

Algorithmen und Datenstrukturen

– Wintersemester 2019 –

Kapitel 02: Komplexitätsanalyse

Fachbereich Informatik
TU Kaiserslautern

Dozent: Dr. Patrick Michel

Folien ursprünglich von Prof. Dr. Adrian Ulges (Hochschule RheinMain)

1

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;
            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;

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

Kosten von Algorithmen

In ADS verfolgen wir Ansatz 2:

- ▶ Wir führen den Algorithmus gedanklich auf einer Maschine mit **bestimmten 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, unabhängig 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;
            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) {

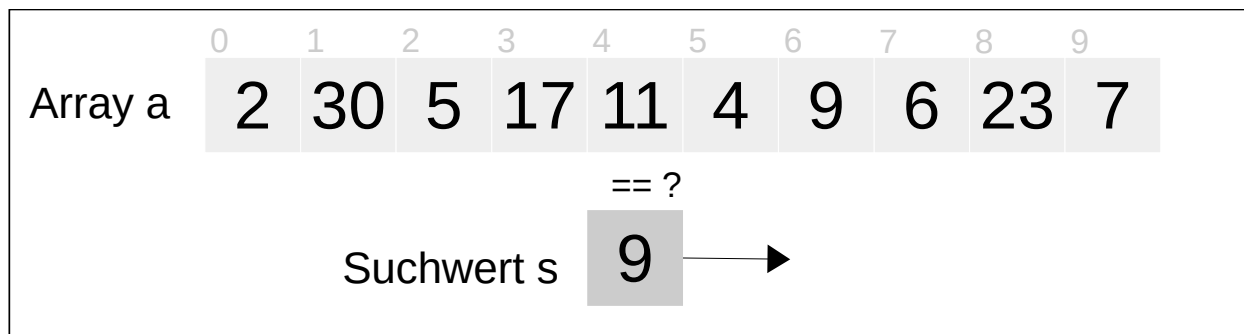
        for(int k=0; k<numbers.length;

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

Outline

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

Beispiel: Lineare Suche



Problemstellung

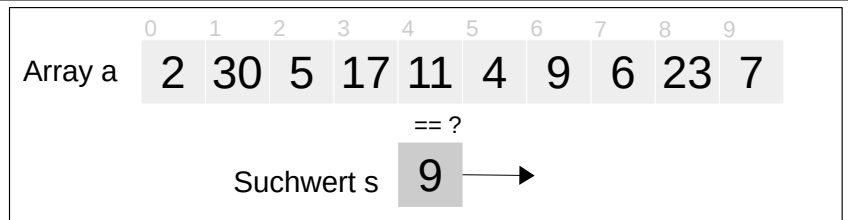
- ▶ Gegeben: Ein **Array** $a[0], a[1], \dots, a[n-1]$, ein **Suchwert** s .
- ▶ Gebe die **Position** zurück, an der der Suchwert im Array vorkommt. Ist der Wert **nicht** vorhanden, gebe n zurück.

Ansatz

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

5

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).
- ▶ Je Durchlauf: Kosten 5.
(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.**

6

Effizienz von Algorithmen: Formalisierung

Generellere Aussage: Abstrahiere 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 kann (je nach Art des zu lösenden Problems) **verschiedene Dinge** bezeichnen:

- ▶ 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.
- ▶ ...

7

(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')

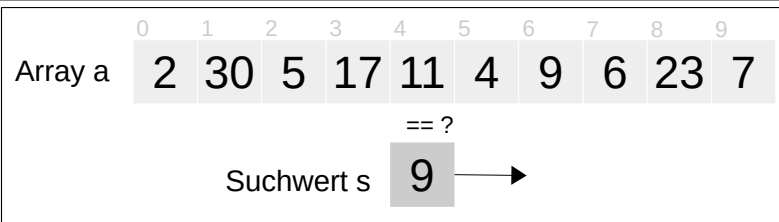
- ▶ Betrachte **alle möglichen Eingaben** (der Größe n) und **middle** die Anzahl der benötigten Schritte.
- ▶ Dies ist meist **relevant**, aber **schwierig** zu berechnen.

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.

8

Beispiel: Lineare Suche



Pseudocode

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

Best Case

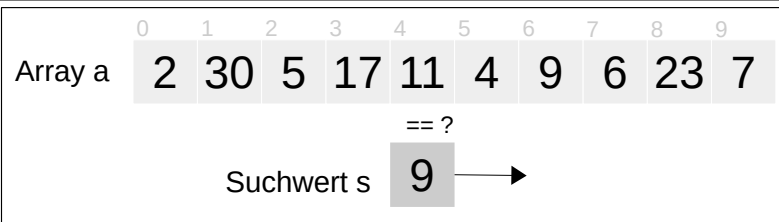
- ▶ Suchwert befindet sich **an der 1. Position** im Array.
- ▶ Kosten: 4 (1 Zuweisung (Zeile 1), 2 Vergleiche & 1 Feldzugriff (Zeile 2))

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: $2 + 5 \cdot n$.**

9

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").

10

Aufwandsschätzung: Do-it-Yourself



Berechnen Sie den **Worst-Case-Aufwand** des folgenden Algorithmus. Zählen Sie nur die **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
4     while pos <= i:
5         result += a[pos]
6         pos += 1
7     return result
```

11

Outline

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

12

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):

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

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

13

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:

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

- ▶ Warum? Weil für große n der **relative Fehler vernachlässigbar** ist (hier für $n=10000$: 0.005%).

14

Vereinfachung von Kostenfunktionen

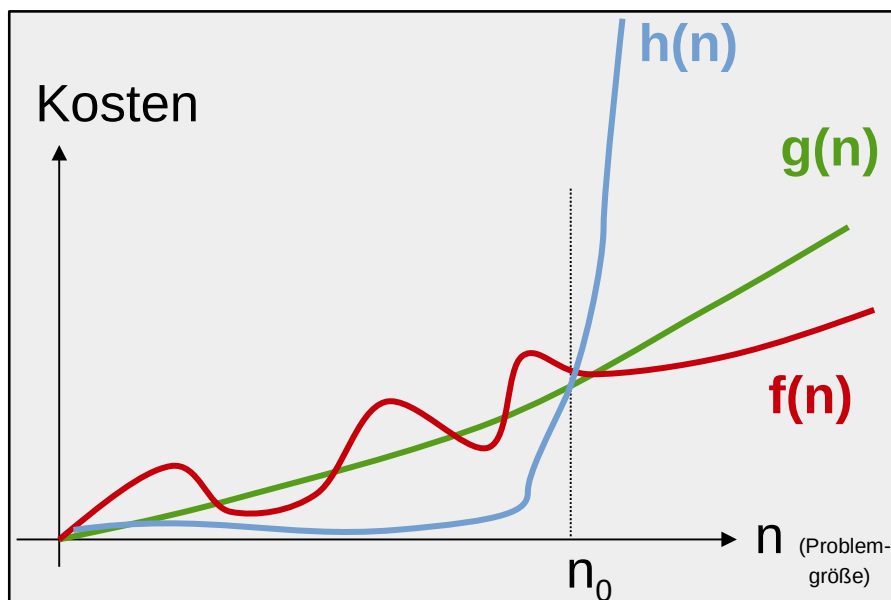
Schritt 2: Faktoren entfernen

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

- ▶ 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*).

15

O-Notation: Illustration



- ▶ 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 schneller als g).

16

Definition: O-Notation

Definition (O-Notation)

Es seien f und g zwei Kostenfunktionen. Wenn es eine Konstante $c \in \mathbb{R}$ 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 f .
- ▶ Wir sprechen: “ f ist von der Ordnung g ” oder auch “**f ist O von g**”.

17

Definition: O-Notation

Anmerkungen (cont'd)

- ▶ 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**).
- ▶ 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 $n^2 = O(n)$.

18

Definition: O-Notation

Beispiel-Klassen

- ▶ “linear”: n , $1000n + 3$
- ▶ “quadratisch”: n^2 , $7n^2 + 5n - 10$
- ▶ “logarithmisch”: $\log_2(n)$, $\log_3(n)$, $\log_8(n) + 4$
- ▶ “exponentiell”: 2^n , $2^n + n^{10000} + 100000$

Weitere Anmerkungen

Wir unterscheiden im Folgenden zwischen der ...

- ▶ **Laufzeit** eines Algorithmus f_n (= exakte Anzahl an Rechenschritten, umständlich zu berechnen).
- ▶ **Komplexität** $O(f_n)$ (= grobe Abschätzung, leicht zu berechnen, “genau genug”).

Man sollte die Komplexität möglichst präzise angeben.

Beispiel: Für $f_n = 2n$ gilt $f_n \in O(2^n)$, aber auch $f_n \in O(n)$ (besser!).

19

O-Notation: Rechenregeln

Theorem (O-Notation: Rechenregeln)

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

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

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

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

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

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

...

*Ignorieren von
Konstanten*

*Stärkster Sum-
mand zählt*

Polynome

Transitivität

Multiplikation

20

Outline

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

21

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
```

2. Sequenzen

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

```
// Algorithmus A, Sequenz
f
g
h
```

```
O(A) = O(f + g + h)
      = O(max(f,g,h))
```

22

Die O-Notation und Code-Primitive

3. Verzweigung

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

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

$$O(B) = O(t) + O(\max(f,g)) + O(h) \\ = O(\max(t,f,g,h))$$

4. Schleifen

- ▶ Aufwand für Ausführung eines Durchlaufs \times # Durchläufe
- ▶ Gilt für While-Schleifen und For-Schleifen

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

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

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

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

23

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 $\rightarrow O(1)$
- ▶ **Kosten je Schleifendurchlauf** (Zeile 2+3) $\rightarrow O(1)$
 - ▶ Prüfung der Schleifenbedingung: $O(1)$
 - ▶ ggfs. Ausführung des Schleifenkörpers: $O(1)$
- ▶ **Anzahl der Durchläufe (Worst Case)** $\rightarrow O(n)$
- ▶ **Gesamtkosten**

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

- ▶ Der Algorithmus "Lineare Suche" besitzt **lineare Komplexität**.

24

Beispiel: Duplicate Checker

```
1 for i = 1..n:  
2  
3     if linear_search(a, i):  
4         return True  
5  
6     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**:

25

Outline

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

26

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)
$O(1)$	<i>konstant</i>	<i>Suche in Array der Länge 42</i>
$O(\log n)$	<i>logarithmisch</i>	<i>Suche in balanciertem Baum</i>
$O(n)$	<i>linear</i>	<i>lineare Suche</i>
$O(n \cdot \log n)$	<i>linearithmisch</i>	<i>Mergesort</i>
$O(n^2)$	<i>quadratisch</i>	<i>Insertionsort</i>

27

Einige wichtige Aufwandsklassen

Theorem (Wichtige Aufwandsklassen (Cont'd))

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

28

Aufwandsklassen: Verhalten

n	log n	n	n · 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 · 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 polynomielle Komplexität ist bei moderaten Problemgrößen noch **praktisch handhabbar**.
- ▶ **Exponentielle** Algorithmen sind **nicht praktikabel**.
→ **rote Linie** = Grenze des praktisch Machbaren.

Quelle: Hartmann – “Mathematik für Informatiker”

29

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. Aufwandsabschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche

31

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?

32

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

Prüfe das mittlere Element $a[m]$

...

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

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

Prüfe das mittlere Element ...

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

...

...

Abbruch.

33

Die binäre Suche: Algorithmus

```
# Binäre Suche
function binsearch(links, rechts):
    if rechts < links:                # Fall 1: Suche erfolglos
        return 0
    m = (links+rechts)/2
    if s == a[m]:                    # Fall 2: Suchwert gefunden
        return m
    else if s < a[m]:
        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)
```

- ▶ **Rekursive** Methode binsearch, die zwei Integer-Werte links und rechts erhält. Diese markieren die **linkeste und rechteste Position** des noch zu durchsuchenden Teils des Arrays.
- ▶ Initialer Aufruf (unten): binsearch(0, n-1).
- ▶ Suchen in **linker/rechter Hälfte** mit rekursivem Aufruf (3a,3b).
- ▶ Mit jedem rekursiven Aufruf **halbiert** sich die Problemgröße.

34

Die binäre Suche: Komplexität

```
# Binäre Suche
function binsearch(links, rechts):
    if rechts < links: # Fall 1: Suche erfolglos
        return 0
    m = (links+rechts)/2
    if s == a[m]: # Fall 2: Suchwert gefunden
        return m
    else if s < a[m]:
        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)
```

Laufzeit $f(n)$?

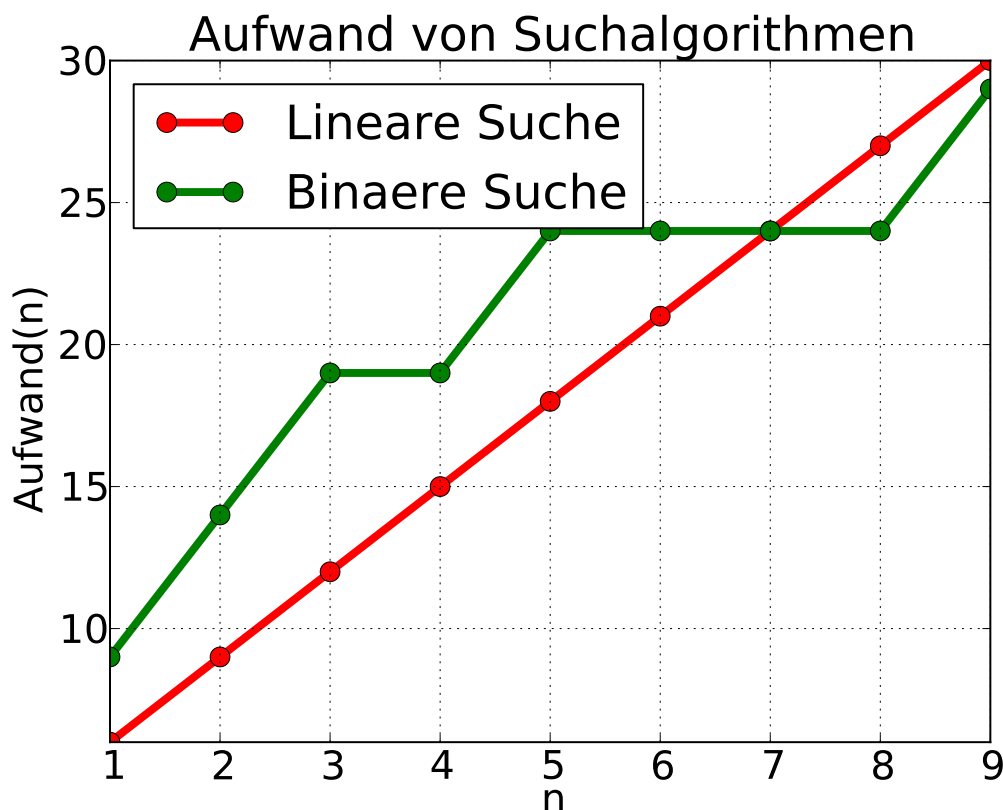
- ▶ Ein Durchlauf von `binsearch()` besitzt konstante Kosten c .
- ▶ Rekursiver Aufruf führt zu **halb so großem Problem**: $f(n/2)$.
- ▶ **Abbruchbedingung** (Bereich besitzt Breite 1): Kosten c' .
- ▶ Wir erhalten eine sogenannte **Rekurrenzgleichung**:

$$f(n) = f(n/2) + c$$
$$f(1) = c'$$

35

Lineare Suche vs. binäre Suche

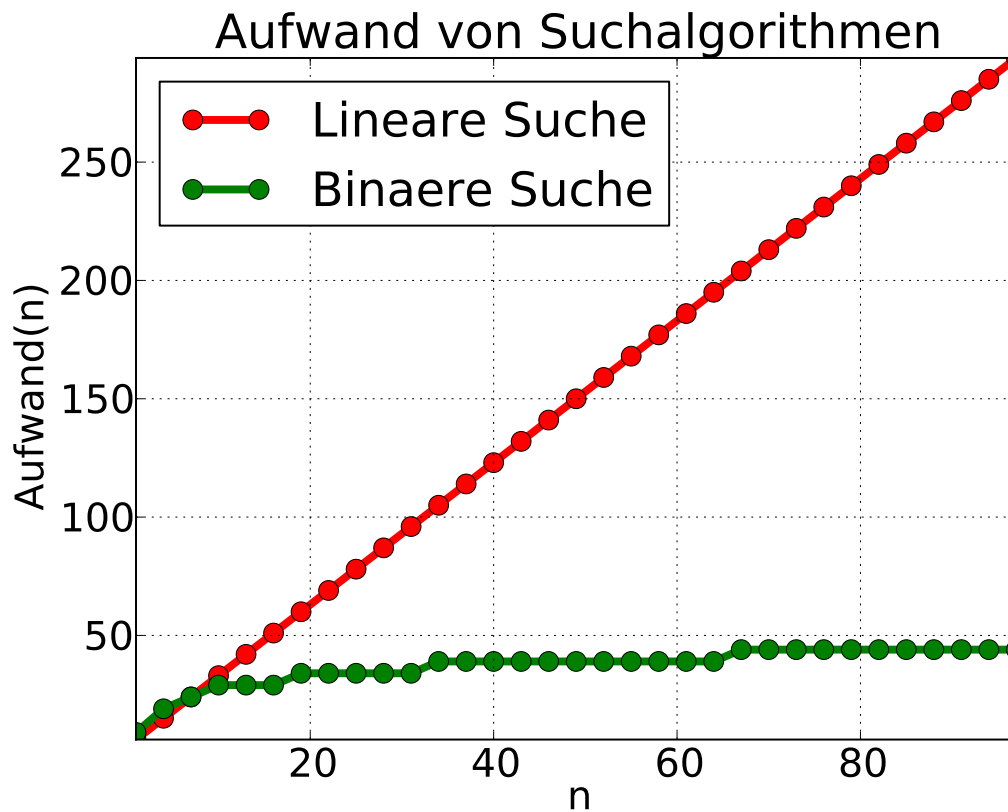
Wir plotten die Laufzeit von linearer Suche ($O(n)$) und binärer Suche ($O(\log(n))$) gegen die Problemgröße n :



36

Lineare Suche vs. binäre Suche

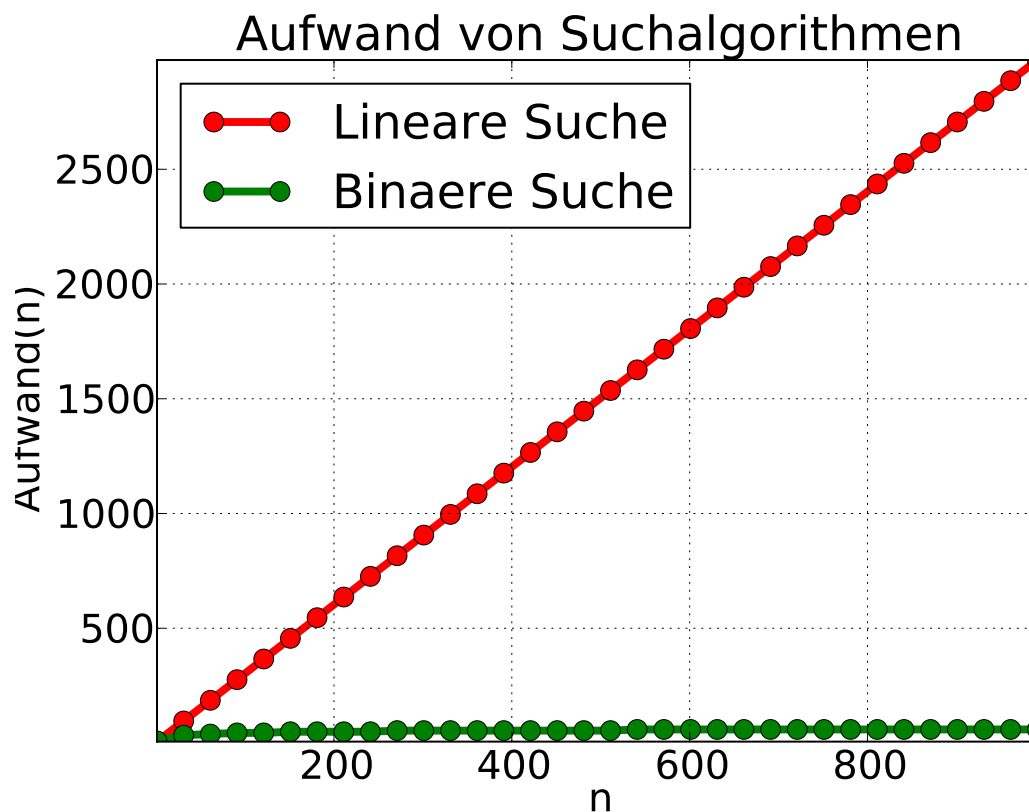
Mit wachsendem n wird der Vorteil der binären Suche deutlich:



37

Lineare Suche vs. binäre Suche

Mit wachsendem n wird der Vorteil der binären Suche deutlich:



38