



Algorithmen und Datenstrukturen

Kapitel 02: Komplexitätsanalyse

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Kosten von Algorithmen

Wieviele Ressourcen (Laufzeit/Speicher) benötigt ein Algorithmus?

Ansätze

1. **Benchmarking:** Implementiere den Algorithmus in einer Programmiersprache und teste ihn mit verschiedenen Eingaben.
2. **Zählen der Elementaroperationen** des Algorithmus, Ableitung einer Kostenformel.

Nachteile von Benchmarking?

- ▶ Benchmarking-Ergebnisse sind abhängig von **Kontextfaktoren (Hardware, Sprache, Compiler, Implementierungsdetails, Last)**.
- ▶ In der Regel sind **nicht alle möglichen Eingaben** testbar.

```
import java.util.Arrays;

class Enigma {

    public static int minPos(int[] numbers,
                           int k) {

        int min_pos = k;

        for(int i=k; i<numbers.length; ++i) {
            if(numbers[i]<numbers[min_pos]) {
                min_pos = i;
            }
        }

        return min_pos;
    }

    public static void swap(int[] numbers,
                          int pos1,
                          int pos2) {

        int help = numbers[pos1];

        numbers[pos1] = numbers[pos2];
        numbers[pos2] = help;
    }

    public static void enigma(int[] numbers) {

        for(int k=0; k<numbers.length; ++k) {

            int min_pos = minPos(numbers, k);
            swap(numbers, min_pos, k);
        }
    }
}
```

Kosten von Algorithmen

In ADS verfolgen wir Ansatz 2:

- Wir führen den Algorithmus gedanklich auf einer Maschine mit **vordefinierten Kosten** für verschiedene Operationen aus.
- Wir **zählen** bestimmte Einzelschritte (*Feldzugriffe, Additionen, Vergleiche, ...*).
- **Schlüsselfrage:** Wie verhält sich der Algorithmus für **große Eingaben**?

Vorteile dieser Kostenschätzung

- **Generelle** Aussage, unabhängigkeit von Plattform+Implementierung.
- Betrachtung **aller möglicher** Eingaben.
- **Aufwandsfrei** (*keine Implementierung, kein Testen*).

```
import java.util.Arrays;

class Enigma {

    public static int minPos(int[] numbers,
                           int k) {

        int min_pos = k;

        for(int i=k; i<numbers.length; ++i) {
            if(numbers[i]<numbers[min_pos]) {
                min_pos = i;
            }
        }

        return min_pos;
    }

    public static void swap(int[] numbers,
                          int pos1,
                          int pos2) {

        int help = numbers[pos1];

        numbers[pos1] = numbers[pos2];
        numbers[pos2] = help;
    }

    public static void enigma(int[] numbers) {

        for(int k=0; k<numbers.length; ++k) {

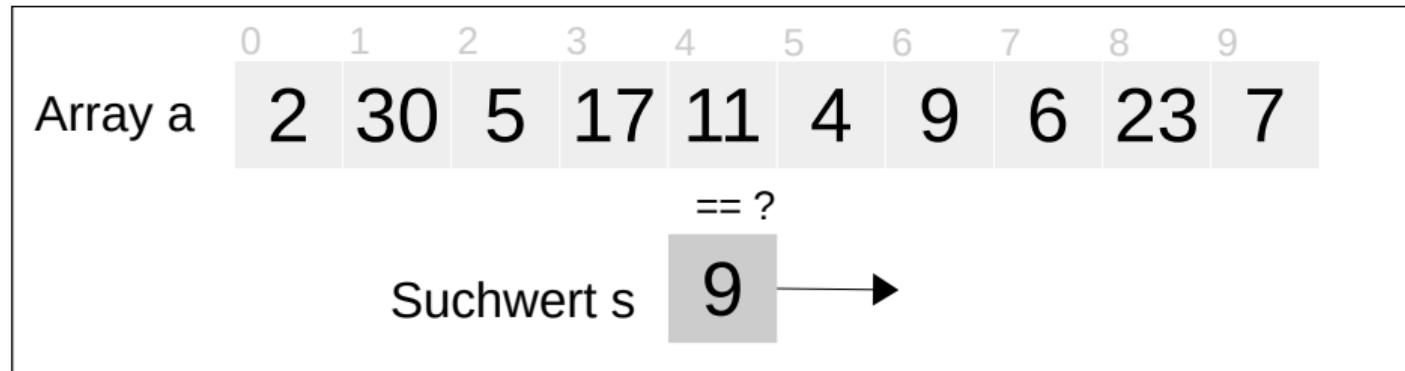
            int min_pos = minPos(numbers, k);
            swap(numbers, min_pos, k);
        }
    }
}
```



Outline

1. Beispiel: Lineare Suche
2. Die O-Notation
3. Code-Abschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche

Beispiel: Lineare Suche



Problemstellung

- Gegeben: Ein **Array** $a[0], a[1], \dots, a[n - 1]$, ein **Suchwert** s .
- Gebe die **Position** zurück, an der der Suchwert im Array vorkommt.
Ist der Wert **nicht** vorhanden, gebe n zurück.

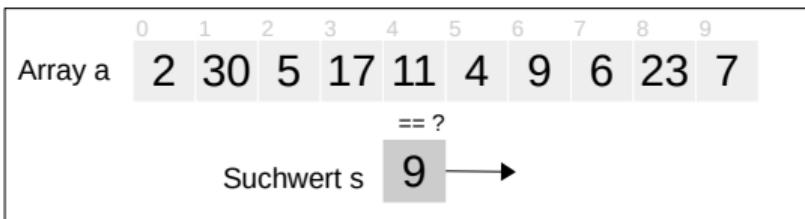
Ansatz

- Durchlaufe das Array von links nach rechts mit Variable pos.
- Breche ab, falls $a[\text{pos}]$ gleich dem Suchwert ist.

Beispiel: Lineare Suche

Pseudocode

```
1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
```



Kostenanalyse (*Beispiel rechts oben*)

Wir zählen **Vergleiche**, **Additionen**, **Feldzugriffe**, **Zuweisungen**:

- ▶ Initiale Zuweisung (Zeile 1): Kosten 1.
- ▶ 6 erfolglose Schleifendurchläufe (Zeile 2+3).
(2 Vergleiche & 1 Feldzugriff (Zeile 2), 1 Addition & 1 Zuweisung (Zeile 3))
- ▶ 7. Schleifendurchlauf: Suchwert gefunden, Kosten 3.
(2 Vergleiche & 1 Feldzugriff (Zeile 2))
- ▶ Verlassen der Schleife, Algorithmus ist terminiert.
- ▶ **Gesamtkosten: $1 + 6 \cdot 5 + 3 = 34$ Schritte.**

Effizienz von Algorithmen: Formalisierung

Für eine **generellere Kostenaussage abstrahieren** wir über die **Eingabedaten**:

- (a) **Umfang**: Wie lang ist das zu durchsuchende Array?
 - (b) **Schwierigkeit**: Wo befindet sich der Suchwert im Array?
-

(a) Umfang: Die Problemgröße

Gegeben ein zu lösendes Problem, bezeichnen wir den Umfang der Eingabedaten als **Problemgröße $n \in \mathbb{N}$** .

Die Problemgröße bezeichnet (je nach zu lösendem Problem) **verschiedene Dinge**:

- ▶ Die Länge eines Arrays
- ▶ Die Anzahl der Knoten in einem Graph
- ▶ Die Länge eines kryptografischen Schlüssels in Bit
- ▶ Die Anzahl der zu planenden Züge eines Schachcomputers.
- ▶ ...

(b) Die Schwierigkeit

Gegeben die Problemgröße n , betrachten wir ...

1. den **besten Fall** (engl. 'best case')

- ▶ Betrachte die "einfachste" Eingabe (der Größe n), welche die **minimal** mögliche Anzahl an Schritten verursacht.
- ▶ Dies ist meist **nicht besonders interessant**.

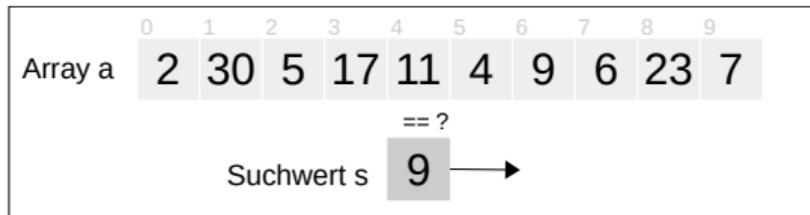
2. den **mittleren Fall** (engl. 'average case')

- ▶ Betrachte **alle** möglichen Eingaben (der Größe n) und **mittle** die Anzahl der benötigten Schritte.
- ▶ Dies ist meist **relevant**, aber **schwierig** zu berechnen.

3. den **schlechtesten Fall** (engl. 'worst case')

- ▶ Betrachte die „schwierigste“ Eingabe (der Größe n) mit der **maximal** möglichen Anzahl an Schritten.
- ▶ Dies ist meist **relevant** und **leicht** zu berechnen.

Beispiel: Lineare Suche



Pseudocode

```

1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
    
```

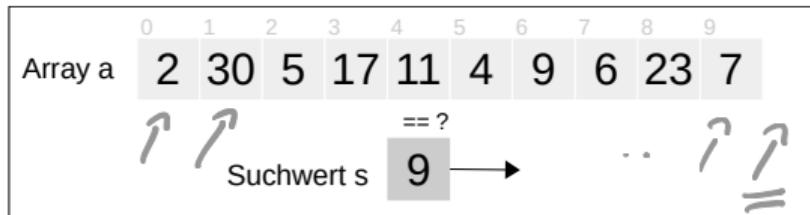
Best Case

- ▶ Suchwert befindet sich **an der 1. Position** im Array.
- ▶ Kosten: 4 (**1 Zuweisung (Zeile 1), 2 Vergleiche & 1 Feldzugriff (Zeile 2)**)

Worst Case

- ▶ Suchwert befindet sich **nicht** im Array.
- ▶ n erfolglose Schleifendurchläufe, jeweils Kosten 5.
- ▶ Zusatzkosten: 2 (**1 Zuweisung (Zeile 1), 1 Schleifenabbruch (Zeile 2)**)
- ▶ **Gesamtkosten:** $2 + 5 \cdot n$.

Beispiel: Lineare Suche



Pseudocode

```
1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
```

Average Case

- Annahme: $n+1$ gleich wahrscheinliche Fälle (Der Suchwert befindet sich an Position 0, 1, 2, ..., $n-1$, oder er ist "nicht enthalten").

Kosten:

$$\left[\underbrace{4 + 0.5}_{\substack{s \text{ an} \\ \text{Position } 0}} + \underbrace{4 + 1.5}_{\substack{s \text{ an} \\ \text{Position } 1}} + \dots + \underbrace{4 + (n-1) \cdot 5}_{\substack{s \text{ an} \\ \text{Position } n-1}} + \underbrace{2 + n \cdot 5}_{\substack{\text{ //} \\ s \text{ nicht} \\ \text{vorhanden}}} \right] \cdot \frac{1}{n+1}$$

$= (4 + n \cdot 5) - 2$

Durchschnitt!

Beispiel: Lineare Suche (cont'd)

*

$$\text{Kosten: } \frac{1}{n+1} \cdot \left(\left(\sum_{i=0}^n 4 + i \cdot 5 \right) - 2 \right)$$

Gauß'sche
Summenformel

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

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

$$= " \left((n+1) \cdot 4 + 5 \cdot \frac{n \cdot (n+1)}{2} - 2 \right)$$

$$= " \left(\overbrace{4n+4}^{4n+4} + \frac{5}{2} \cdot n^2 + \frac{5}{2} \cdot n - 2 \right)$$

$$= \frac{1}{n+1} \left(\frac{5}{2} n^2 + \cancel{\frac{13}{2} n + 2} \right) \approx \frac{\frac{5}{2} n^2}{n+1} \approx \frac{5}{2} n$$

!

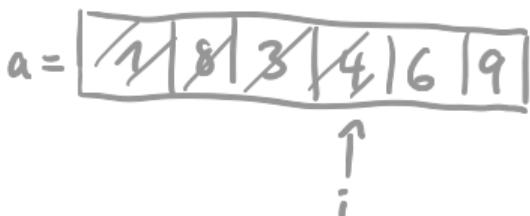
Aufwandsschätzung: Do-it-Yourself



Berechnen Sie den Worst-Case-Aufwand des folgenden Algorithmus.

Zählen Sie nur die Feldzugriffe.

Was tut $\text{psum}(a, i)$?



$$\text{psum}(a, 3) = 1 + 8 + 3 + 4 = 16$$

$\Rightarrow i+1$ Feldzugriffe!

```
1 # Gegeben: Ein n-elementiges
2 # Array a
3 b := ein n-elementiges Array
4 for i = 0, ..., n-1:
5   b[i] = psum(a, i)
6 return b
```

```
1 function psum(a, i):
2   result = 0
3   pos = 0  $\rightsquigarrow$  läuft durch a
4   while pos <= i:  $\rightsquigarrow$  bis zur
5     result += a[pos]
6     pos += 1
7   return result
```

(best case = worst case)

Aufwandsschätzung: Do-it-Yourself

Gesamtaufwand: je Durchlauf i (Zeile 4)

- 1 Zugriff $b[i]$ (Zeile 5)
- $i+1$ Zugriffe in $\text{psum}(a, i)$

$$\begin{aligned}
 &\Rightarrow \sum_{i=0}^{n-1} \left(1 + \underbrace{(i+1)}_{\text{blue}} \right) \\
 &= 2n + \sum_{i=0}^{n-1} i \quad // \text{Gauß'sche Summenformel} \\
 &= 2n + \frac{\cancel{(n-1) \cdot n}}{2} \\
 &= \frac{1}{2}n^2 - \frac{3}{2}n \quad \text{teuer...?}
 \end{aligned}$$

Aufwandsschätzung: Do-it-Yourself



Outline

1. Beispiel: Lineare Suche
2. Die O-Notation
3. Code-Abschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche



Kostenfunktionen

Definition (Kostenfunktion)

Gegeben sei ein Algorithmus \mathcal{A} . Die **Kostenfunktion** (oder **Laufzeit**) $a : \mathbb{N} \rightarrow \mathbb{R}^+$ ordnet jeder Problemgröße n den Ressourcenbedarf (z.B. die Anzahl der Operationen) $a(n)$ zu, die \mathcal{A} zur Verarbeitung einer Eingabe der Größe n benötigt.

Anmerkungen

- Wir können Kostenfunktionen für den Worst/Best/Average Case definieren. Für die lineare Suche gilt z.B. (siehe oben):

$$a^{\text{best}}(n) = 4 \quad a^{\text{worst}}(n) = 2 + 5n \quad a^{\text{avg}}(n) = \frac{5/2 \cdot n^2 + 13/2 \cdot n + 2}{n + 1}$$

- Die Kostenfunktion ist eine mathematische **Folge**:
Wir können für den Funktionswert a_n oder $a(n)$ schreiben.

Vereinfachung von Kostenfunktionen

Statt der exakten Anzahl der Einzelschritte reicht uns eine **grobe Abschätzung**. Dies führt zur **O-Notation, dem zentralen Konzept** der Aufwandsschätzung.

Schritt 1: Stärkstes Wachstum

- Wir konzentrieren uns auf den **am stärksten wachsenden Summanden** der Kostenfunktion:

$$\cancel{4n^2 + 2n + 5} \longrightarrow 4n^2$$

- Warum? Weil für große n der **relative Fehler vernachlässigbar** ist.
- Im obigen Beispiel für $n=10000$: 0.005%.

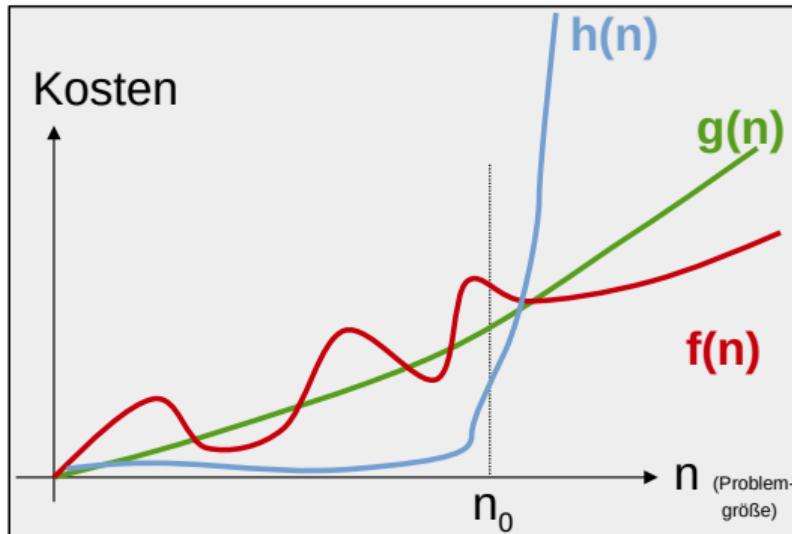
Vereinfachung von Kostenfunktionen

Schritt 2: Faktoren entfernen

$$\cancel{4} n^2 \longrightarrow n^2$$

- ▶ Konstante Faktoren beeinträchtigen die **wichtigsten Aussagen** nicht, wie z.B. “Bei einer Verdopplung der Eingabegröße braucht der Algorithmus doppelt so lange”.
- ▶ Eine Konstante 4 könnte auch durch eine vier mal **schnellere Maschine** erreicht werden.
- ▶ Diese “Details” interessieren uns hier nicht (*sondern die generelle Güte eines Algorithmus*).

O-Notation: Illustration



- ▶ Uns interessieren nur große n , ab einem beliebig großen n_0 .
- ▶ f wächst “nicht viel schneller” als g , oder kurz: $f \in O(g)$.
- ▶ Es gilt auch: $g \in O(f)$ (g wächst nicht viel schneller als f).
- ▶ Es gilt auch: $g \in O(h)$ (g wächst nicht viel schneller als h).
- ▶ Es gilt **nicht**: $h \in O(g)$ (h wächst viel schneller als g).

Definition: O-Notation

Definition (O-Notation)

Es seien f und g zwei Kostenfunktionen. Wenn es ein $c \in \mathbb{R}$ und ein $n_0 \in \mathbb{N}$ gibt, so dass

$$f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0,$$

dann schreiben wir $f \in O(g)$ (oder $f(n) \in O(g(n))$).

Anmerkungen

- ▶ **Umgangssprachlich** bedeutet $f \in O(g)$:
“ f wächst nicht deutlich schneller als g ”.
- ▶ $O(g)$ ist demnach die **Menge aller Kostenfunktionen**, die nicht deutlich schneller wachsen als g .
- ▶ Wir sprechen: “ f ist von der Ordnung g ” oder auch “ f ist **O von g** ”.

O-Notation: Beispiele

Beispiel I (multiplikative Konstante)

Gilt $\underline{3n^2} \in O(n^2)$?

$$3n^2 \leq c \cdot n^2$$

Wähle $c=3$

✓ (gilt)

Beispiel II (additive Konstante)

Gilt $\underline{n^2 + 3} \in O(n^2)$?

$$n^2 + 3 \leq c \cdot n^2$$

Wähle $c=2$

$n_0 = 100$

(✓) (wäre noch zu beweisen)

O-Notation: Beispiele

Beispiel I (multiplikative Konstante)

Gilt $n \in O(n^2)$?

$\stackrel{=}{=}$

$$n \leq c \cdot n^2 \quad \checkmark$$

$(c=1)$

Beispiel II (additive Konstante)

Gilt $n^2 \in O(n)$?

$\stackrel{=}{=}$

$$n^2 \leq c \cdot n \quad (\text{für alle } "gro\ddot{e}" n \text{ als } n_0)$$

$\not\subseteq$ (gilt nicht)

(Wir können kein $c \in \mathbb{R}$ finden so dass die Ungleichung für beliebig große n erfüllt ist)



Definition: O-Notation

- ▶ Mit der O-Notation fassen wir ähnliche Algorithmen/Aufwandsfunktionen zu **Klassen** zusammen.
- ▶ Algorithmen, deren Aufwand ähnlich schnell wächst, gehören zur gleichen Klasse (sie besitzen gleiche **Komplexität**).

Beispiel-Klassen

“lineare” Algorithmen → $n, 1000n + 3, \dots$

“quadratische” Algorithmen → $n^2, 7n^2 + 5n - 10, \dots$

“logarithmische” Algorithmen → $\log_2(n), \log_3(n), \log_8(n) + 4, \dots$

“exponentielle” Algorithmen → $2^n, 2^n + n^{10000} + 100000, \dots$

...

Definition: O-Notation

Weitere Anmerkungen

- Gängig ist auch die Schreibweise $f = O(g)$ (statt $f \in O(g)$). Dies ist aber missverständlich, denn die O-Beziehung ist **nicht symmetrisch**: Aus $n = O(n^2)$ folgt nicht dass $n^2 = O(n)$.

Laufzeit vs Komplexität

Wir unterscheiden zwischen der **Laufzeit** und der **Komplexität** eines Algorithmus:

- Die **Laufzeit** f_n entspricht der **exakten Anzahl** an Rechenschritten (*umständlich zu berechnen, siehe "lineare Suche"*).
- Die **Komplexität** $O(f_n)$ entspricht der **groben Abschätzung** des Aufwandes (*leicht zu berechnen, "genau genug"*).

Achtung: Komplexität möglichst präzise angeben.

Es gilt $2n \in O(2^n)$, aber auch $2n \in O(n)$ (*besser!*).

Komplexitätsklassen als Mengen



O-Notation: Beweis (Variante 1)



$$\cancel{4n^2 + 2n + 5} \rightarrow n^2$$

Wir zeigen per vollständiger Induktion: $\underbrace{4n^2 + 2n + 5}_{f} \in O(\underbrace{n^2}_{g})$.

Zu zeigen: $f \in O(g)$

„es gilt“ $\exists c, n_0 : 4n^2 + 2n + 5 \leq c \cdot n^2$ für alle $n \geq n_0$

Wir wählen: $c=5, n_0=1000$

A ($n=1000$) $4 \cdot 1000^2 + 2 \cdot 1000 + 5 \leq \underline{5} \cdot \underline{1000}^2$ ✓

O-Notation: Beweis (Variante 1)

IV Für ein beliebiges n gelte: $\underline{4n^2 + 2n + 5} \leq \underline{5n^2}$

I. Schritt

$$n \rightsquigarrow n+1$$

Zu zeigen: $\underline{4(n+1)^2 + 2(n+1) + 5} \leq \underline{5(n+1)^2}$

$$\begin{aligned}
 \underline{4(n+1)^2 + 2(n+1) + 5} &= \underline{4n^2} + 8n + 4 + \underline{2n} + 2 + \underline{5} \\
 &= (\underline{4n^2 + 2n + 5}) + 8n + 6 \\
 \text{(IV)} &\leq \underline{5n^2} + 8n + 6 \\
 &\leq \underline{5n^2} + 10n + 5 \\
 &= \underline{5 \cdot (n+1)^2}
 \end{aligned}$$

✓

O-Notation: Theorie

Theorem (Grenzwerte und die O-Notation)

1. Ist $\frac{f_n}{g_n}$ konvergent, folgt $f_n \in O(g_n)$
2. Gilt $\lim_{n \rightarrow \infty} \frac{f_n}{g_n} = \infty$, folgt $f_n \notin O(g_n)$.

Beweis (zu 1.)

$\frac{f_n}{g_n}$ sei konvergent

$\rightarrow \frac{f_n}{g_n}$ ist **beschränkt** (siehe Analysis).

\rightarrow Es gibt eine **Schranke** $c \in \mathbb{R}$, so dass $\frac{f(n)}{g(n)} \leq c$ für alle $n \in \mathbb{N}$.

\rightarrow Es gibt ein $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$, so dass $\frac{f(n)}{g(n)} \leq c$ für alle $n \geq n_0$.

\rightarrow Es gibt ein $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

$\rightarrow f \in O(g)$.

O-Notation: Beweis (Variante 2)



$$\cancel{4n^2 + 2n + 5} \rightarrow n^2$$

Wir zeigen per **Folgengrenzwert**: $\underbrace{4n^2 + 2n + 5}_{f} \in O(n^2)$.

$$\frac{f}{g} = \frac{4n^2 + 2n + 5}{n^2} = \frac{\cancel{n^2}}{\cancel{n^2}} \cdot \frac{4 + \cancel{2/n} + \cancel{5/n^2}}{1} \xrightarrow[n \rightarrow \infty]{\text{u.a.}} 4$$

$\Rightarrow f/g$ ist konvergent

$\Rightarrow f \in O(g)$.

O-Notation: Beweis (Variante 2)



Die O-Notation: Weitere Symbole

Definition ($f \in o(g)$)

Ist $\frac{f_n}{g_n}$ eine Nullfolge, schreiben wir $f_n \in o(g_n)$.

Anmerkungen

- Umgangssprachlich: "f wächst deutlich langsamer als g".
- **Beispiel:** $n \in o(n^2)$, aber $2n^2 \notin o(n^2)$.

Definition ($f \in \Theta(g)$)

Gilt sowohl $f_n \in O(g_n)$ als auch $g_n \in O(f_n)$, so schreiben wir $f_n \in \Theta(g_n)$.

Anmerkungen

- Ugs.: "f und g wachsen ungefähr gleich schnell."
- **Beispiel:** $2n^3 \in \Theta(n^3)$, aber $2n^2 \notin \Theta(n^3)$.



Die O-Notation: Weitere Symbole

Definition ($f \in \Omega(g)$)

$f \in \Omega(g)$ gilt genau dann wenn $g \in O(f)$.

Anmerkungen

- Ugs.: “ f wächst nicht deutlich langsamer als g .”
- **Beispiel:** $2n^3 \in \Omega(n^2)$, aber $2n \notin \Omega(n^2)$.

Do-Aufwandsklassen–Yourself



a_n	$1000n^3$	2^n	$\log_2(n)$
b_n	n^4		1
$a_n \in O(b_n) ?$		✓	
$b_n \in O(a_n) ?$		✓	
$a_n \in o(b_n) ?$		—	
$b_n \in o(a_n) ?$		✓	
$a_n \in \Omega(b_n) ?$		—	
$b_n \in \Omega(a_n) ?$		✓	
$a_n \in \Theta(b_n) ?$		✓	
$b_n \in \Theta(a_n) ?$		✓	

O-Notation: Rechenregeln

Theorem (O-Notation: Nützliche Rechenregeln)

$$O(c \cdot f_n) = O(f_n)$$

Ignorieren von Konstanten

$$O(f_n + c) = O(f_n)$$

$$O(f_n + g_n) = O(\max(f_n, g_n))$$

Stärkster Summand zählt

$$O(a_k n^k + a_{k-1} n^{k-1} + \dots + a_0) = O(n^k)$$

Polynome

$$(f \in O(g) \text{ und } g \in O(h)) \rightarrow f \in O(h)$$

Transitivität

$$(f \in O(h) \text{ und } g \in O(k)) \rightarrow f \cdot g \in O(h \cdot k)$$

Multiplikation

...



Outline

1. Beispiel: Lineare Suche
2. Die O-Notation
3. Code-Abschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche



Die O-Notation und Code-Primitive

In der Praxis können wir mit Hilfe der O-Notation die Aufwandsklasse von Code bestimmen (*hier für den Worst Case*).

1. Einmalige Ausführung

- ▶ Initialisierung und “Aufräumen” bestehen häufig aus Einzelbefehlen (z.B. *Variablen-Initialisierungen*).
- ▶ Deren Aufwand ist **konstant** ($O(1)$) und somit **vernachlässigbar**.

```
# Lineare Suche
```

1. pos = 0
2. while pos < n and a[pos] != s:
3. pos = pos+1
4. return pos

```
// Algorithmus A, Sequenz
```

```
f  
g  
h
```

$$\begin{aligned}O(A) &= O(f + g + h) \\&= O(\max(f,g,h))\end{aligned}$$

2. Sequenzen

- ▶ Bei hintereinander ausgeführten Algorithmenteilen zählt nur der Aufwändigste.
- ▶ Siehe auch unsere Rechenregel:
 $O(f + g) = O(\max(f, g))$

Die O-Notation und Code-Primitive

3. Verzweigung

- ▶ Gesamtkosten = Kosten für Prüfen der if-Bedingung, plus Kosten des teureren Zweigs.

4. Schleifen

- ▶ Aufwand für Ausführung eines Durchlaufs
 × # Durchläufe
- ▶ Gilt für While-Schleifen und For-Schleifen.

```
// Algorithmus B, Verzweigung
if t:
    f
else:
    g
    h
```

$$\begin{aligned}O(B) &= O(t) + O(\max(f,g)) + O(h) \\&= O(\max(t,f,g,h))\end{aligned}$$

```
// Algorithmus A, While-Schleife
while t:
    f
```

$$O(A) = O((t+f) \cdot \# \text{Durchläufe})$$

```
// Algorithmus A, For-Schleife
for i in 1...n:
    f
```

$$O(A) = O(f \cdot n)$$

Beispiel: Lineare Suche

```
1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
```

Abschätzung (Worst Case)

- ▶ Initialisierung und “Aufräumen” (Zeile 1+4) besitzen konstanten Aufwand
→ $O(1)$
- ▶ Kosten je Schleifendurchlauf (Zeile 2+3) → $O(1)$
 - ▶ Prüfung der Schleifenbedingung: $O(1)$
 - ▶ ggfs. Ausführung des Schleifenkörpers: $O(1)$
- ▶ Anzahl der Durchläufe (Worst Case) → $O(n)$
- ▶ Gesamtkosten

$$O(1) + O(1) \cdot O(n) = O(n)$$

- ▶ Der Algorithmus “Lineare Suche” besitzt **lineare Komplexität**.

Beispiel: Duplicate Checker

```
1 for i = 1..n:  
2     if linear_search(a, i):  
3         return True  
4     return False
```

```
1 function linear_search(a, i):  
2     suchwert = a[i]  
3     for j = i+1..n:  
4         if a[j]==suchwert:  
5             return True  
6     return False
```

Algorithmus

- ▶ **Gegeben:** Array a der Länge n (*beginnt bei $a[1]$*).
- ▶ **Ergebnis:** True wenn a mind. einen Wert mehrfach enthält.
- ▶ Wir ermitteln die **Komplexität** im **Worst Case**:

Beispiel: Duplicate Checker

```
1 for i = 1..n:  
2     if linear_search(a, i):  
3         return True  
4  
5     return False
```

```
1 function linear_search(a, i):  
2     suchwert = a[i]  
3     for j = i+1..n:  
4         if a[j]==suchwert:  
5             return True  
6     return False
```



Beispiel: Duplicate Checker

```
1 for i = 1..n:  
2     if linear_search(a, i):  
3         return True  
4  
5     return False
```

```
1 function linear_search(a, i):  
2     suchwert = a[i]  
3     for j = i+1..n:  
4         if a[j]==suchwert:  
5             return True  
6     return False
```



Outline

1. Beispiel: Lineare Suche
2. Die O-Notation
3. Code-Abschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche

Einige wichtige Aufwandsklassen

Theorem (Wichtige Aufwandsklassen)

Die folgende Aufstellung zeigt typische Aufwandsklassen, die bei Algorithmen (Suchen, Sortieren, Planen, ...) häufig auftreten:

Klasse	Name	Beispiel-Algorithmus (Worst-case-Aufwand)
$O(1)$	konstant	<i>Suche in Array der Länge 42</i>
$O(\log n)$	logarithmisch	<i>Suche in balanciertem Baum</i>
$O(n)$	linear	<i>lineare Suche</i>
$O(n \cdot \log n)$	linearithmisch	<i>Mergesort</i>

Einige wichtige Aufwandsklassen

Theorem (Wichtige Aufwandsklassen (Cont'd))

Klasse	Name	Beispiel-Algorithmus (Worst-case-Aufwand)
$O(n^2)$	quadratisch	<i>Insertionsort</i>
$O(n^p)$	polynomiell	<i>Multiplikation von zwei $n \times n$-Matrizen</i>
$O(2^n)$	exponentiell	<i>Naiver SAT-Solver (n Variablen)</i>
$O(n!)$	Fakultät	<i>Naiver TSP-Solver (n Städte) $O(n!)$</i>

Aufwandsklassen: Verhalten

n	$\log n$	n	$n \cdot \log(n)$	n^2	n^3	2^n
10	3,32	10	33,22	100	1000	1024
100	6,64	100	66,44	10000	10^6	$1,27 \cdot 10^{30}$
1000	9,97	1000	9966	10^6	10^9	10^{301}
10000	13,29	10000	132877	10^8	10^{12}	10^{3010}

- ▶ Algorithmen bis zu linearithmischer Komplexität sind in der Regel **effizient**.
- ▶ Höhergradig polynomiale Komplexität ist bei moderaten Problemgrößen noch **praktisch handhabbar**.
- ▶ **Exponentielle** Algorithmen sind **nicht praktikabel**.
→ **rote Linie** = Grenze des praktisch Machbaren.

O-Notation: Diskussion



Sinn der O-Notation

- ▶ Die O-Notation dient uns als **erste Näherung**, um schnell das Skalierbarkeitsverhalten von Algorithmen **abzuschätzen**.

Grenzen der O-Notation

- ▶ In der Praxis sind konstante Faktoren oft **nicht vernachlässigbar** (engl.: “*constants matter*”). **Beispiel:** Es ist wichtig ob ein Mausklick in 0.2 oder 2 Sekunden verarbeitet wird.
- ▶ Wenn die **Problemgröße gering** ist, kann ein laut O-Notation schlechteres Verfahren besser sein.
- ▶ Spezielle **Hardware** wird nicht berücksichtigt (*Bsp. GPUs im Deep learning*).
- ▶ Oft betrachtet man den Worst Case, der **Average Case** kann aber wichtiger sein (*Bsp. Quicksort, Simplex*).



Outline

1. Beispiel: Lineare Suche
2. Die O-Notation
3. Code-Abschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche



Die binäre Suche

- ▶ Bisher: Komplexität einzelner **Algorithmen**.
- ▶ **Achtung:** Ein **Problem** kann durch **viele verschiedene Algorithmen** gelöst werden!

Definition (Komplexität eines Problems)

*Die Komplexität eines Problems entspricht der Komplexität des **effizientesten Algorithmus**, der das Problem löst.*

Anmerkungen

- ▶ Die Komplexität eines Problems ist **schwierig** zu bestimmen. *Gibt es bessere Algorithmen als die uns bekannten?*

Die Komplexität des Problems “Suchen”

- ▶ Wir nehmen nun an, das Feld $a[0], \dots, a[n - 1]$ sei **sortiert**.
- ▶ Hier ist das Suchproblem **maximal $O(n)$** (*lineare Suche*).
- ▶ **Frage:** Geht es noch besser?

Die binäre Suche: Ansatz

Suche nicht von links nach rechts, sondern prüfe ob das Element in der **Mitte** $a[m]$ (mit $m = \lfloor \frac{n}{2} \rfloor$) dem Suchwert s entspricht.

- ▶ Ist $a[m] = s$, brechen wir ab (Erfolg).
- ▶ Ist $a[m] < s$, suchen wir in der rechten Hälfte weiter.
- ▶ Ist $a[m] > s$, suchen wir in der linken Hälfte weiter.

Beispiel (Suchwert $s=8$)

1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9
1	2	2	4	6	6	7	9

Prüfe das mittlere Element $a[m]$...

$a[m] < s \rightarrow$ rechte Hälfte ...

Prüfe das mittlere Element ...

$a[m] < s \rightarrow$ rechte Hälfte ...

...

...

Abbruch.

Die binäre Suche: Algorithmus

```
# Binäre Suche
function binsearch(links, rechts):
    if rechts < links:                                # Fall 1: Suche erfolglos
        return n # Länge des Arrays
    m = (links+rechts)/2
    if s == a[m]:                                     # Fall 2: Suchwert gefunden
        return m
    else if s < a[m]:                                 # Fall 3a: Suche links weiter
        return binsearch(links, m-1)
    else                                              # Fall 3b: Suche rechts weiter
        return binsearch(m+1, rechts)

return binsearch(0, n-1)
```

- ▶ **Rekursive** Methode `binsearch`, mit zwei Parametern `links` und `rechts`. Diese markieren die **linke und rechte Grenze** des noch zu durchsuchenden Array-Teils.
- ▶ Initialer Aufruf (unten): `binsearch(0, n-1)`.
- ▶ Suchen in **linker/rechter Hälfte** mit rekursivem Aufruf (3a,3b).
- ▶ Mit jedem rekursiven Aufruf **halbtiert** sich die Problemgröße.

Die binäre Suche: Komplexität

Laufzeit $f(n)$?

- ▶ Ein Durchlauf von `binsearch()` (ohne Rekursion): konstante Kosten c .
- ▶ Rekursiver Aufruf führt zu **halb so großem Problem**: $f(n/2)$.
- ▶ **Abbruchbedingung** (Bereich besitzt Breite 1): Kosten c' .
- ▶ Wir erhalten eine sogenannte **Rekurrenzgleichung**:

$$f(n) = f(n/2) + c$$

$$f(1) = c'$$

```
# Binäre Suche
function binsearch(links, rechts):
    if rechts < links:                                # Fall 1: Suche erfolglos
        return n # Länge des Arrays
    m = (links+rechts)/2
    if s == a[m]:                                     # Fall 2: Suchwert gefunden
        return m
    else if s < a[m]:                                 # Fall 3a: Suche links weiter
        return binsearch(links, m-1) # Fall 3a: Suche links weiter
    else
        return binsearch(m+1, rechts) # Fall 3b: Suche rechts weiter

return binsearch(0, n-1)
```

Die binäre Suche: Komplexität

*

Sei $n = 2^k$ eine Zweierpotenz.

$$f(n) = f(n/2) + c \quad (*)$$

$$f(1) = c' \quad (*)$$

$$n=1 \quad \square \quad f(1) = c' \quad (*)$$

$$n=2 \quad \square\square \quad f(2) = c + f(1) = c + c'$$

$$n=4 \quad \square\square\square\square \quad f(4) = c + f(2) = c + c + c'$$

$$n=8 \quad \overline{\square\square\square\square\square\square\square} \quad f(8) = c + f(4) = c + c + c + c' \\ 2^3 \qquad \qquad \qquad \underbrace{c+c+c}_{2 \cdot c}$$

$$\vdots$$

$$f(2^k) = k \cdot c + c'$$

$$f(n) = \cancel{\log_2(n)} \cdot c + c' \in O(\log n)$$

Die binäre Suche: Komplexität

Für allgemeine n ("Nicht-Zweierpotenzen"):

Schätzung mit der nächsten Zweierpotenz ab ($n=103 \approx n=128$)

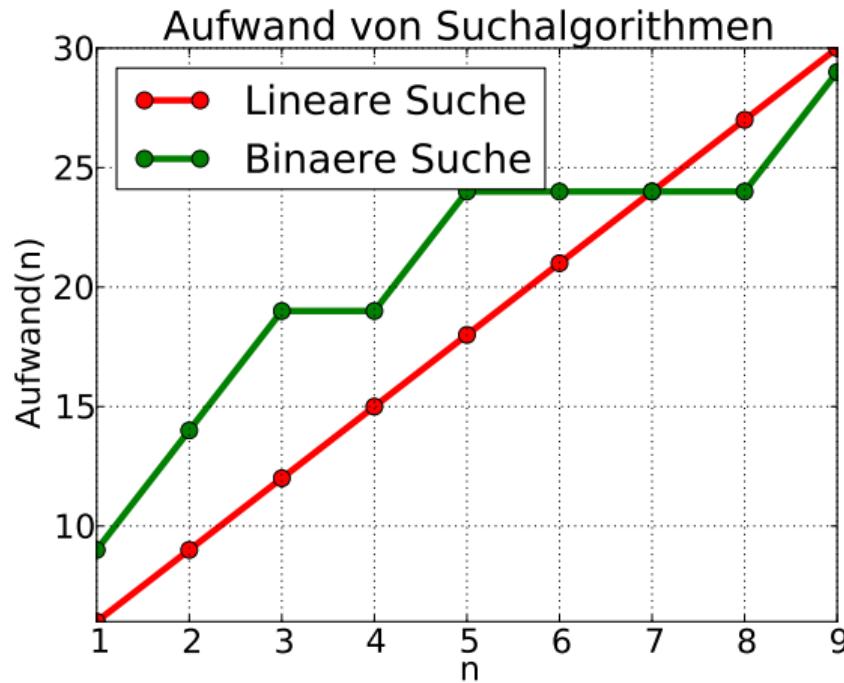
$$f(n) = \lceil \log_2(n) \rceil \cdot c + c' \in O(\log n) \quad \checkmark$$

Die binäre Suche: Komplexität



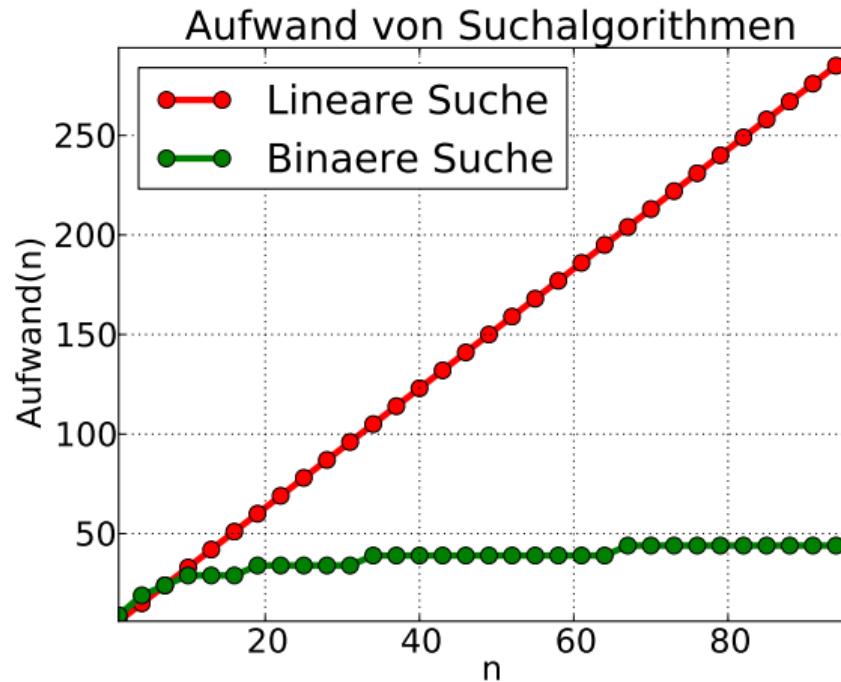
Lineare Suche vs. binäre Suche

Wir plotten die Laufzeit von linearer Suche ($O(n)$) und binärer Suche ($O(\log(n))$) gegen die Problemgröße n :



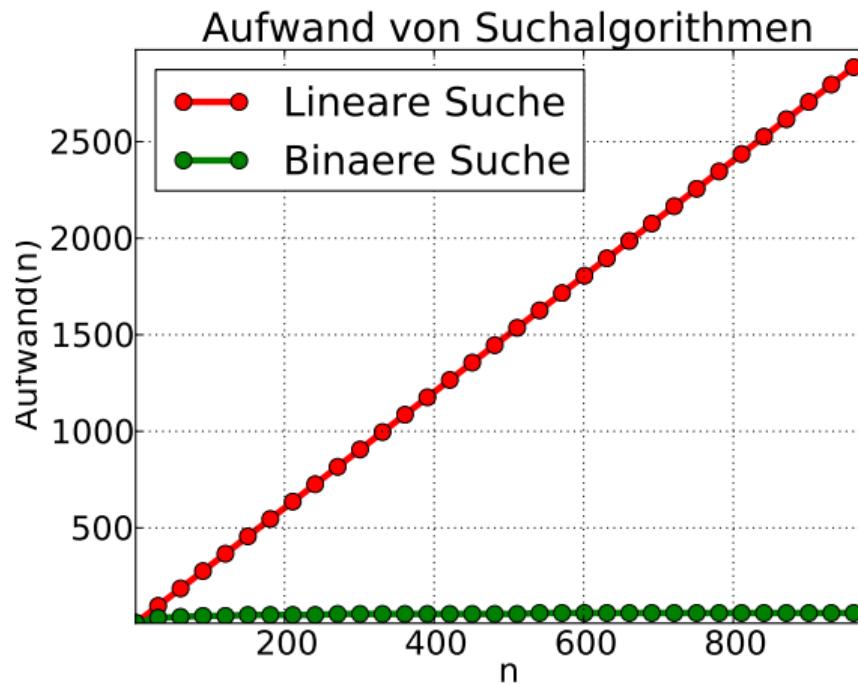
Lineare Suche vs. binäre Suche

Mit wachsendem n wird der Vorteil der binären Suche deutlich:



Lineare Suche vs. binäre Suche

Mit wachsendem n wird der Vorteil der binären Suche deutlich:



References I

