen f und g

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$$