Algorithmen und Datenstrukturen

Analyse von Algorithmen

Prof. Justus Piater, Ph.D.

6. März 2024

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Einführung

Viele Datenverarbeitungsprobleme, wie zum Beispiel das Sortieren einer Zahlensequenz, lassen sich auf unterschiedliche Weise lösen, d.h. mit unterschiedlichen Algorithmen. Manche dieser Algorithmen sind effizienter als andere. In diesem Kapitel beschäftigen wir uns mit der Frage, wie man die Effizienz von Algorithmen charakterisieren kann.

Die Effizienz eines Algorithmus bemisst sich unter anderem an seinem *Zeitbedarf*. Wir können hier einmal die Laufzeit eines Algorithmus als Funktion seiner Eingabegröße darstellen.

Ein typisches Laufzeitverhalten ist *linear*: Wenn ich die Eingabegröße um eins erhöhe, dann erhöht sich die Laufzeit des Algorithmus um einen konstanten Wert.

Vielleicht ist mein Algorithmus effizienter: Seine Laufzeit vergrößert sich um einen konstanten Wert, wenn ich die Eingabegröße um einen konstanten *Faktor* erhöhe. Dies wäre ein *logarithmisches* Laufzeitverhalten. Zum Beispiel nehme ich doppelt so viele Datenelemente als Eingabe, und die Laufzeit erhöht sich um eine Sekunde.

Die effizientesten Algorithmen haben ein *konstantes* Laufzeitverhalten: Deren Laufzeit ist *unabhängig* von der Eingabegröße.

Bei weniger effizienten Algorithmen wächst die Laufzeit z.B. mit dem *Quadrat* der Eingabegröße: Wenn ich doppelt so viele Elemente eingebe, dann vervierfacht sich die Laufzeit.

Oder, bei einem *exponentiellen* Laufzeitverhalten wächst die Laufzeit bei einer Vergrößerung der Eingabegröße um eins um einen konstanten *Faktor*, nämlich um die Basis des Exponenten.

Ein gutes Verständnis dieser Wachstumsraten ist unabdingbare Voraussetzung für die korrekte Nutzung und die Entwicklung effizienter Algorithmen.

1 Ressourcenbedarf

Video 1 beginnt hier.

Ressourcenbedarf [Slide 1]

Welche Ressourcen nimmt mein Programm in Anspruch:

- Zeit (CPU, etc.)
- Raum (RAM, etc.)
- andere (offene Datei-Deskriptoren, …)

Wie finden wir dies heraus?

Mein „Programm“:

- Algorithmus
- seine Implementation

Erklärung

Bei der Effizienz geht es in erster Linie um die Laufzeit eines Programms, aber nicht nur. Auch andere Ressourcen sind von Interesse, beispielsweise der Speicherbedarf oder andere Ressourcen, die vom Betriebssystem zur Verfügung gestellt werden.

Wenn wir hier informell von einem Programm sprechen, dann interessieren wir uns für den Algorithmus, d.h., für die Abfolge der einzelnen Schritte, die das Programm ausführt. Details der Implementierung können ebenfalls einen Einfluss auf den Ressourcenbedarf haben, beispielsweise die Wahl der Programmiersprache oder ob auf Arrays über Pointer oder über Indizes zugegriffen wird. Dieser Einfluss fällt jedoch im Vergleich zur Wahl des Algorithmus kaum ins Gewicht, wie wir noch sehen werden.

Unter den verschiedenen Ressourcen ist die Laufzeit die wichtigste. Insbesondere bildet der Zeitbedarf eine obere Schranke für den Platzbedarf. Warum das so ist, darüber dürfen Sie selber nachdenken. Daher werden wir uns hier auf die Analyse des sogenannten Laufzeitverhaltens beschränken.

Wie können wir das Laufzeitverhalten eines Algorithmus analysieren? Hier bieten sich zwei Methoden an: Wir können den Algorithmus implementieren und seine Laufzeit empirisch, also experimentell ermitteln. Oder wir können den Algorithmus theoretisch analysieren. Im Folgenden werden wir uns beide Methoden anschauen.

2 Empirische Laufzeitanalyse

Zeitmessung [Slide 2]

```
long startTime = System.currentTimeMillis(); // record the starting time
/* run the algorithm */
long endTime = System.currentTimeMillis(); // record the ending time
long elapsed = endTime - startTime;      // compute the elapsed time
```

Erklärung

Unter empirischer Laufzeitanalyse versteht man das Messen der Laufzeit des Programs als Funktion der Problemgröße. Dafür werden die Start- und Endzeit vor und nach dem zu messenden Code gespeichert, und die Differenz ergibt die empirische Laufzeit.

Iterative Erweiterung von Zeichenketten [Slide 3]

```
/** Uses repeated concatenation to compose a String
    with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j = 0; j < n; j++)
    answer += c;
  return answer;
}

/** Uses StringBuilder to compose a String
    with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int j = 0; j < n; j++)
    sb.append(c);
  return sb.toString();
}
```

Beispiel

Als Beispiel betrachten wir zwei verschiedene Java-Implementierungen zur iterativen Erweiterung von Zeichenketten. `repeat1()` beginnt mit einer leeren Zeichenkette und hängt mittels des +=-Operators in jeder Iteration ein weiteres Zeichen an. `repeat2()` verwendet dagegen einen `StringBuilder` und fügt in jeder Iteration mit Hilfe der append-Funktion ein weiteres Zeichen an. Welche Implementierung ist wohl besser?

Die relevante Problemgröße ist hier die Anzahl `n` der anzuhängenden Zeichen.

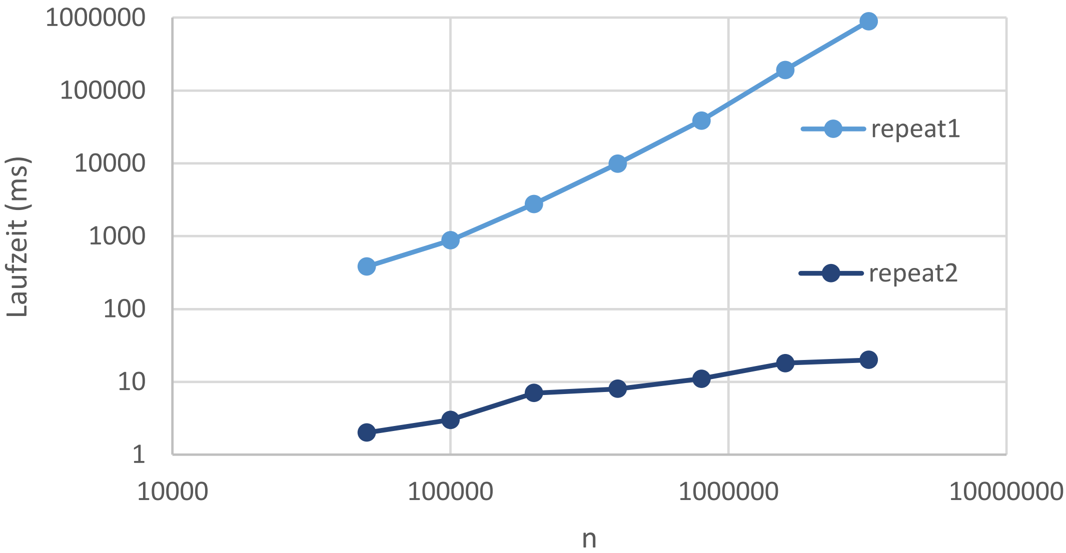
Laufzeiten [Slide 4]

n	`repeat1` [ms]	`repeat2` [ms]
50.000	382	2
100.000	876	3
200.000	2.769	7
400.000	9.875	8
800.000	38.633	11
1.600.000	191.121	18
3.200.000	894.805	20

Q: Damit `repeat1` so schnell läuft wie jetzt `repeat2`, benötige ich einfach einen entsprechend schnelleren Prozessor!

A: richtig; **B:** falsch; **D:** weiß nicht

Laufzeiten [Slide 5]



Beispiel

Vergleichen wir nun die Laufzeiten der zwei Implementierungen abhängig von der Problemgröße n. Man kann sofort erkennen, dass `repeat2()` schneller ist und auch bei großem n nicht in die Knie gezwungen wird. In diesem Graphen sind beide Achsen exponentiell zur Basis 10 skaliert. So sieht man gut, dass, wenn man n um den Faktor 10 erhöht, sich auch die Laufzeit von `repeat2()` etwa um den Faktor 10 erhöht. Die Laufzeit von `repeat2()` ist also etwa proportional zu n. Die Laufzeit von `repeat1()` erhöht sich dabei hingegen etwa um den Faktor 100; die Laufzeit von `repeat1()` ist also quadratisch in n.

Aber woher kommt dieser große Unterschied? Dafür werfen wir noch einmal einen Blick auf die Implementierungen und überlegen, wie Java mit Zeichenketten umgeht. `repeat1()` erzeugt bei jeder Iteration eine neue Zeichenkette. Diese ist eine Kopie der alten Zeichenkette ergänzt um den neuen Buchstaben. Für jedes Anhängen eines Zeichens muss also die gesamte Zeichenkette kopiert werden. Da die Zeichenkette sich mit jeder Iteration um eins verlängert, sind die Gesamtkosten dieser Kopien also linear in der Summe der ganzen Zahlen von 1 bis n. Diese Summe ist bekanntlich gleich $n(n+1)/2$, also quadratisch in n.

Im Gegensatz dazu vermeidet `repeat2()` mithilfe des StringBuilders die meisten dieser Kopien, und erzielt so ein lineares Laufzeitverhalten. Wie der StringBuilder dies bewerkstelligt, werden wir in Kürze in diesem Kurs sehen.

Woher diese Diskrepanz? [Slide 6]

- `repeat1` erzeugt bei jedem `+=` einen neuen String, dessen Inhalt vom alten kopiert wird.

  Dies führt zu einem quadratischen Laufzeitverhalten in n.

  Jedes Hinzufügen hat Kosten, die der aktuellen Stringlänge entsprechen:

  $$1+2+3+\cdots+n=\sum_{i=1}^{n} i=\frac{1}{2}n(n+1)=\frac{1}{2}n^2+\frac{1}{2}n$$

- `repeat2` vermeidet die Zusatzkosten dieses Kopierens.

  Wie, sehen wir später.

  Das Laufzeitverhalten ist linear in n.

  Die Kosten für jedes Einfügen bleiben konstant:

  $$1+1+\cdots+1=\sum_{i=1}^{n} 1=n$$

Algorithmen... [Slide 7]

Finde das größte Element einer Sequenz.

Algorithmus 1

1. Sortiere die Elemente in aufsteigender Reihenfolge.
2. Liefere das letzte Element zurück.

Algorithmus 2

1. Durchlaufe die Sequenz, und speichere das größte Element.
2. Liefere das gespeicherte Element zurück.

Q: Welcher Algorithmus ist effizienter?

A: Algorithmus 1; **B:** Algorithmus 2; **C:** kaum ein Unterschied; **D:** weiß nicht

… und Implementationen [Slide 8]

In C:

```
long max_index(const long *v, unsigned long n) {
  long vmax = v[0];
  for (unsigned long i = 1; i < n; i++)
    if (v[i] > vmax)
      vmax = v[i];
  return vmax;
}

long max_pointer(const long *v, unsigned long n) {
  long vmax = *v;
  for (const long *pv = v + 1, *pvn = v + n; pv < pvn; pv++)
    if (*pv > vmax)
      vmax = *pv;
  return vmax;
}
```

`max_pointer()` ist etwas schneller, aber nur bei ausgeschalteter Optimierung.

Bei komplexeren Array-Operationen können Pointer-basierte Implementationen jedoch auch mit Optimierung deutlich schneller sein als Index-basierte.

Quiz [Slide 9]

Was ist im Allgemeinen wichtiger für die praktische Laufzeit, die Effizienz des Algorithmus oder die Effizienz des implementierenden Codes?

A: Algorithmus

B: Code

C: beide gleichermaßen

D: weiß nicht

Grenzen empirischer Laufzeitbestimmung [Slide 10]

- Empirische Laufzeit hängt von vielen Faktoren ab (Hardware, Betriebssystem, andere Prozesse, etc.)
- Nur punktuelle Tests, keine volle Charakterisierung
- Keine Informationen über die Ursachen des beobachteten Laufzeitverhaltens
- Algorithmus muss implementiert werden

Erklärung

Empirische Laufzeitbestimmung ist konzeptuell einfach, aber hat ihre Grenzen. Die Laufzeitmessung hängt von vielen Faktoren ab wie Hardware, Betriebssystem und anderen Prozessen. Es handelt sich nur um punktuelle Tests, und die Laufzeit kann nicht allgemein für alle Problemgrößen charakterisiert werden. Sie gibt uns keine Informationen über die Ursachen des beobachteten Laufzeitverhaltens. Überhaupt muss der Algorithmus implementiert werden, um ihn empirisch messen zu können. Dies ist in der Praxis ein entscheidender Nachteil.

Unabhängig von Maschine und Implementierung funktioniert die asymptotische Laufzeitanalyse, die wir im nächsten Video genauer betrachten werden.

3 Zählen primitiver Operationen

Video 2 beginnt hier.

Komplexität [Slide 11]

Wir werden hauptsächlich das *Laufzeitverhalten* von *Algorithmen* analysieren.

Diese bildet eine obere Schranke für den Speicherbedarf.

Jeder sinnvoll reservierte Speicherbereich muss auch beschrieben und gelesen werden; das kostet Zeit.

Das Laufzeitverhalten von Algorithmen dominiert den Einfluss der Implementation.

Primitive Operationen [Slide 12]

```
Algorithm arrayMax(A,n):
  Require: An array A storing n ≥ 1 integers.
  Ensure: Return the maximum element in A.
  m ← A[0]
  for i ← 1 to n − 1 do
    if m < A[i] then
      m ← A[i]
  return m
```

- Methodenaufruf; Rückkehr von einer Methode
- Zuweisung
- arithmetische, logische, etc. Operation
- Arrayzugriff per Index

  Anmerkung
  Indexberechnung erfordert arithmetische Operationen.

Erklärung

Wie können wir das Laufzeitverhalten von Algorithmen bestimmen, ohne sie vorher programmieren zu müssen? Nehmen wir an, jede primitive Operation entspreche einer Zeiteinheit. Als eine primitive Operation zählt z.B. ein Methodenaufruf, die Rückkehr von einer Methode, Zuweisung, Arrayzugriff, arithmetische und logische Operation. Dann können wir im Prinzip primitive Operationen zählen.

Zählen primitiver Operationen [Slide 13]

Algorithm `arrayMax`(A,n):
Require: An array A storing $n \geq 1$ integers.
Ensure: Return the maximum element in A.
$m \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $m < A[i]$ **then**
    $m \leftarrow A[i]$
return m

Anzahl primitiver Operationen

im besten Fall:
im schlechtesten Fall:
im Erwartungsfall:

Beispiel

Im Beispiel `arrayMax()` sind das für den schlechtesten Fall: 1 (für die Indizierung) + 1 (für die Zuweisung) + 1 (für die Initialisierung von i) + 1 (für die Berechnung von $n-1$) + $(n-1)$ (Iterationen der for-Schleife) *

1 (für die Indizierung $A[i]$) + 1 (für den Vergleich mit m) + 1 (für die Zuweisung an m) + 1 (für das Inkrementieren der Laufvariablen i) + 1 (für den Vergleich von i mit $n-1$)

+ 1 (für den Rücksprung von der Methode) $= 4+(n-1)\cdot 5+1 = 5n$ primitive Operationen.

Zählen primitiver Operationen [Slide 14]

Algorithm `arrayMax`(A,n):
Require: An array A storing $n \geq 1$ integers.
Ensure: Return the maximum element in A.
$m \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $m < A[i]$ **then**
    $m \leftarrow A[i]$
return m

Anzahl primitiver Operationen

im besten Fall:
im schlechtesten Fall: $5n$
im Erwartungsfall:

Beispiel

Im besten Fall wird die vom if abhängige Zuweisung nie ausgeführt. Damit reduziert sich die Anzahl der primitiven Operationen um $n-1$ auf insgesamt $4n+1$.

Zählen primitiver Operationen [Slide 15]

```
Algorithm arrayMax(A,n):
  Require: An array A storing n ≥ 1 integers.
  Ensure: Return the maximum element in A.
  m ← A[0]
  for i ← 1 to n − 1 do
    if m < A[i] then
      m ← A[i]
  return m
```

Anzahl primitiver Operationen

im besten Fall:	$4n+1$
im schlechtesten Fall:	$5n$
im Erwartungsfall:	

Beispiel

Der Erwartungsfall wird irgendwo dazwischen liegen, zwischen $4n+1$ und $5n$ primitiven Operationen.

Welche dieser Zahlen sind uns wichtig? [Slide 16]

Bester Fall: meist weitgehend uninteressant

Erwartungsfall: Interessant, aber oft schwierig zu ermitteln

Schlechtester Fall:

- Relevant und oft einfach bestimmbar
- garantierte Obergrenze für *jede* Eingabe
- Optimierung führt oft zu besseren Algorithmen

Erklärung

Welche Fälle sind für uns tatsächlich wichtig, der beste, der schlechteste, oder der Erwartungsfall?

Der beste Fall ist meist weitgehend uninteressant, da er ggf. weit unter dem Erwartungsfall liegen kann. Der schlechteste Fall ist dagegen sehr relevant und oft einfach bestimmbar. Er dient als garantierte Obergrenze für *jede* Eingabe. Es zahlt sich also aus, bei der Entwicklung von Algorithmen den schlechtesten Fall zu optimieren.

Der Erwartungsfall beschreibt Situationen, wie sie in der Praxis typischerweise auftreten werden. Falls er deutlich unter dem schlechtesten Fall liegt, ist es wertvoll, ihn ebenfalls zu optimieren. Allerdings ist er oft schwieriger zu ermitteln.

Quiz [Slide 17]

Die Optimierung welcher dieser Größen (in Abhängigkeit von der Problemgröße) ist meist am effektivsten?

A: Die Anzahl primitiver Operationen im schlechtesten Fall

B: Die Anzahl primitiver Operationen im Erwartungsfall

C: Die Anzahl primitiver Operationen im besten Fall

D: weiß nicht

Fazit [Slide 18]

- Wir können im Prinzip durch Zählen der primitiven Operationen den Ressourcenbedarf eines Algorithmus als Funktion $f(n)$ der Problemgröße n hinschreiben.
- Die genaue Anzahl und Dauer der primitiven Operationen ***bestimmen die Koeffizienten*** von $f(n)$, jedoch ***nicht die funktionale Form*** von $f(n)$.
- Die genaue Anzahl und Dauer der primitiven Operationen ist für uns kaum zu ermitteln.
- Beschränken wir uns bei der Analyse auf die funktionale Form, und verzichten auf das Bestimmen der Koeffizienten!

Erklärung

Beim Zählen der primitiven Operationen sind wir sehr nachlässig vorgegangen. Was genau ist eine primitive Operation? Zählt das Laden eines Wertes in ein Prozessor-Register dazu? Dauern alle diese Operationen tatsächlich gleich lange?

Damit ist unklar, ob die Laufzeit von `arrayMax()` tatsächlich $5n$ Zeiteinheiten beträgt. Vielleicht sind es in Wirklichkeit $5n + 8$, oder $7n + 6$ Zeiteinheiten. Diese Konstanten können wir nicht wirklich bestimmen; sie sind damit bedeutungslos. Was hingegen sehr bedeutsam ist, ist die Tatsache, dass die Laufzeit linear in n ist. Egal, wie wir diese Konstanten wählen – das Ergebnis ist immer eine lineare Funktion in n.

Wir können also auf das Zählen primitiver Operationen komplett verzichten. Das vereinfacht unsere Analyse drastisch, ohne ihre Aussagekraft zu beeinträchtigen.

Wir werden also den Ressourcenbedarf von Algorithmen weitgehend unabhängig von multiplikativen und additiven Konstanten charakterisieren, und uns darauf konzentrieren, wie häufig *irgendetwas* in Abhängigkeit von der Problemgröße n passiert. Insbesondere gilt unser Interesse der Wachstumsrate des Ressourcenbedarfs als Funktion von n.

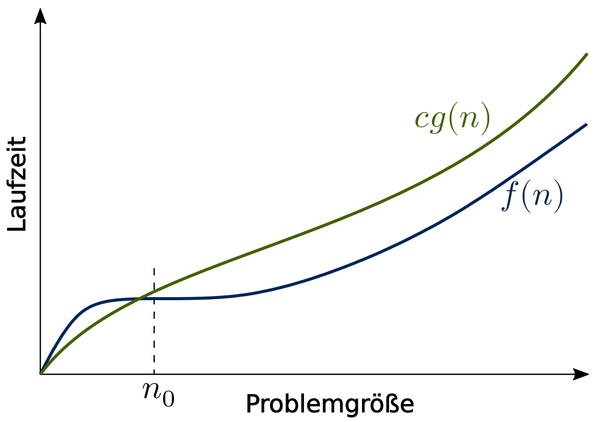
4 Groß-O Notation

Groß-O *(Big-Oh)* [Slide 19]

Video 3 beginnt hier.

Definition: Seien $f(n)$ und $g(n)$ Funktionen, die nicht-negative ganze Zahlen auf nicht-negative reelle Zahlen abbilden. $f(n) \in O(g(n))$ falls eine reelle Konstante $c > 0$ und eine ganzzahlige Konstante $n_0 \geq 1$ existieren, so dass

$f(n) \leq cg(n) \quad \forall n \geq n_0.$



Lies: „f von n ist Groß-O von g von n".

Intuition: $f(n)$ besitzt keine höhere Wachstumsrate (ist nicht stärker nach oben gekrümmt) als $g(n)$.

Beispiele:

- $f(n) = 2n$ und $g(n) = n$ wachsen mit derselben – nämlich mit konstanter – Rate (sie sind gleich stark nach oben gekrümmt, nämlich gar nicht).
- $f(n) = 3n^2$ wächst mit geringerer Rate (ist schwächer nach oben gekrümmt) als $g(n) = n^3$, denn ab einem bestimmten n (nämlich $n = 3$) wird $f(n)$ $g(n)$ nicht mehr übersteigen.

Definition

Die wichtigste Methode zur Analyse des Ressourcenbedarfs von Algorithmen ist die sogenannte Groß-O-Charakterisierung, auf englisch Big-Oh.

Im Folgenden bezeichnet n die Problemgröße, also z.B. bei `arrayMax()` die Größe des Arrays, und $f(n)$ bezeichnet den Ressourcenbedarf als Funktion der Problemgröße, also z.B. das Laufzeitverhalten. Wir characterisieren nun die im Detail unbekannte Funktion $f(n)$ mittels einer sehr einfachen Funktion $g(n)$ unserer Wahl, mit der wir $f(n)$ für große n nach oben beschränken können. Sehen wir uns die Definition an:

Seien $f(n)$ und $g(n)$ Funktionen, die nicht-negative ganze Zahlen auf nicht-negative reelle Zahlen abbilden. $f(n) \in O(g(n))$ falls eine reelle Konstante $c > 0$ und eine ganzzahlige Konstante $n_0 \geq 1$ existieren, so dass $f(n) \leq c \cdot g(n) \forall n \geq n_0$.

So drücken wir aus, dass $f(n)$ nicht schneller wächst als $g(n)$ (bis auf einen beliebigen konstanten Faktor c).

Und wir interessieren uns nur für große Problemgrößen n. Bei kleinen Problemgrößen, nämlich für $n < n_0$, darf $f(n)$ $c \cdot g(n)$ gerne übersteigen.

Die Konstanten n_0 und c dürfen wir frei wählen. Gelingt es uns, diese so zu wählen, dass $c \cdot g(n)$ für alle $n \geq n_0$ mindestens so groß ist wie $f(n)$, dann ist $f(n) \in O(g(n))$. Dies ist genau dann der Fall, wenn die Wachstumsrate von $f(n)$ nicht größer ist als die

Wachstumsrate von $g(n)$. Intuitiv ist $g(n)$ mindestens so stark nach oben gekrümmt wie $f(n)$.

Beispiel

Zum Beispiel ist $2n \in O(n)$, mit $n_0 = 1$ und $c = 2$. Beide sind gleich stark nach oben gekrümmt, nämlich gar nicht.

$3n^2$ ist schwächer nach oben gekrümmt als n^3, denn ab einem bestimmten Wert von n wird $3n^2$ n^3 nicht mehr übersteigen. $3n^2$ ist also $O(n^3)$.

$3n^2$ ist sogar $O(n^2)$, wie wir mit $n_0 = 1$ und $c = 3$ sofort sehen.

Erklärung

Mit der Notation $f(n)$ ist Element von $O(g(n))$ betrachten wir $O(g(n))$ als eine Menge, nämlich die Menge aller Funktionen $f(n)$, die die Eigenschaft gemäß dieser Definition besitzen. Verbal behandeln wir dagegen $O(g(n))$ als eine Eigenschaft von $f(n)$. Wir schreiben also $f(n)$ ist *Element der Menge* $O(g(n))$, und wir lesen $f(n)$ *ist* $O(g(n))$.

Beispiel [Slide 20]

Proposition:

$f(n) = 8n + 4 \in O(n)$.

Beweis (durch vollständige Induktion): Sei $c = 9$ und $n_0 = 4$.

Induktionsanfang: $8 \cdot 4 + 4 = 36 \leq 36 = 9 \cdot 4$

Induktionsschritt: Angenommen, $8n + 4 \leq 9n$ für einen bestimmten Wert von n. Dann gilt diese Ungleichung auch für $n + 1$:

$$8(n+1) + 4 \leq 9(n+1)$$
$$8n + 4 + 8 \leq 9n + 9$$

Es gibt i.d.R. unendlich viele mögliche Beweise, da c und n_0 voneinander abhängen.

In der Praxis ist ein Induktionsbeweis oft nicht nötig, wenn offensichtlich ist, dass das, was für n_0 gilt, ebenfalls für alle $n > n_0$ gilt. Hier ist dies dadurch offensichtlich, dass wir $c = 9$ als Faktor von n gewählt haben, was größer ist als der Faktor 8 in $f(n)$.

Beispiel

Betrachten wir nun ein ausführliches Beispiel.

Wir wollen zeigen, dass $f(n) = 8n + 4 \in O(n)$ ist. Hierzu wählen wir c und n_0 so, dass $8n_0 + 4 \leq cn_0$.

c können wir durch Koeffizientenvergleich bestimmen: Wir sehen, dass $8n_0$ und cn_0, beides lineare Terme in n_0, die am schnellsten wachsenden Terme auf beiden Seiten der Gleichung sind. Wir wählen nun $c = 9$, um eins größer als den korrespondierenden Faktor auf der linken Seite, damit der additive Term $+4$ die linke Seite nicht über die rechte hinauswachsen lässt, jedenfalls für $n \geq n_0$.

Einen passenden Wert für n_0 können wir nun leicht finden, indem wir die Ungleichung nach n_0 auflösen. Wir subtrahieren auf beiden Seiten $8n_0$, und erhalten $n_0 \geq 4$.

Nun müssen wir noch beweisen, dass diese Ungleichung nicht nur für n_0 gilt, sondern auch für alle $n > n_0$. Dies können wir durch vollständige Induktion tun. Den Induktionsanfang haben wir mit der Ungleichung für n_0 bereits abgehakt. Im Induktionschritt nehmen wir an, dass die Ungleichung $8n + 4 \leq 9n$ für einen beliebigen Wert von $n \geq n_0$ zutreffe, und zeigen dann, dass daraus folgt, dass sie damit auch für $n + 1$ zutreffen muss. Wir tun dies, indem wir n durch $n + 1$ ersetzen, ohne dass die Ungleichung verletzt wird. Da wir die Induktionsannahme im Induktionsanfang nachgewiesen haben, muss die Ungleichung für alle $n \geq n_0$ gelten. Damit ist der Beweis beendet.

> Es gibt in der Regel unendlich viele mögliche Beweise, da c und n_0 voneinander abhängig frei gewählt werden können.
>
> Wie geht man in der Praxis vor? Hier haben wir c um eins größer gewählt als den Koeffizienten des am schnellsten wachsenden Terms von $f(n)$, nämlich die 8. Damit haben wir sichergestellt, dass $g(n)$ schneller wächst als $f(n)$. Nun mussten wir lediglich noch n_0 so bestimmen, dass $c \cdot g(n_0)$ tatsächlich größer ist als $f(n_0)$.
>
> Indem wir durch die Wahl von c sicherstellen, dass $g(n)$ schneller wächst als $f(n)$, können wir uns üblicherweise sparen, dies durch den Induktionsschritt erneut zu beweisen.

Quiz [Slide 21]

Groß-O ist Blödsinn. Nehmen wir doch einmal diese beiden Funktionen:

$$\begin{aligned} 100000n &\in O(n) \\ n^2 &\in O(n^2) \end{aligned}$$

Dabei ist $100000n$ doch viel größer als n^2!

A: Stimmt. Hier ist Groß-O sinnlos.

B: Stimmt nicht. Auch hier ist Groß-O aussagekräftig.

C: Mindestens eine der beiden o.g. Groß-O-Charakterisierungen ist falsch.

D: weiß nicht

Wichtige Eigenschaften von Groß-O [Slide 22]

Video 4 beginnt hier.

1. Wenn $f(n) \in O(g(n))$, dann $kf(n) \in O(g(n))$.

   Beweisskizze: k wird in c hineinmultipliziert.

   Insbesondere, $a^b \in O(1)$, $a^{n+b} \in O(a^n)$, und die Basis von Logarithmen wird weggelassen.

   a^b hängt nicht von n ab, $a^{n+b} = a^b a^n$, und da $\log_a n = \frac{1}{\log_b a} \log_b n$ (siehe unten), $O(\log_a n) = O(\log_b n)$.

2. Wenn $e(n), f(n) \in O(g(n))$, dann $e(n) + f(n) \in O(g(n))$.

   Beweis: Wächst o.B.d.A. $f(n)$ schneller als $e(n)$, dann ist $e(n) + f(n) < 2f(n)$.

   Zusammen folgt insbesondere, dass $f(n) = \sum_{d=0}^{D} a_d n^d \in O(n^D)$.

   Wichtig

   Ein Polynom D-ten Grades in n ist immer $O(n^D)$.

3. Wenn $e(n) \in O(g(n))$ und $f(n) \in O(h(n))$, dann $e(n)f(n) \in O(g(n)h(n))$.

   Beweisskizze:

   $c_{ef} = c_e c_f$; $n_{0,ef} = \max\{n_{0,e}, n_{0,f}\}$

   Dies ist keine Überraschung. Von Bedeutung ist hier, dass keine stärkere Aussage möglich ist.

Logarithmen-Spickzettel

- Per Definition ist $\log_b n = x$ genau dann, wenn $b^x = n$. Folglich sind $b^{\log_b n} = n$, $\log_b b^x = x$ und $\log_b b = 1$.
- Sei zusätzlich $\log_b m = y$, also $b^y = m$. Es folgt $nm = b^x b^y = b^{x+y}$, also $\log_b(nm) = x + y = \log_b n + \log_b m$.

  Logarithmen erlauben uns, Multiplikationen durch Additionen zu ersetzen!
- Logarithmen verschiedener Basen sind zueinander proportional:

$$\begin{aligned}
b^{cx} &= (b^c)^x \quad = n \\
\log_b n &= cx \\
\log_{b^c} n &= x \\
\frac{\log_b n}{c} &= \log_{b^c} n = x \\
a &= b^c \\
\frac{\log_b n}{\log_b a} &= \log_a n \ = x
\end{aligned}$$

  Folglich sind Logarithmen verschiedener Basen a und b durch konstante Faktoren der Form $\log_b a = \frac{1}{\log_a b}$ untereinander verbunden.
- $b^{\log_a n}$ ist eine Potenz von n:

$$b^{\log_a n} = b^{\frac{\log_b n}{\log_b a}} = \left(b^{\log_b n}\right)^{\frac{1}{\log_b a}} = n^{\frac{1}{\log_b a}}$$

  Insbesondere, falls $\log_b a < 1$, dann $n^{\frac{1}{\log_b a}} \notin O(n)$.

  Bei Algorithmen mit solcher Laufzeit ist i.d.R. $b > 1$. In diesem Fall ist $\log_b a < 1$ falls $a < b$. Umgekehrt bedeutet dies, dass $b^{\log_a n} \in O(n)$ falls $1 \leq b \leq a$.

Erklärung

Als Nächstes betrachten wir weitere, wichtige Eigenschaften von Groß-O. Alle diese Eigenschaften folgen recht direkt aus der Definition von Groß-O; Sie sollten diese Beweise selbstständig führen.

Erstens: Wenn $f(n) \in O(g(n))$, dann ist $kf(n) \in O(g(n))$ für eine beliebige Konstante k. Insbesondere ist $a^b \in O(1)$, und $a^{n+b} \in O(a^n)$ und die Basis von Logarithmen wird entsprechend weggelassen. Überlegen Sie sich, warum!

Zweitens: Wenn $e(n)$ und $f(n) \in O(g(n))$, dann ist auch die Summe von $e(n) + f(n) \in O(g(n))$.

Aus diesen beiden Eigenschaften folgt insbesondere, dass Polynome D-ten Grades immer $O(n^D)$ sind.

Drittens: Wenn $e(n) \in O(g(n))$ und $f(n) \in O(h(n))$, dann ist $e(n)f(n) \in O(g(n)h(n))$. Dies ist nicht weiter überraschend. Wir halten hier lediglich fest, dass keine stärkere Aussage möglich ist.

Quiz [Slide 23]

Welche ist die beste Charakterisierung von $f(n) = 3n^2 \log n$?

A: $f(n) \in O(n^2)$

B: $f(n) \in O(n^3)$

C: $f(n) \in O(n^2 \log n)$

D: $f(n) \in O(3n^2 \log n)$

Quiz [Slide 24]

Welche der folgenden Aussagen ist falsch:

A: $\log n^2 \in O(\log n)$

B: $3^{3n} \in O(3^n)$

C: $5n^4 + 3n^3 \in O(3n^4)$

D: weiß nicht

O(so klein wie möglich) [Slide 25]

Gefragt ist die *kleinste Menge* Groß-O, die die analysierte Funktion enthält, und zwar in ihrer *kompaktesten* Darstellung.

$f(n) = 4n^3 + 3n^2 \in ...$

- Inkorrekt:

  $O(1), O(n^2), O(\sin n)$

- Korrekt, aber nicht hilfreich:

  $O(n^4), O(2^n), O(4n^3), O(n^3 + n^2)$

  Punktabzug bei Hausübungen und Prüfungen!

- Korrekt und aussagekräftig:

  $O(n^3)$

Wichtig

In der Praxis genügt es meist,

- alle konstanten Faktoren und
- alle bis auf den am schnellsten wachsenden Term

zu eliminieren.

Aber: $2^{an} = (2^a)^n \notin O(2^n)$!

Erklärung

Damit Groß-O-Charakterisierungen aussagekräftig sind, ist es wichtig, dass wir jeweils die kleinste Menge Groß-O und ihre kompakteste Darstellung finden. Hier ist zum Beispiel die Funktion $f(n) = 4n^3 + 3n^2$ gegeben. Welche Groß-O-Charakterisierung ist aussagekräftig?

$O(1)$, $O(n^2)$ und $O(sinn)$ sind inkorrekt, da sie zu klein gewählt sind und $f(n)$ nicht darin enthalten ist.

$O(n^4)$ und $O(2^n)$ sind unnötig große Mengen; eine engere Charakterisierung von $f(n)$ ist möglich.

$O(4n^3)$ und $O(n^3+n^2)$ sind zwar korrekt, aber ihre Darstellung ist nicht kompakt. Beide sind nämlich mit der Menge $O(n^3)$ identisch.

Korrekt und aussagekräftig ist $O(n^3)$, da diese Menge minimal und kompakt dargestellt ist.

In der Praxis genügt es meist, alle konstanten Faktoren und alle bis auf den am schnellsten wachsenden Term zu eliminieren, um eine enge und kompakte Groß-O-Charakterisierung zu erhalten.

Aber Achtung z.B. bei Exponentialfunktionen: $2^{an} = (2^a)^n$, und das ist nicht $O(2^n)$!

Groß-O-Analyse von Algorithmen [Slide 26]

Ziel: Den Ressourcenbedarf ohne Zählen der primitiven Operationen ermitteln

Weg:

1. Wir ermitteln die funktionale Form des Ressourcenbedarfs $f(n)$ des Algorithmus.

   D.h., wie oft geschieht *irgendetwas* in Abhängigkeit von n?

2. Wir finden eine koeffizientenlose Funktion $g(n)$, so dass $f(n) \in O(g(n))$ den asymptotischen Ressourcenbedarf möglichst gut charakterisiert.

   Bei der Groß-O-Charakterisierung von $f(n)$ sind seine Koeffizienten ja weitgehend irrelevant!

Anmerkung

Mit Groß-O (oder verwandten Konzepten) charakterisiertes Laufzeitverhalten wird als ***asymptotische Laufzeit*** oder ***Laufzeit-Komplexität*** bezeichnet.

Erklärung

Wegen ihrer Konzentration auf den Verlauf der Funktion bei großen n wird die Groß-O-Analyse als asymptotisch bezeichnet. Der asymptotische Ressourcenbedarf eines Algorithmus wird auch seine asymptotische Komplexität genannt.

Die Groß-O-Notation ist unser wichtigstes Hilfsmittel zur Charakterisierung des Ressourcenbedarfs von Algorithmen. Da sie multiplikative und additive Konstanten nicht berücksichtigt, erübrigt sich die Analyse primitiver Operationen. Dies macht die Groß-O-Notation in der Praxis sehr einfach anwendbar. Nichtsdestotrotz macht sie starke Aussagen über den Ressourcenbedarf von Algorithmen in Abhängigkeit von der Problemgröße, wie wir nun an einigen Beispielen sehen werden.

Analyse: Einige einfache Fälle [Slide 27]

Video 5 beginnt hier.

Primitive Operationen: konstante Ausführungszeit, $O(1)$

Sequenzen von Operationen: Die Summe, d.h. das Maximum der Komplexitäten der Elemente der Sequenz.

Schleifen über n: $O(nf(n))$ wenn der Schleifenkörper $O(f(n))$ ist

for $i \leftarrow 1$ **to** n **do**
  $\ldots$

Schleifen mit exponentiellem Inkrement: $O(\log n)$

$i \leftarrow 1$
while $i \leq n$ **do**
  $i \leftarrow 2i$

Die entscheidende Frage: Wie häufig wird die Schleife (in Abhängigkeit von n) durchlaufen?

Erklärung

Betrachten wir nun diverse Beispielfälle zur Laufzeitanalyse.

Primitive Operationen laufen in konstanter Zeit; daher sind sie grundsätzlich $O(1)$. Die asymptotische Laufzeit von Sequenzen von Operationen ist die Summe der asymptotischen Laufzeiten der einzelnen Operationen. Aus der Polynom-Eigenschaft von Groß-O folgt, dass diese Summe gleich der maximalen asymptotischen Laufzeit aller Operationen der Sequenz ist.

Einfache Schleifen mit n Iterationen haben eine asymptotische Laufzeit von $O(n)$ mal der asymptotischen Laufzeit des Schleifenkörpers. Schleifen, deren Zähler exponentiell wächst, haben entsprechend eine asymptotische Laufzeit von $O(\log n)$ mal der asymptotischen Laufzeit des Schleifenkörpers.

Analyse: Geschachtelte Schleifen [Slide 28]

Identische Schleifen: Wie oben: $O(n^2)$. Drei geschachtelte Schleifen sind $O(n^3)$, etc.

```
for i ← 1 to n do
  for j ← 1 to n do
    ...
```

Unabhängige Schleifen:

$O(mn)$

```
for i ← 1 to m do
  for j ← 1 to n do
    ...
```

Inkrementierte Schleifen:

$O(n^2)$

```
for i ← 1 to n do
  for j ← 1 to i do
    ...
```

Erklärung

Bei geschachtelten Schleifen folgt das Laufzeitverhalten aus dem bereits Gesagten: Zwei geschachtelte Schleifen mit jeweils n Iterationen haben eine asymptotische Laufzeit von $O(n^2)$, drei eine von $O(n^3)$, usw., jeweils multiplizert mit der asymptotischen Laufzeit des innersten Schleifenkörpers.

Manchmal ist es hilfreich, ein Laufzeitverhalten anhand mehrerer Problemgrößen zu charakterisieren. Haben wir beispielsweise zwei geschachtelte Schleifen, die m-mal bzw. n-mal durchlaufen werden, wobei m und n nicht voneinander abhängen, dann ist die Gesamtlaufzeit folglich $O(mn)$.

Die hier gezeigte inkrementierte Schleife lässt sich übrigens sich nicht enger charakterisieren als $O(n^2)$, obwohl der innere Schleifenkörper weniger oft durchlaufen wird als n^2 mal. Es ist eine wertvolle Übung, dies formal nachzuweisen.

Analyse: `arrayMax` [Slide 29]

```
Algorithm arrayMax(A,n):
  Require: An array A storing n ≥ 1 integers.
  Ensure: Return the maximum element in A.
  m ← A[0]
  for i ← 1 to n − 1 do
    if m < A[i] then
      m ← A[i]
  return m
```

- Die Schleife wird $n-1$ Mal durchlaufen.
- Inner- und außerhalb der Schleife finden sich nur Operationen, die in konstanter Zeit ablaufen.
- Daher ist `arrayMax` $O(n)$.

Beispiel

Am besten festigen wir diese Erkenntnisse anhand eines bereits bekannten Beispiels: `arrayMax()` sucht das größte Element in einem Array A. Der Algorithmus besteht aus einer Sequenz von drei Operationen: einer Zuweisung, einer Schleife, und der return-Anweisung. Da die asymptotische Laufzeit einer Sequenz gleich der maximalen asymptotischen Laufzeit seiner Elemente ist, ist die asymptotische Laufzeit von `arrayMax()` gleich der asymptotischen Laufzeit der Schleife. Die Schleife wiederum besteht aus einer Sequenz verschiedener Operationen, deren Laufzeiten jedoch alle konstant sind. Daher bleibt die Feststellung, dass die Schleife $n-1$ mal durchlaufen wird. Somit ist `arrayMax()` $O(n)$.

Quiz [Slide 30]

$i \leftarrow 1$
while $i \leq n$ **do**
  $i \leftarrow 2i$
for $j \leftarrow 1$ **to** i **do**
  $n \leftarrow n + 1$

Was ist die asymptotische Laufzeit dieses Codesegments in Abhängigkeit von n?

A: $O(\log n)$

B: $O(n)$

C: $O(n \log n)$

D: $O(2^n)$

Analyse: `prefixAverages` [Slide 31]

Berechnen wir $A_i = \frac{1}{i+1} \sum_{j=0}^{i} X_j$ für $i = 0, ..., n-1$:

Algorithm prefixAverages1(X):
  Require: *An $n-$element array X of numbers.*
  Ensure: *Return an $n-$element array A of numbers such that*
      $A[i]$ is the average of elements $X[0], \ldots, X[i]$.
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $a \leftarrow 0$
    for $j \leftarrow 0$ **to** i **do**
      $a \leftarrow a + X[j]$
    $A[i] \leftarrow a/(i+1)$
  return A

- Es gibt zwei geschachtelte Schleifen des inkrementierten Typs.
- Alle anderen Operationen sind primitiv und konstanter Laufzeit.
- Daher ist die Gesamtlaufzeit $O\left(n^2\right)$.

Ein effizienter Algorithmus [Slide 32]

Algorithm prefixAverages2(X):
  Require: *An $n-$element array X of numbers.*
  Ensure: *Return an $n-$element array A of numbers such that*
    $A[i]$ is the average of elements $X[0], \dots, X[i]$.
  $s \leftarrow 0$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $s \leftarrow s + X[i]$
    $A[i] \leftarrow s/(i+1)$
  return A

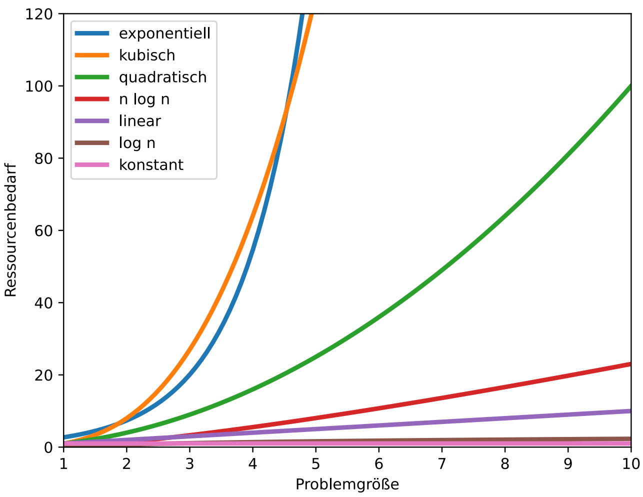
Laufzeit?

Eine Hierarchie von Funktionen [Slide 33]

Groß-O bezeichnet eine obere Schranke der *Wachstumsrate* von Funktionen.

Funktionen im mathematischen Sinn.

$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^{a>1}) \subset O(2^n)$



Erklärung

Welche Funktionen g(n) treten in der Praxis bei Charakterisierungen der Form $O(g(n))$ auf? Hier sehen wir die wichtigsten. $O(1)$ ist die konstante Laufzeit primitiver Operationen. $O(\log n)$ ist beispielsweise die Höhe einer ausgeglichenen Baumstruktur oder die Laufzeit einer Binärsuche. $O(n)$ ist die lineare Laufzeit einer Schleife über Datenelemente. $O(n \log n)$ ergibt sich aus einer Verschachtelung der beiden Letztgenannten, und ist die schnellstmögliche Laufzeit von Sortieralgorithmen, die auf paarweisen Vergleichen beruhen. $O(n^2)$ oder $O(n^3)$ tritt typischerweise bei zwei bzw. drei geschachtelten Schleifen auf, z.B. zum Aufzählen aller Paare bzw. Tripel. Exponentielle Laufzeiten von $O(2^n)$ ergeben sich beispielsweise bei der Aufzählung aller Kombinationen einer Menge von n Elementen. Algorithmen mit exponentieller Laufzeit lassen sich nur auf relativ kleine Datenmengen anwenden. Auch die Verwendung vielfach schnellerer Rechner ändert kaum etwas daran. Algorithmen mit logarithmischer Laufzeit gelten hingegen als effizient und lassen sich in der Praxis auf nahezu beliebig große Datenmengen anwenden.

Typische Laufzeit-Komplexitäten [Slide 34]

konstant: eine primitive Operation; eine Sequenz von Operationen unabhängig von den Eingangsdaten

logarithmisch: die Höhe eines ausgeglichenen Baums; binäre Suche

linear: Schleifen über Datenelemente

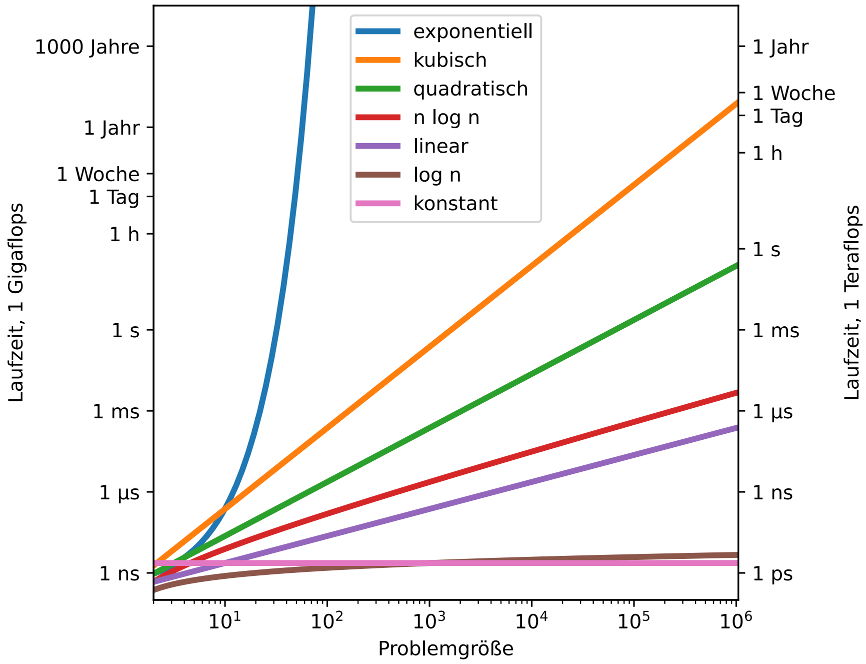
$n \log n$**:** Schachtelung des Obigen; Sortieren

quadratisch: zwei geschachtelte Schleifen; Aufzählen aller Paare

kubisch: drei geschachtelte Schleifen; Aufzählen aller Tripel

exponentiell: Kombinationen

Laufzeiten [Slide 35]



Wie große Probleme können wir innerhalb eines gegebenen Zeitbudgets lösen, z.B. 1 ns?

	1 Gflops	1 Tflops	Faktor 1000
$\log_2 n$	16	praktisch ∞	
n	4	4 000	Faktor 1000
n^2	2	63	Faktor $\sqrt{1000} \approx 32$
2^n	2	11	plus knapp 10

Bei exponentiellem Laufzeitverhalten bewirkt eine Vervielfachung der Rechenleistung lediglich eine *additive* Erweiterung der handhabbaren Problemgröße!

Hier z.B. vergrößert eine weitere Ver1000fachung der Rechenleistung die Problemgröße von 11 auf 21.

Mehr Ressourcen lösen größere Probleme? [Slide 36]

- *Ressource*: Laufzeit bei fixer Rechenleistung bzw. Rechenleistung bei fixer Laufzeit
- Ressourcenbedarf $r(n)$ als Funktion der Problemgröße n
- Umkehrfunktion: Lösbare Problemgröße $n(r)$ als Funktion der zur Verfügung stehenden Ressourcen

Wachstum der lösbaren Problemgröße, wenn wir die zur Verfügung stehenden Ressourcen um einen Faktor k erhöhen:

$r(n)$	$n(r)$	$n(kr)$			Wachstum
$\log_2 n$	2^r	2^{kr}	$=$	$(2^r)^k$	exponentiell
n	r	kr			multiplikativ mit Faktor k
n^2	$\sqrt{r}$	$\sqrt{kr}$	$=$	$\sqrt{k}\sqrt{r}$	multiplikativ mit Faktor $\sqrt{k}$
2^n	$\log_2 r$	$\log_2(kr)$	$=$	$\log_2 k + \log_2 r$	additiv

Groß-Omega und Groß-Theta [Slide 37]

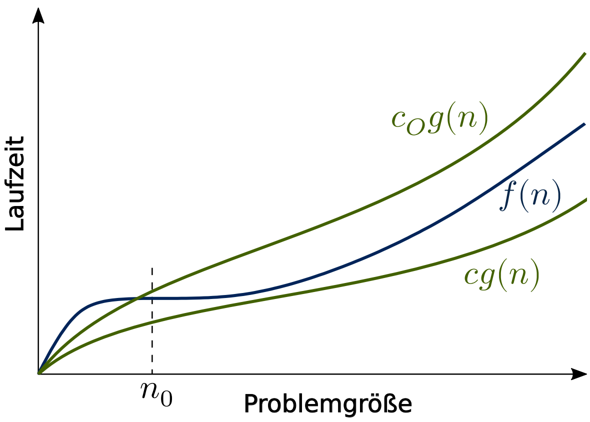
Video 6 beginnt hier.

Vgl. Groß-O

Definition: Seien $f(n)$ und $g(n)$ Funktionen, die nicht-negative ganze Zahlen auf nicht-negative reelle Zahlen abbilden. $f(n) \in \Omega(g(n))$ wenn $g(n) \in O(f(n))$, d.h., eine reelle Konstante $c > 0$ und eine ganzzahlige Konstante $n_0 \geq 1$ existieren, so dass

$$f(n) \geq cg(n) \quad \forall n \geq n_0.$$

Definition: $f(n) \in \Theta(g(n))$ wenn $f(n) \in O(g(n)) \cap \Omega(g(n))$.



Für $\Theta(g(n))$ werden die Beweise für $O(f(n))$ und $\Omega(f(n))$ *separat* geführt. Die Konstanten werden immer $c_O \neq c_\Omega$ sein müssen (wie hier illustriert). Die n_0 dürfen sich also ebenfalls unterscheiden, oder man wählt ein gemeinsames $n_0 = \max\{n_{0,O}, n_{0,\Omega}\}$.

Definition

Im vorherigen Video haben wir bereits Groß-O kennengelernt. In dieser Einheit besprechen wir zwei verwandte Konzepte.

Groß-Omega funktioniert genauso wie Groß-O, aber beschreibt eine untere statt einer oberen Schranke. $f(n) \in \Omega(g(n))$ wenn $g(n) \in O(f(n))$, das heißt eine reelle Konstante $c > 0$ und eine ganzzahlige Konstante $n_0 \geq 1$ existieren, so dass $f(n) \geq c \cdot g(n)$ für alle $n \geq n_0$ gilt. Hier wird die Formel von Groß-O umgedreht.

Groß-Theta bildet die Kombination von Groß-O und Groß-Omega, genau genommen die Schnittmenge. Das heißt, $f(n) \in \Theta(g(n))$ wenn $f(n) \in O(g(n))$ und auch $\in \Omega(g(n))$ ist.

Asymptotische Charakterisierung von Funktionen und Algorithmen [Slide 38]

- O, Ω und Θ charakterisieren *Funktionen*.
- Bei uns beschreibt eine solche Funktion typischerweise den *Ressourcenbedarf eines Algorithmus*, z.B. seine Laufzeit im schlechtesten Fall.
- Wir können die *Koeffizienten* unserer Funktionen meist nicht bestimmen, aber sie sind bei der asymptotischen Charakterisierung ohnehin *irrelevant*.

Wir verwendenden die asymptotische Notation zur Charakterisierung von *Funktionen*. Diese Funktionen beschreiben für uns i.d.R. den Ressourcenbedarf eines Algorithmus.

- Mittels O und Ω geben wir asymptotische obere bzw. untere Schranken für eine Funktion (und sind diese identisch, dann nutzen wir Θ).
- Der Ressourcenbedarf eines Algorithmus lässt sich mittels Funktionen charakterisieren. I.A. gibt es mehrere solcher Funktionen, z.B. jeweils eine für die Laufzeit im besten (sagen wir, f), schlechtesten (g) und Erwartungsfall.

  Allerdings können wir meist die Koeffizienten dieser Funktionen nicht bestimmen.
- Jede dieser Funktionen können wir wie oben mittels O, Ω und Θ beschreiben. Beispielsweise sind bei Insertion Sort $f(n) \in \Theta(n)$ und $g(n) \in \Theta(n^2)$, wie wir später sehen werden.

  Hierfür werden die nicht bestimmbaren Koeffizienten gar nicht benötigt.
- Wir können diese Funktionen auch zu einer gemeinsamen Funktion h kombinieren, die z.B. zufällig zwischen f und g auswählt. Dann ist bei Insertion Sort $h \in O(n^2)$ und $h \in \Omega(n)$.

Insbesondere beziehen sich *nicht* etwa O (nur) auf den schlechtesten Fall und Ω (nur) auf den besten Fall. Beide beziehen sich jeweils auf eine bestimmte Funktion, und diese Funktion beschreibt einen bestimmten Ressourcenbedarf.

Grenzen von Groß-O [Slide 39]

Was ist in der Praxis vorzuziehen:

- 2^n oder n^{100}?

  Ab $n = 997$ ist $2^n > n^{100}$, aber 997^{100} ist eine absurd große Zahl.

- $10^{100}n$ oder n^2?

- 10 oder $\log \log n$?

  Astronomen nehmen an, dass das Universum weniger als ein *Googol* (10^{100}) Atome enthält; $\log \log 10^{100} < 6$.

Mein Algorithmus ist doppelt so schnell wie deiner!

Diese wird durch Groß-O nicht erfasst; $O(2n) = O(n)$

Wichtig

Bei Problemen endlicher Größe darf man die *konstanten Faktoren* nicht aus den Augen verlieren, die bei der asymptotischen Analyse unter den Tisch fallen.

Erklärung

Die Groß-O-Notation bezieht sich auf den Grenzfall unendlich großer Probleme. Hier ist sie einfach anwendbar und sehr aussagekräftig. Dennoch sollte man real erwartbare Problemgrößen nicht aus dem Blick verlieren.

Welcher Algorithmus ist vorzuziehen, einer mit einer Laufzeit von 2^n oder einer mit einer Laufzeit von n^{100}?

$O(n^{100})$ ist eine Untermenge von $O(2^n)$, also empfiehlt uns Groß-O den n^{100}-Algorithmus. Bei einem derart großen Exponenten lohnt es sich jedoch, genauer hinzuschauen. Bis $n = 996$ ist $2^n < n^{100}$. Allerdings ist 996^{100} eine so große Zahl, dass Probleme dieser Größe in der Praxis nicht mehr berechnet werden können. Für alle praktisch relevanten Problemgrößen ist der $O(2^n)$-Algorithmus also schneller als der $O(n^{100})$-Algorithmus – bis auf den konstanten Faktor c, der sich hinter der Groß-O-Notation versteckt.

Diesen konstanten Faktor sollte man ebenfalls nicht aus den Augen verlieren: $O(10^{100}n) = O(n)$, aber 10^{100} ist ein so enorm großer Faktor, dass n^2 ihn in der Praxis kaum erreichen wird. Astronomen schätzen, dass das Universum weniger als 10^{100} Atome enthält. Allerdings werden derartig große Konstanten in realen Algorithmen niemals auftreten.

Umgekehrt wächst $\log \log n$ so enorm langsam, dass $\log \log 10^{100} < 6$ ist. In der Praxis kann $\log \log n$ also als konstant angesehen werden.

Vielleicht der relevanteste Aspekt, der bei der Groß-O-Analyse unter den Tisch fällt, ist die Frage, ob ein Algorithmus um einen konstanten Faktor schneller ist als ein anderer. Um diese Frage zu beantworten, wurde die Werkzeugkiste der asymptotischen Analyse um die Tilde-Notation erweitert.

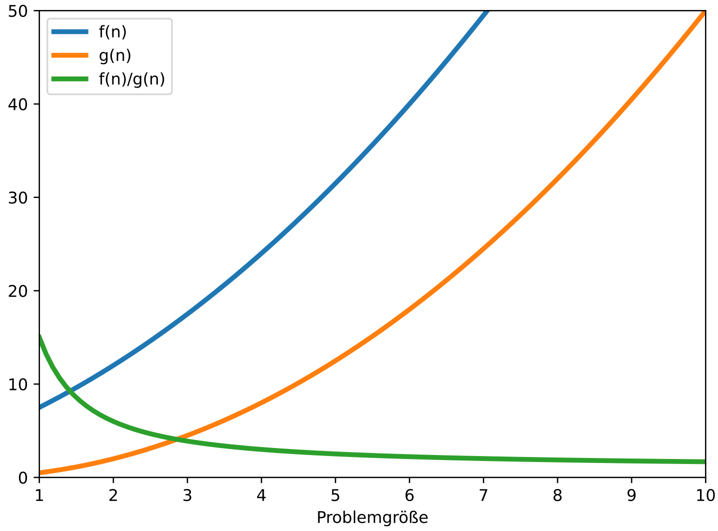
5 Tilde-Approximation

Tilde-Approximation [Sedgewick und Wayne 2014] [Slide 40]

Definition: $f(n) \sim g(n)$ genau dann wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

Beispiel:

$$\frac{1}{2}n^2 + 3n + 4 \;\sim\; \frac{1}{2}n^2$$



Erklärung

Bei der Tilde-Approximation bleibt, im Gegensatz zur Groß-O-Notation, der Koeffizient des am schnellsten wachsenden Terms erhalten. $f(n)$ ist Tilde $g(n)$ heißt, dass sich das Verhältnis von $f(n)$ zu $g(n)$ bei wachsendem n dem Wert 1 annähert.

In unserem Beispiel heißt das konkret: $1/2n^2+3n+4$ nähert sich für $n \to \infty$ an $1/2n^2$ an. Der quadratische Term dominiert für große n über die linearen und konstanten Terme.

Θ und Tilde [Slide 41]

2	$\Theta(1)$	~ 2
$2n+3$	$\Theta(n)$	$\sim 2n$
$2n^2+3n+4$	$\Theta(n^2)$	$\sim 2n^2$
$\log_2 n + 1$	$\Theta(\log n)$	$\sim \log_2 n$

Θ vs. Tilde [Slide 42]

Θ

Definition einfacher anwendbar

Konstanten vernachlässigbar

Tilde

Definition anschaulicher

Algorithmus-bedingte Konstanten bleiben erhalten

Bei ungenauer Analyse möglicherweise irreführend:

for $i \leftarrow 1$ **to** $2n$ **do**
  $s \leftarrow s + i$

for $i \leftarrow 1$ **to** n **do**
  $s \leftarrow s + 2i + n$

Für (ungenaue) Tilde-Approximation – aber nicht für Θ – sieht eine kürzere Schleife besser aus, auch wenn innerhalb jeder Iteration mehr Rechenzeit anfällt als bei der längeren Schleife.

In diesem Beispiel allerdings, in dem beide Schleifen dasselbe Ergebnis berechnen, läuft die zweite Schleife tatsächlich fast doppelt so schnell wie die erste, da Arithmetik effizienter ist als Sprünge.

Erklärung

Wenn wir Theta und Tilde vergleichen, erkennen wir noch einmal klar, dass bei der Tilde-Approximation der Koeffizient des am schnellsten wachsenden Terms erhalten bleibt. Bei der Theta-Analyse wird dieser nicht berücksichtigt.

Dies ist ein Vorteil der Tilde-Approximation mit großer praktischer Bedeutung. Er wird jedoch durch einen hohen Preis erkauft: Um diese Konstante zu ermitteln, müssen wir fast so genau hinschauen wie beim Zählen elementarer Operationen.

In diesem Beispiel sieht die zweite Schleife für die Tilde-Approximation doppelt so schnell aus wie die erste, obwohl sich die Gesamtanzahl aller primitiven Operationen nicht so sehr unterscheidet. Andererseits läuft auf heutigen Rechnern die zweite Schleife tatsächlich fast doppelt so schnell wie die erste, da Arithmetik effizienter ist als Sprünge.

6 Beweistechniken

Beispiel und Gegenbeispiel [Slide 43]

Beispiel

Proposition: Es gibt ganze Zahlen größer als eins, die gleich dem Produkt der Summe und des Produkts ihrer Ziffern sind.

Beweis (durch Beispiel):

$135 = 9 \cdot 15$

$144 = 9 \cdot 16$

(und das sind alle.)

Beispiel

Proposition: Prof. Überschlau behauptet, alle Zahlen $2^i - 1$ seien prim. Er hat Unrecht.

Beweis (durch Gegenbeispiel): $2^4 - 1 = 15 = 3 \cdot 5$

Kontraposition [Slide 44]

Wir beweisen $A \Rightarrow B$, indem wir die äquivalente Aussage $\neg B \Rightarrow \neg A$ beweisen.

Beispiel

Proposition: Seien a und b ganze Zahlen. Wenn ab gerade ist, dann ist mindestens eine der Zahlen a und b gerade.

Beweis: *Sind a und b ungerade, dann ist ab ungerade.* Seien $a = 2i + 1$ und $b = 2j + 1$. Dann gilt $ab = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1$; also ist ab ungerade.

Widerspruch [Slide 45]

Wir beweisen $A \Rightarrow B$, indem wir zeigen, dass $\neg B$ im Widerspruch zu A steht.

... woraus also $\neg A$ folgt. Widerspruchs- und Kontrapositionsbeweise sind tatsächlich nahe verwandt und folgen derselben Logik.

Beispiel

Proposition: Seien a und b ganze Zahlen. Ist ab ungerade, dann sind sowohl a als auch b ungerade.

Beweis: Angenommen, *ab sei ungerade.* Sei o.B.d.A. a gerade, also, $a = 2i$. Also ist $ab = 2ib$, d.h., *ab ist gerade*, was der o.g. Annahme widerspricht. Es folgt, dass sowohl a als auch b ungerade sind.

Vollständige Induktion [Slide 46]

Zu beweisen:

Induktionsanfang: Die Proposition ist für $n = n_0$ wahr.

Induktionsschritt: Ist die Proposition für ein beliebiges $n \geq n_0$wahr, dann ist sie auch für $n + 1$ wahr.

Es gibt viele Varianten dieses Prinzips.

Vollständige Induktion [Slide 47]

Zu beweisen:

Induktionsanfang: Die Proposition ist für $n = n_0$ wahr.

Induktionsschritt: Ist die Proposition für ein beliebiges $n \geq n_0$wahr, dann ist sie auch für $n + 1$ wahr.

Beispiel

Proposition: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Induktionsanfang: Bei $n = 0$ ergibt die Proposition $0 = 0$.

Induktionsschritt: Angenommen, die Proposition sei für ein beliebiges $n \geq 0$ wahr. Dann gilt $\sum_{i=1}^{n+1} i = \frac{n(n+1)}{2} + n + 1 = \frac{(n+1)(n+2)}{2}$.

- Im Induktionsschritt ist die erste Gleichung aufgrund der Induktionsannahme wahr; wir haben lediglich $n + 1$ auf beiden Seiten hinzuaddiert.
- Die zweite Gleichung können wir leicht verifizieren.
- Die Ausdrücke ganz links und ganz rechts entsprechen exakt der Proposition, außer dass n durch $n + 1$ ersetzt wurde. Wir haben gezeigt, dass sie gleich sind, genau wie in der Proposition. Damit haben wir die Proposition für den Nachfolger von n bewiesen.
- Da $n \geq n_0$ beliebig gewählt werden kann, haben wir damit (per Induktion) die Proposition für alle Werte von $n > 0$ bewiesen.

Vollständige Induktion [Slide 48]

Zu beweisen:

Induktionsanfang: Die Proposition ist für $n = n_0$ wahr.

Induktionsschritt: Ist die Proposition für ein beliebiges $n \geq n_0$wahr, dann ist sie auch für $n + 1$ wahr.

Beispiel

Proposition: $F(n) < 2^n$, wobei F die Fibonacci-Funktion ist, definiert durch $F(0) = 0$, $F(1) = 1$ und $F(n) = F(n-2) + F(n-1)$ für $n > 1$.

Induktionsanfang: $F(0) < 2^0$ und $F(1) < 2^1$.

Dies ist ein *doppelter* Induktionsanfang, für zwei *aufeinander folgende* Werte von n. Im Induktionsschritt gehen wir also ebenfalls von zwei aufeinander folgenden Werten aus, für die die Induktionsannahme gelte:

Induktionsschritt: Angenommen, die Proposition sei für $F(n-2)$ und $F(n-1)$ wahr. Dann gilt $F(n) \;=\; F(n-2) + F(n-1) \;<\; 2^{n-2} + 2^{n-1} \;<\; 2 \cdot 2^{n-1} = 2^n$.

Schleifeninvarianten [Slide 49]

Ähnliches Prinzip wie vollständige Induktion

Zu beweisen:

Schleifenvoraussetzung: P_0 gilt direkt vor der Schleife.

Schleifeninvarianz: Gilt P_{i-1} vor Iteration i, dann gilt P_i nach Iteration i.

Schleifenkonsequez: Die endgültige Proposition P_k gilt unmittelbar nach dem Schleifenausstieg.

Beispiel

```
Algorithm arrayMax(A,n):
  Require: An array A storing
           n ≥ 1 integers.
  Ensure: Return the maximum
           element in A.
  m ← A[0]
  for i ← 1 to n − 1 do
    if m < A[i] then
      m ← A[i]
  return m
```

Schleifeninvarianten [Slide 50]

Beispiel

Algorithm arrayMax(A,n):
  Require: An array A storing $n \geq 1$ integers.
  Ensure: Return the maximum element in A.
  $m \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    if $m < A[i]$ **then**
      $m \leftarrow A[i]$
  return m

Invariante P_i: $m = \max\{A[0, ..., i]\}$

Schleifenvoraussetzung:
$P_0 : m = \max\{A[0]\} = A[0]$.

Schleifeninvarianz: Gelte P_{i-1} vor Iteration i, also $P_{i-1} : m = \max\{A[0, \dots, i-1]\}$.

- Falls $m < A[i]$, wird m auf $A[i]$ gesetzt.
- Andernfalls bleibt m unverändert.

In beiden Fällen gilt hinterher $P_i : m = \max\{A[0, \dots, i]\}$.

Schleifenkonsequenz: Es gilt $P_{n-1} : m = \max\{A[0, \dots, n-1]\}$.

7 Zusammenfassung

Zusammenfassung [Slide 51]

- Ressourcenbedarf:
  - Asymptotische Analyse
    * Groß-O, Groß-Omega und Groß-Theta
    * Tilde-Approximation
  - Die Laufzeit-Komplexität eines Algorithmus dominiert die Effizienz seiner Implementation.
- Beweistechniken:
  - durch (Gegen-)*Beispiel*
  - durch *Kontraposition* und *Widerspruch*
  - durch *vollständige Induktion*
  - durch *Schleifeninvarianz*

Bibliographie [Slide 52]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java.* Wiley.
Sedgewick, Robert und Kevin Wayne (2014). *Algorithmen.* Pearson.