Programmiertechnik II

Ralf Herbrich

Analyse von Algorithmen

Überblick

1. Wissenschaftliche Methode (*scientific method*)
2. Messung von Laufzeit
3. Mathematische Modelle für Laufzeiten
4. Klassifikation von Komplexität: Θ-, $\mathcal{O}$-, und Ω-Notation

Überblick

1. Wissenschaftliche Methode (*scientific method*)
2. Messung von Laufzeit
3. **Mathematische Modelle für Laufzeiten**
4. Klassifikation von Komplexität: Θ-, $\mathcal{O}$-, und Ω-Notation

Mathematische Modelle der Laufzeit

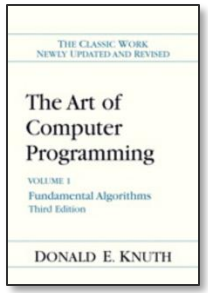
- **Ein vereinfachtes Laufzeitmodell**. Für die Laufzeit $T(n)$ gilt

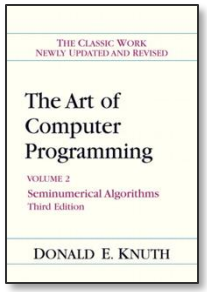
$$T(n) = \sum_{\{\text{ops}\}} C(\text{ops}) \cdot N(\text{ops})$$

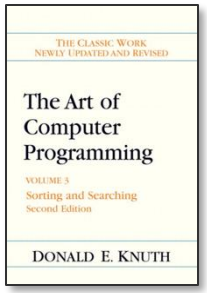
Zeit(kosten) der Einzeloperation ops

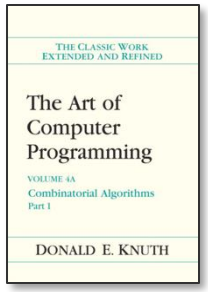
Anzahl Einzeloperationen des Typs ops

  - $C(\text{ops})$ hängt von Hardware und Compiler/Interpreter ab
  - $N(\text{ops})$ hängt von Algorithmus und Eingabedaten ab
- Im Prinzip gibt es akkurate Modelle für Laufzeiten!









Donald Knuth
(1938 –)

Laufzeit-(Kosten) $C(\text{ops})$ von Einzeloperationen

- **Beispiel für $C(\text{ops})$ (OS-X auf Macbook Pro 2.2 GHz)**

Operation	Beispiel	$C(\text{ops})$
Ganzzahladdition	`a + b`	2.1 ns
Ganzzahlmultiplikation	`a * b`	2.4 ns
Ganzzahldivision	`a / b`	5.4 ns
Gleitkommazahladdition	`a + b`	4.6 ns
Gleitkommazahlmultiplikation	`a * b`	4.2 ns
Gleitkommazahldivision	`a / b`	13.5 ns

- Viele **Basisoperationen** brauchen **konstante Laufzeit**

Operation	Beispiel	$C(\text{ops})$
Zuweisung	`a = b`	c_1
Ganzzahlvergleich	`a < b`	c_2
Arrayzugriff	`a[i]`	c_4
Speicheranforderung	`a = new int[n]`	$n \cdot c_3$

Beispiel: 1-Summen Problem

- Wie viele Operationen hat das Programm als Funktion der Eingabelänge n?

```
int count_1sum(const int* list, const int size) {
    int count = 0;
    for (auto i = 0; i < size; i++) {
        if (list[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Operation	$N(\text{ops})$
Zuweisung	2
Kleiner-Vergleich	$n + 1$
Gleichheits-Vergleich	n
Arrayzugriff	n
Inkrement	n bis $2n$

Beispiel: 2-Summen Problem

- Wie viele Operationen hat das Programm als Funktion der Eingabelänge n?

```
// counts the number of tuples that sum to exactly 0
int count_2sums(const int* list, const int size) {
    int count = 0;
    for (auto i = 0; i < size; i++) {
        for (auto j = i + 1; j < size; j++) {
            if (list[i] + list[j] == 0) {
                count++;
            }
        }
    }
    return count;
}
```

Kleiner-Vergleich $(n+1) + n + \cdots + 1 = \frac{(n+2)\cdot(n+1)}{2}$

Gleichheits-Vergleich $(n-1) + (n-2) + \cdots + 0 = \frac{n\cdot(n-1)}{2}$

Operation	$N(\text{ops})$
Zuweisung	$n+2$
Kleiner-Vergleich	$\frac{(n+1)\cdot(n+2)}{2}$
Gleichheits-Vergleich	$\frac{(n-1)\cdot n}{2}$
Arrayzugriff	$(n-1)\cdot n$
Inkrement	$\frac{(n-1)\cdot n}{2}$ bis $(n-1)\cdot n$

← Dominante Operation!

Tilde-Notation

- Wir wollen die Laufzeit als Funktion der Eingabelänge n schätzen
- **Idee**: Wir ignorieren alle Ausdrücke niedriger Ordnung/Potenz von n
  - **Fall 1**: n ist groß: ignorierte Ausdrücke sind vernachlässigbar
  - **Fall 1**: n ist klein: die gesamte Laufzeit ist klein
- **Tilde-Notation**. Für zwei Funktion f und g bedeutet $f \sim g$, dass

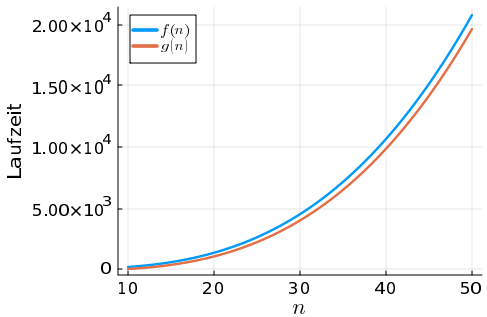
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

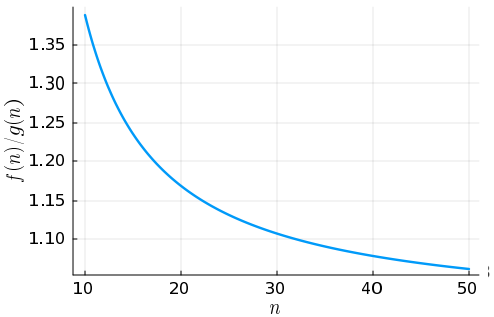
- **Beispiele**:

$$\frac{1}{6}n^3 + 20n + 16 \quad \sim \frac{1}{6}n^3$$

$$\frac{1}{6}n^3 + 100n^{\frac{4}{3}} + 56 \quad \sim \frac{1}{6}n^3$$

$$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n \quad \sim \frac{1}{6}n^3$$





Beispiel: 2-Summen Problem

- Wie viele Operationen hat das Programm als Funktion der Eingabelänge n?

```
int count_2sum(const int* list, const int size) {
    int count = 0;
    for (auto i = 0; i < size; i++) {
        for (auto j = i+1; i < size; i++) {
            if (list[i] + list[j] == 0) {
                count++;
            }
        }
    }
    return count;
}
```

Operation	$N(\text{ops})$	Tilde-Notation
Zuweisung	$n + 2$	$\sim n$
Kleiner-Vergleich	$\frac{(n+1)\cdot(n+2)}{2}$	$\sim \frac{1}{2}n^2$
Gleichheits-Vergleich	$\frac{(n-1)\cdot n}{2}$	$\sim \frac{1}{2}n^2$
Arrayzugriff	$(n-1)\cdot n$	$\sim n^2$
Inkrement	$\frac{(n-1)\cdot n}{2}$ bis $(n-1)\cdot n$	$\sim \frac{1}{2}n^2$ bis $\sim n^2$

3-Summen Problem

- **Frage**: Wie viele Arrayzugriffe braucht das 3-Summen Problem approximativ?

```
// counts the number of triples that sum to exactly 0
int count_3sums(const int* list, const int size) {
    int count = 0;
    for (auto i = 0; i < size; i++) {
        for (auto j = i + 1; j < size; j++) {
            for (auto k = j + 1; k < size; k++) {
                if (list[i] + list[j] + list[k] == 0) {
                    count++;
                }
            }
        }
    }
    return count;
}
```

Innere Schleife

- **Antwort**: In der inneren Schleife haben wir

Anzahl Möglichkeiten, 3 eindeutige Indizes ohne Beachtung der Reihenfolge aus n auszuwählen

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j}^{n} 3 = 3 \cdot \binom{n}{3} = 3 \cdot \frac{n \cdot (n-1)(n-2)}{6} \sim \frac{1}{2}n^3$$

Mathematische Modelle der Laufzeit

- **Im Prinzip** gibt es akkurate Modelle für Laufzeiten!

- **In der Praxis**
  - können die Gleichungen kompliziert sein
  - Ist aufwendige Berechnung von Reihen notwendig
  - Sind exakte Modelle sehr aufwendig

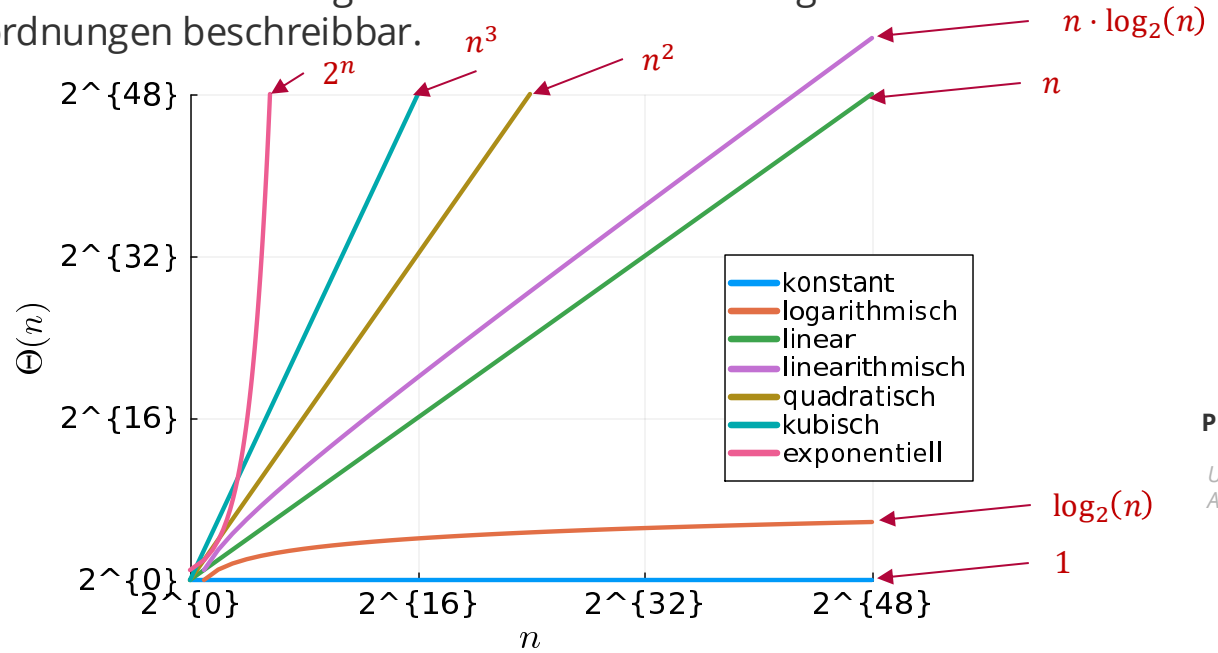
- Daher werden in der Praxis, **drei Vereinfachungen** benutzt:
  1. Ein lineares Modell von Laufzeit von Einzeloperationen und deren Häufigkeit
  2. Fokus auf die dominanten Einzeloperationen
  3. Approximation durch Tilde-Notation

Überblick

1. Wissenschaftliche Methode (*scientific method*)
2. Messung von Laufzeit
3. Mathematische Modelle für Laufzeiten
4. **Klassifikation von Komplexität: Θ-, $\mathcal{O}$-, und Ω-Notation**

Wachstumsordnung

- **Definition**. Wir sagen, dass die Funktion $f: \mathbb{N} \to \mathbb{N}$ die Wachstumsordnung $\Theta(g)$ hat, wenn es eine Konstante $c \in \mathbb{R}$ gibt, so dass $f \sim c \cdot g$.
- In der Praxis sind die meisten Algorithmen mit Hilfe der folgenden Wachstumsordnungen beschreibbar.



Wachstumsordnung: Beispiel

$\Theta(n)$	Name	Beispielcode	Algorithmus	$T(2n)/T(n)$
1	Konstant	`a = b + c`	Addition zweier Zahlen	1
$\log_2(n)$	Logarithmisch	`while (n > 1) { … n /= 2; … }`	Binäre Suche	~ 1
n	Linear	`for(int i=0; i < n; i++) { … }`	Schleife	2
$n \cdot \log_2(n)$	Linearitmisch	[Quick-Sort Vorlesung]	*divide & conquer*	~ 2
n^2	Quadratisch	`for(int i=0; i < n; i++) { for(int j=0; j < n; j++) { … } }`	Doppelschleife	4
n^3	Kubisch	`for(int i=0; i < n; i++) { for(int j=0; j < n; j++) { for(int k=0; k < n; k++) { … } }`	Dreifachschleife	8
2^n	Exponentiell	[Algorithmenparadigmen Vorlesung]	Erschöpfenden Suche	$T(n)$

Beschleunigung vom 3-Summen Problem

- **Idee**: Wenn wir zwei der drei Zahlen kennen, d.h. `a[i]` und `a[j]`, dann muss dritte Element `-(a[i] + a[j])` sein, damit die Summe Null ist.
- **Problem**: Wie schnell können wir in einer Liste von eindeutigen Zahlen eine spezielle Zahl finden?
  - **Naiv**: Eine Schleife, die jede Zahl mit der Schlüsselzahl vergleicht. ← $\sim n$
  - **Clever**: Wenn die Liste sortiert ist, binäre Suche

John Mauchly
(1907 – 1980)

```c
// binary search algorithm to find the key in the list
int binary_search(const int* list, const int size, const int key) {
    int lo = 0;
    int hi = size - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (key < list[mid]) {
            hi = mid - 1;
        } else if (key > list[mid]) {
            lo = mid + 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

← 3-wertiger Vergleich

Binäre Suche: Mathematische Analyse

- **Satz**. Binäre Suche findet den Schlüssel oder terminiert nach höchstens $1 + \log_2(n)$ 3-wertigen Vergleichen in einem sortierten Feld der Länge n.

- **Beweis**. Wir definieren $T(n)$ als die maximale Anzahl der 3-wertigen Vergleiche in einem Array der Länge kleiner oder gleich n.

  Aufgrund des Algorithmus wissen wir dass

  $$T(n) \leq T(n/2) + 1 \quad \text{und} \quad T(1) = 1$$

  Daher gilt

$$\begin{aligned} T(n) &\leq T(n/2) + 1 \\ &\leq T(n/4) + 1 + 1 \\ &\vdots \\ &\leq T(n/n) + 1 + 1 + \cdots + 1 \\ &\leq 1 + \log_2(n) \end{aligned}$$

Die Rekurrenz kann maximal $\log_2(n)$ angewandt werden

```java
// binary search algorithm to find the key in the list
int binary_search(const int* list, const int size, const int key) {
    int lo = 0;
    int hi = size - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (key < list[mid]) {
            hi = mid - 1;
        } else if (key > list[mid]) {
            lo = mid + 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

Ein $\Theta(n^2 \cdot \log_2(n))$ Algorithmus für 3-Summen

- **Algorithmus**:
  1. Sortierte die n eindeutigen Zahlen in der Liste.
  2. Für alle eindeutigen Paare `a[i]` und `a[j]`, suche `-(a[i] + a[j])` mit binärer Suche in den verbleibenden Elementen.
- **Analyse**:
  1. Das Sortieren kann mit Hilfe eines Algorithmus in $\Theta(n^2)$
  2. Für die $\Theta(n^2)$ eindeutigen Paare benötigt binäre Suche $\Theta(\log_2(n))$ Vergleiche

- **Ergebnis**: Der Algorithmus wird von Schritt 2. dominiert, welcher $\Theta(n^2 \cdot \log_2(n))$ Schritte benötigt!

- **Bemerkung**: Wir können sogar $\Theta(n^2)$ erzielen (aber viel komplizierter Algorithmus)

Empirische Analyse

3 Summen Problem ($\Theta(n^3)$)

n	Laufzeit (in s)
1000	0.02104
2000	0.107
4000	0.6583
8000	4.904
16000	37.93

3 Summen Problem ($\Theta(n^2 \cdot \log_2(n))$)

n	Laufzeit (in s)
1000	0.02714
2000	0.0795
4000	0.2644
8000	1.105
16000	4.557
32000	18.91

Beste, Schlechteste und Durchschnittliche Laufzeit

- In der Praxis hängt die Laufzeit stark von der **konkreten** Eingabe ab!
- **Bester Fall (*Best Case*)**: Untere Schranke an die Laufzeit
  - Wird durch die einfachste Eingabe bestimmt
  - Gibt ein Ziel für alle Eingaben
- **Schlechtester Fall (*Worst Case*)**: Obere Schranke an die Laufzeit
  - Wird durch die schwierigste Eingabe bestimmt
  - Gibt eine Garantie für alle Eingaben
- **Durchschnittlicher Fall (*Average Case*)**: Erwartete Laufzeit bei "zufälliger" Eingabe
  - Benötigt ein Modell für "zufällige" Eingabe
  - Möglichkeit, die Laufzeit vorherzusagen

Arrayzugriffe bei naive 3-Summen

Bester Fall:	$\sim 1/2 \cdot n^3$
Schlechtester Fall:	$\sim 1/2 \cdot n^3$
Durchschnittlicher Fall:	$\sim 1/2 \cdot n^3$

3-wertige Vergleich bei binärer Suche

Bester Fall:	~ 1
Schlechtester Fall:	$\sim \log_2(n)$
Durchschnittlicher Fall:	$\sim \log_2(n)$

Komplexitätsnotationen in Algorithmentheorie

Notation	Bedeutung	Beispiel	Abkürzung für …	Benutzung
Tilde	dominierender Ausdruck	$\sim 10n^2$	$10n^2$ $10n^2 + 22n \cdot \log_2(n)$ $10n^2 + 2n + 37$	Approximatives Laufzeitmodell
Big Theta	Wachstumsordnung	$\Theta(n^2)$	$1/2 \cdot n^2$ $10n^2$ $5n^2 + 22n \cdot \log_2(n) + 3n$	Klassifikation von Algorithmen
Big-O	$\Theta(n^2)$ oder kleiner	$\mathcal{O}(n^2)$	$10n^2$ $100n$ $22n \cdot \log_2(n) + 3n$	Entwicklung von oberen Schranken
Big-Omega	$\Theta(n^2)$ oder größer	$\Omega(n^2)$	$1/2 \cdot n^2$ n^5 $n^3 + 22n \cdot \log_2(n) + 3n$	Entwicklung von unteren Schranken

Algorithmentheorie

- **Ziele**:
  1. Bestimmung der “Schwierigkeit” eines Problems (Komplexität)
  2. Entwicklung von “optimalen” Algorithmen

Optimaler Algorithmus:
Untere Schranke = Obere Schranke

- **Ansatz**:
  1. Analyse der Komplexität bis auf einen multiplikativen Faktor
  2. Keine Annahmen über die Eingabe machen und Fokus auf schwierigsten Fall

- **Beispiel**: 1-Summen Problem (d.h., Wie viele “0” gibt es in einem Array?)
  - **Obere Schranke**: Überprüfe jedes einzelne Element: $\mathcal{O}(n)$
  - **Untere Schranke**: Jedes nicht-untersuchte Element könnte 0 sein: $\Omega(n)$
  - **Optimaler Algorithmus**: Brute-Force Suche durch das Array, da untere und obere Schranke übereinstimmen

```
int count_1sum(const int* list, const int size) {
    int count = 0;
    for (auto i = 0; i < size; i++) {
        if (list[i] == 0) {
            count++;
        }
    }
    return count;
}
```

- Mehr Details und Tiefe zu Algorithmentheorie in **TI 1**!

Zusammenfassung

- **Empirische Analyse**
  - Programm ausführen, um experimentelle Ergebnisse zu sammeln
  - Annahme eines *power laws* und Formulierung einer Laufzeithypothese
  - Schätzung der Parameter des *power laws* um Laufzeitvorhersagen zu machen
- **Mathematische Analyse**
  - Algorithmen analysieren, um Häufigkeit von Einzeloperationen zu zählen
  - Tilde Notation benutzen, um (approximative) Analyse zu vereinfachen
  - Model erlaubt es, Laufzeitverhalten zu erklären
- **Wissenschaftliche Methode**
  - Mathematischen Modelle sind unabhängig von Hard- und Software; gelten auch für zukünftige Computer!
  - Empirische Analyse ist notwendig, um mathematische Analyse zu validieren durch Laufzeitvorhersagen

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!