Algorithmen – eine kurze Einführung

Einführung in die Programmierung 1

Sommersemester 21

TU Wien

Überblick

- Algorithmen allgemein
- Einführendes Beispiel
- Komplexitätsanalyse von Algorithmen
- Typische Komplexitätsfunktionen

Algorithmen allgemein

Beispiele für Algorithmen in dieser Vorlesung

- Bisher schon behandelte Beispiele
  - Euklidischer Algorithmus (ggT-Berechnung)
  - Zahlensummen bilden
  - Sieb des Eratosthenes
  - Matrizenmultiplikation
  - ...

Definition

*Ein Algorithmus ist ein **endliches**, **schrittweises** Verfahren zur **Berechnung** von gesuchten aus gegebenen Größen, in dem jeder **Schritt** aus einer Anzahl **ausführbarer**, **eindeutiger** Operationen und einer **Angabe über den nächsten Schritt** besteht.*

[Peter Rechenberg, *Was ist Informatik?: Eine allgemeinverständliche Einführung*, Hanser Fachbuch, 3 Auflage, 2000]

Endlichkeit

- Statische Endlichkeit
  - Algorithmus muss mit **endlich vielen Zeichen** beschreibbar sein
- Dynamische Endlichkeit
  - Ein Algorithmus darf zu jedem Zeitpunkt während seiner Ausführung nur **endlich viel Speicherplatz** benötigen

Algorithmus und Berechnung

- Ein Algorithmus transformiert Eingabedaten in Ausgabedaten („berechnet“ Ausgabedaten)
- Der Begriff **Berechnung** ist dabei allgemeiner zu verstehen, z. B.
  - Berechnungen im klassischen Sinn (Addition, Subtraktion etc.)
  - Manipulation von Texten
  - Verarbeitung von beliebigen Signalen (z. B. Bilder) oder Sequenzen (von Zahlen etc.)
- Annahme für diese Vorlesung
  - Die Eingabedaten stehen schon zu Beginn fest

Eindeutigkeit und Ausführbarkeit

- Eindeutigkeit
  - Algorithmus besteht aus einer Folge von **eindeutigen Schritten**
  - Es darf keine Unklarheiten geben
  - Computer „denkt“ nicht mit, sondern führt die Befehle der Reihe nach aus
- Ausführbarkeit
  - Jeder Schritt muss **durch den Computer ausführbar** sein

Determiniertheit und Determinismus

- Determiniertheit
  - Ein Algorithmus liefert bei **gleichen Voraussetzungen** (Parameter, Startbedingungen) stets das **gleiche Ergebnis**
- Determinismus
  - Zu **jedem Zeitpunkt** der Ausführung besteht **höchstens eine Möglichkeit** der Fortsetzung
  - Der gesamte Ablauf ist eindeutig bestimmt
- Es gilt
  - Jeder deterministische Algorithmus ist determiniert
  - Nicht jeder determinierte Algorithmus ist deterministisch
    - Beispiele folgen noch

Terminiertheit

- Algorithmus muss für alle möglichen Eingaben nach **endlich vielen Verarbeitungsschritten** anhalten und ein Resultat liefern
- Hinweis
  - Ob ein (beliebiger) Algorithmus mit einer beliebigen Eingabe terminiert, ist nicht durch einen Algorithmus lösbar (Halteproblem)

Algorithmen und Problemlösen

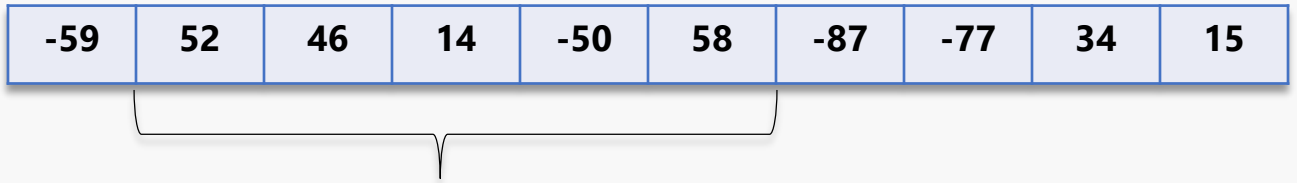
- Ein Algorithmus löst immer ein allgemeines Problem (eine Klasse von Problemen)
  - Problem beschreibt, was gelöst wird
  - Algorithmus beschreibt, wie es gelöst wird
- Anwendungsprogramme bestehen meist aus
  - Benutzerschnittstellen (Fenster, Menüs etc.)
  - Verwaltungsteile für das Verarbeiten (z. B. Lesen) von Daten,
  - Sammlung von Algorithmen zur Berechnung und Manipulation von Daten

Algorithmen machen oft nur einen kleinen Teil der Programme aus, sie bestimmen aber oft, wie effizient ein Programm ist

Einführendes Beispiel

Beispiel – Maximale Abschnittssumme

- Finden der maximalen Abschnittssumme in einem Array von *n* Zahlen (positive und negative Zahlen)
  - Ist die Teilfolge von aufeinanderfolgenden Zahlen, die unter allen möglichen Teilfolgen die größte Summe liefert
  - Beispiel



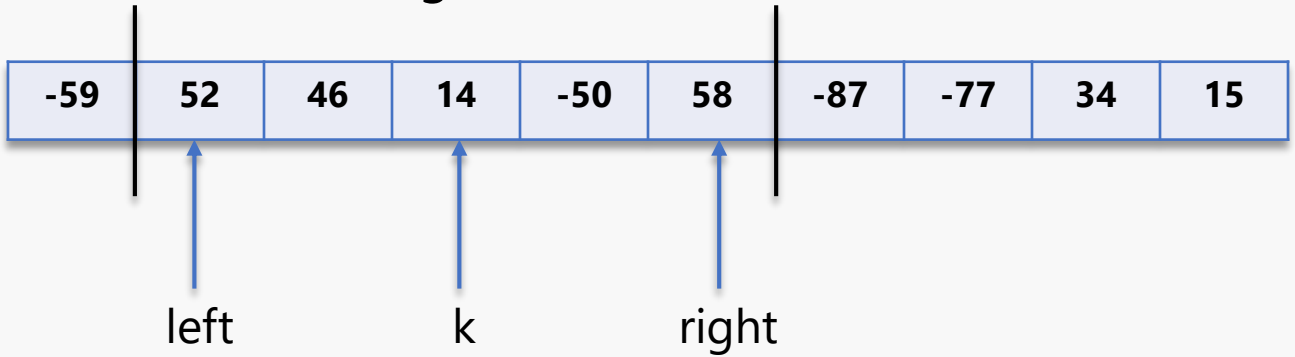
Maximale Summe = 52 + 46 + 14 + -50 + 58 = 120

Maximale Abschnittssumme – Spezialfälle

- Wenn alle Werte > 0 sind
  - Maximale Summe entspricht der Summe aller Elemente
- Wenn alle Werte < 0 sind
  - Maximale Summe entspricht dem Element mit dem kleinsten Absolutbetrag
- Ein leeres Array ist nicht zulässig
- Es kann mehrere Teilfolgen geben, die die maximale Summe ergeben
  - Es wird aber nur eine Summe ermittelt und zurückgeliefert

1. Variante – Idee

- Idee
  - Alle Teilfolgen bilden, die im Array enthalten sind
  - Die Werte einer Teilfolge aufaddieren
  - Die maximale von all diesen Summen bestimmen
- Benötigt werden
  - Linker und rechter Rand der Teilfolge (left, right) und aktuelle Position in der Teilfolge (k)



1. Variante – Implementierung

```
private static int maxSum1(int[] input) {
    int maxSum = input[0];
    for (int left = 0; left < input.length; left++) {
        for (int right = left; right < input.length; right++) {
            int sum = 0;
            for (int k = left; k <= right; k++) {
                sum += input[k];
            }
            if (sum > maxSum) {
                maxSum = sum;
            }
        }
    }
    return maxSum;
}
```

Beispiel für Ablauf (mit Ausgabe)

```java
private static int maxSum1(int[] input) {
    int maxSum = input[0];
    for (int left = 0; left < input.length; left++) {
        for (int right = left; right < input.length; right++) {
            int sum = 0;
            for (int k = left; k <= right; k++) {
                System.out.println("left = " + left + " right = " + right + " k = " + k + " with value: " + input[k]);
                sum += input[k];
            }
            if (sum > maxSum) {
                maxSum = sum;
            }
        }
    }
    return maxSum;
}
```

Erzeugt durch folgenden Aufruf:
maxSum1(**new int**[]{2, 3, -6, 4})

```
left = 0 right = 0 k = 0 with value: 2
left = 0 right = 1 k = 0 with value: 2
left = 0 right = 1 k = 1 with value: 3
left = 0 right = 2 k = 0 with value: 2
left = 0 right = 2 k = 1 with value: 3
left = 0 right = 2 k = 2 with value: -6
left = 0 right = 3 k = 0 with value: 2
left = 0 right = 3 k = 1 with value: 3
left = 0 right = 3 k = 2 with value: -6
left = 0 right = 3 k = 3 with value: 4
left = 1 right = 1 k = 1 with value: 3
left = 1 right = 2 k = 1 with value: 3
left = 1 right = 2 k = 2 with value: -6
left = 1 right = 3 k = 1 with value: 3
left = 1 right = 3 k = 2 with value: -6
left = 1 right = 3 k = 3 with value: 4
left = 2 right = 2 k = 2 with value: -6
left = 2 right = 3 k = 2 with value: -6
left = 2 right = 3 k = 3 with value: 4
left = 3 right = 3 k = 3 with value: 4
```

2. Variante – Idee

- 1. Variante ist ineffizient
  - Innerste Schleife (Summation der Folgenelemente) ist nicht notwendig
- Beispiel für Problem
  - *left*=2 und *right*=4
    - Die innerste Schleife addiert die Elemente mit Indizes 2, 3 und 4
  - Danach folgt *left*=2 und *right*=5
    - Die innerste Schleife addiert die Elemente mit Indizes 2, 3, 4 und 5
    - In diesem Fall müsste nur das Element am Index 5 zur vorherigen Summe addiert werden

2. Variante – Implementierung

```
private static int maxSum2(int[] input) {
    int maxSum = input[0];
    for (int left = 0; left < input.length; left++) {
        int sum = 0;
        for (int right = left; right < input.length; right++) {
            sum += input[right];
            if (sum > maxSum) {
                maxSum = sum;
            }
        }
    }
    return maxSum;
}
```

Beispiel für Ablauf (mit Ausgabe)

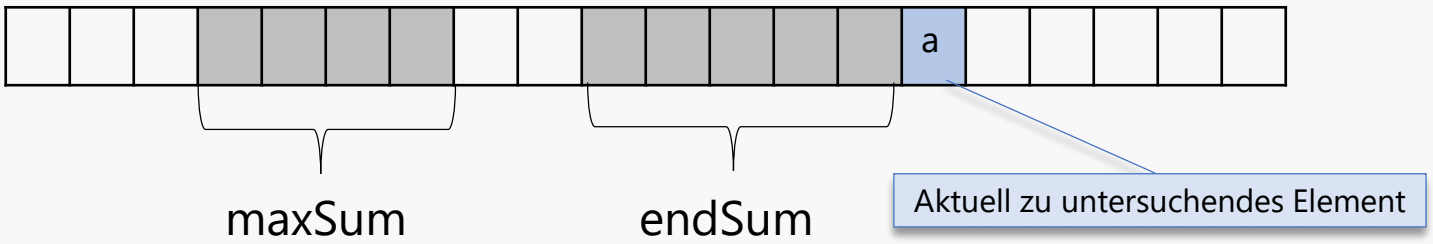
```java
private static int maxSum2(int[] input) {
    int maxSum = input[0];
    for (int left = 0; left < input.length; left++) {
        int sum = 0;
        for (int right = left; right < input.length; right++) {
            System.out.println("left = " + left + " right = " + right + " with value " + input[right]);
            sum += input[right];
            if (sum > maxSum) {
                maxSum = sum;
            }
        }
    }
    return maxSum;
}
```

Erzeugt durch folgenden Aufruf:
maxSum2(**new int**[]{2, 3, -6, 4})

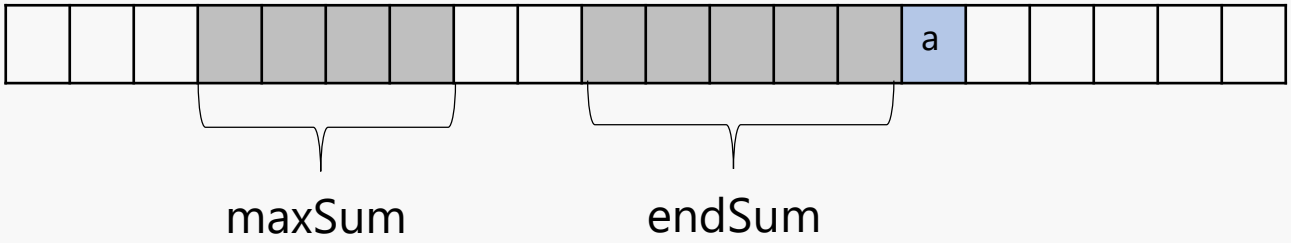
```
left = 0 right = 0 with value 2
left = 0 right = 1 with value 3
left = 0 right = 2 with value -6
left = 0 right = 3 with value 4
left = 1 right = 1 with value 3
left = 1 right = 2 with value -6
left = 1 right = 3 with value 4
left = 2 right = 2 with value -6
left = 2 right = 3 with value 4
left = 3 right = 3 with value 4
```

3. Variante – Idee (1)

- Array wird von links nach rechts einmal durchlaufen
- Informationen an der zu untersuchenden Stelle
  - Die maximale Summe einer Teilfolge im gesamten bisher inspizierten Anfangsstück (maxSum)
  - Die aktuelle an der zu untersuchenden Stelle endende Summe der aktuellen Teilfolge (endSum)



3. Variante – Idee (2)



- Maximale Teilsumme
  - endSum′ ist Maximum von endSum + a und a
  - maxSum′ ist Maximum von maxSum und endSum′

3. Variante – Implementierung

```
private static int maxSum3(int[] input) {
    int maxSum = input[0], endSum = maxSum;
    for (int pos = 1; pos < input.length; pos++) {
        endSum = Math.max(endSum + input[pos], input[pos]);
        maxSum = Math.max(maxSum, endSum);
    }
    return maxSum;
}
```

Beispiel für Ablauf (mit Ausgabe)

```java
private static int maxSum3(int[] input) {
    int maxSum = input[0], endSum = maxSum;
    for (int pos = 1; pos < input.length; pos++) {
        System.out.println("pos = " + pos + " with value " + input[pos]);
        endSum = Math.max(endSum + input[pos], input[pos]);
        maxSum = Math.max(maxSum, endSum);
    }
    return maxSum;
}
```

Erzeugt durch folgenden Aufruf:
maxSum3(**new int**[]{2, 3, -6, 4})

```
pos = 1 with value 3
pos = 2 with value -6
pos = 3 with value 4
```

Maximale Abschnittssumme – Vergleich

- Unterschied zwischen den drei Versionen?
  - Ergebnis: Kein Unterschied
  - Laufzeit für große Arrays: **Enorme Unterschiede**

Komplexitätsanalyse von Algorithmen

Effizienz vergleichen

- Messen
  - Wie lange braucht ein Algorithmus für eine bestimmte Eingabe der Größe *n* (Laufzeit)?
- Anzahl von Operationen zählen
  - Wie viele Operationen führt ein Algorithmus für eine bestimmte Eingabe der Größe *n* aus?
- Asymptotische Analyse
  - Wie lässt sich die Laufzeit als Funktion der Eingabegröße *n* beschreiben?

Messen

- Beispiele für maximale Abschnittssumme
- Vergleich der Laufzeiten
  - Hier Mittelwert von fünf Messungen pro Eingabegröße auf Rechner mit geringer Last
  - System: Intel Core i7 9750H, Windows 10, Java 14
- Verdopplungsexperiment
  - Eingabedaten werden immer verdoppelt
  - Laufzeiten wachsen entsprechend an
    - Die Unterschiede in den Implementierungen zeigen sich erst bei großen Werten für *n*
    - Bei kleinen Werten dominieren andere Effekte, z. B. Caching

Verdopplungsexperiment

Eingabegröße n	1. Variante (sec)	2. Variante (sec)	3. Variante (sec)
1000	0.085	0.003	0
2000	0.661	0.003	0
4000	2.532	0.004	0
8000	20.089	0.008	0
16000	159.936	0.034	0
32000	1280	0.139	0
64000	10240	0.555	0
128000	81920	2.08	0
256000	655360	8.318	0
512000	5242880	33.214	0
1024000	41943040	133.285	0,001
2048000	335544320	532	0,002
4096000	2684354560 (ca. 85 Jahre)	2128	0,004

Doppelte Datenmenge führt zu ca. achtfacher Laufzeit!
Rote Werte extrapoliert (sehr ungenau)

Doppelte Datenmenge führt zu ca. vierfacher Laufzeit!

Doppelte Datenmenge führt zu ca. doppelter Laufzeit!
Blaue Werte – nicht messbar oder sehr ungenau

Probleme mit Messungen

- Beziehen sich auf eine Hardware/Software-Umgebung
  - Unterschiedliche Werte auf unterschiedlichen Umgebungen
- Beziehen sich nur auf eine beschränkte Anzahl an Eingaben
  - Sind die verwendeten Eingaben (Eingabegrößen) repräsentativ?
- Algorithmus muss implementiert werden, um seine Laufzeit zu messen

Ziele einer alternativen Analyse

- Evaluierung von Algorithmen unabhängig von der Hardware/Software-Umgebung
- Kann auch auf allgemeinere Beschreibungen (Pseudocode) angewendet werden, die noch keine konkrete Implementierung darstellen
- Berücksichtigt alle möglichen Eingaben

Einfache Operationen zählen

- Einfache Operationen
  - Zuweisung
  - Arithmetische Operation
  - Vergleich
  - Zugriff auf ein Element eines Arrays (über Index)
  - Aufruf einer Methode
  - Rückkehr aus einer Methode
- Annahmen zur Ausführungszeit
  - Konstant für eine spezifische einfache Operation
  - Unterschied zwischen einzelnen Operationen gering (optimistisch bei Methodenaufrufen) und konstant

Beispiel (vereinfacht)

- Array input mit n Elementen

```
private static int maxSum3(int[] input) {
    int maxSum = input[0], endSum = maxSum;
    for (int pos = 1; pos < input.length; pos++) {
        endSum = Math.max(endSum + input[pos], input[pos]);
        maxSum = Math.max(maxSum, endSum);
    }
    return maxSum;
}
```

3 op

2 op

5 op

2 op

n-1 mal

1 op

Realistisch?
Aber: Genaue Anzahl pro Zeile nicht unbedingt wichtig.
Wichtig: Wie oft wird eine Zeile ausgeführt!

Operationen zählen

- Ist unabhängig von der Hardware
- Ist aber immer noch von der konkreten Implementierung abhängig
- Es ist auch unklar, was alles gezählt werden soll
  - Alle Operationen?
  - Nur bestimmte?
  - Nur die in der (innersten) Schleife?

Zählen für maxSum1 und maxSum2

- Wie oft wird der Code (`sum += input[…]`) in der inneren Schleife für n Elemente ausgeführt?
- maxSum1 und maxSum2

n	maxSum1
10	220
20	1540
30	4960
40	11480
50	22100
100	171700
1000	167167000

n	maxSum2
10	55
20	210
30	465
40	820
50	1275
100	5050
1000	500500

Alternatives Beispiel

- Verschiedene (aber gleich große) Inputs können das Verhalten eines Algorithmus beeinflussen
- Beispiel – Suche in einem Array

```
private static boolean search(int[] input, int element) {
    for (int i = 0; i < input.length; i++) {
        if (input[i] == element) {
            return true;
        }
    }
    return false;
}

public static void main(String[] args) {
    System.out.println(search(new int[]{1, 2, 3, 4, 5, 2, 3, 9}, 1));
    System.out.println(search(new int[]{1, 2, 3, 4, 5, 2, 3, 9}, 4));
    System.out.println(search(new int[]{1, 2, 3, 4, 5, 2, 3, 9}, 6));
}
```

Unterschiedliche Fälle – es wird nicht immer alles durchsucht

```
true
true
false
```

Unterschiedliche Fälle

Bester Fall

(Best Case)

- Ist die Laufzeit eines Algorithmus bei der bestmöglichen Eingabe mit Größe **n**.
- z. B. Suche: Gesuchter Wert befindet sich an erster Position im Array, d. h. nur ein Element muss untersucht werden.

Durchschnittlicher Fall

(Average Case)

- Ist die durchschnittliche Laufzeit eines Algorithmus über alle möglichen gültigen Eingaben mit Größe **n**.
- z. B. Suche: Bei Gleichverteilung ist der gesuchte Wert mit jeweils gleicher Wahrscheinlichkeit an jeder möglichen Stelle im Array. Im Durchschnitt werden ca. **n/2** Elemente untersucht.

Schlechtester Fall

(Worst Case)

- Ist die größtmögliche Laufzeit eines Algorithmus bei einer Eingabe mit Größe **n**.
- z. B. Suche: Gesuchter Wert befindet sich an letzter Position bzw. nicht im Array, d. h. alle **n** Elemente müssen untersucht werden.

Worst-Case

- In der Praxis wird vom Worst-Case ausgegangen
- Vorteile
  - Meist einfacher den Worst-Case zu analysieren
  - Führt zu besseren Algorithmen
    - Algorithmus muss auf allen Eingaben effizient ablaufen

Allgemeinerer Ansatz

- Noch immer Operationen zählen
  - Aber **allgemeiner**
  - Kleine Variationen werden ignoriert
    - Z. B. ob 4, 5 oder 10 Operationen bei einer Iteration benötigt werden
- Abhängigkeit von der Problemgröße
  - Wie verhält sich der Algorithmus, wenn die Größe der Eingabe beliebig groß wird?
- Größe der Eingabe muss genau beschrieben werden
  - Z. B. eine einzige Zahl, Länge eines Arrays

Asymptotische Analyse – Ziele

- Effizienz für große Eingabegrößen evaluieren
- Wachstum der Laufzeit bei steigender Eingabegröße beschreiben
- Möglichst enge obere Schranke für das Wachstum angeben
  - Worst-Case berücksichtigen
- Muss nicht ganz exakt sein
  - Ordnung zählt
  - Nur größte Faktoren bei der Laufzeit berücksichtigen

Asymptotische obere Schranke (O-Notation)

- Einschränkung
  - Wir betrachten Funktionen deren Definitionsbereich und Wertebereich die nicht-negativen ganzen Zahlen sind
- Es gilt für zwei Funktionen $f(n)$ und $g(n)$

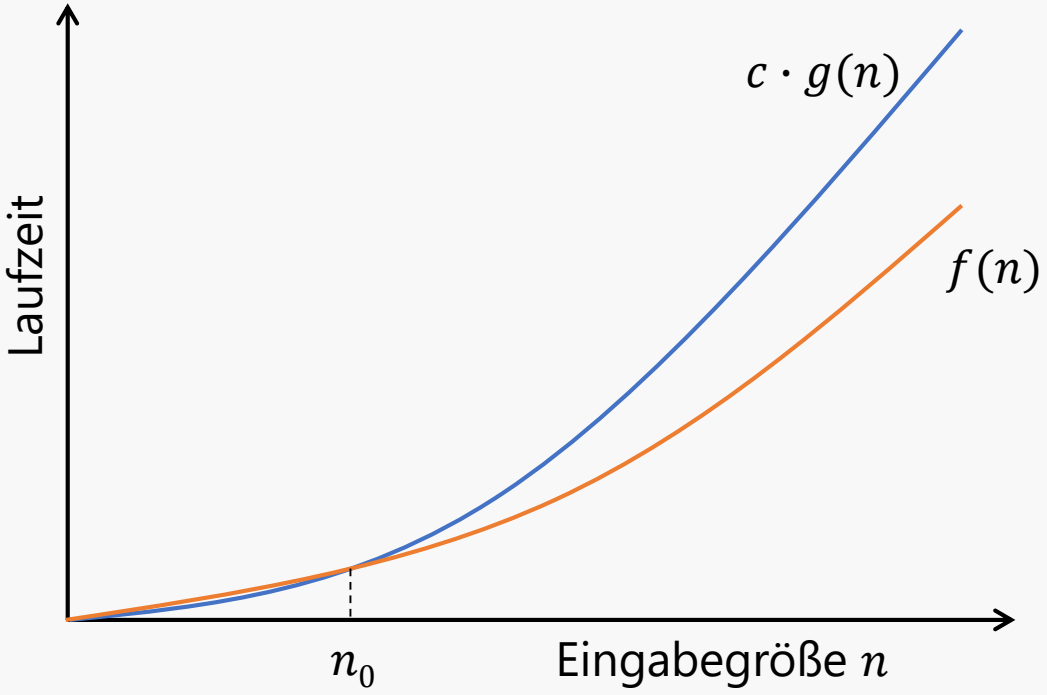
$$f(n) \ ist \ in \ O\big(g(n)\big)$$

- Falls

$$\exists c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 \ f(n) \leq c \cdot g(n)$$

Asymptotische obere Schranke

- Laufzeit $f(n)$ zu Eingabegröße n und Schranke $c \cdot g(n)$ visualisiert



Notation

- Allgemein

$$f(n) \ ist \ in \ O\big(g(n)\big)$$

- Eigentlich

$$f(n) \in O\big(g(n)\big)$$

O(g(n)) bezeichnet eigentlich die Menge aller Funktionen, die asymptotisch durch c•g(n) von oben beschränkt wird.

- Verkürzte (nicht ganz korrekte) Schreibweise

$$f(n) = O\big(g(n)\big)$$

Nicht als Gleichheit zu lesen!

Beispiele

- $8n + 5 \; ist \; in \; O(n)$
  - Wir müssen Konstanten $c > 0$ und $n_0 > 0$ finden, sodass gilt

$$8n + 5 \leq c \cdot n \text{ für jedes } n \geq n_0$$

  - Mögliche Wahl: $c = 9$ und $n_0 = 5$

Es sind auch andere Werte möglich, solange diese eine korrekte obere Schranke ergeben.

n	8n + 5	9n
1	13	9
2	21	18
3	29	27
4	37	36
5	**45**	**45**
6	**53**	**54**
…		

Allgemein

- Falls $f(n)$ ein Polynom vom Grad d ist

$$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$$

- und $a_d > 0$, dann gilt

$$f(n)\ ist\ in\ O(n^d)$$

Allgemein – Erklärung

- Für $n \geq 1$ haben wir $1 \leq n \leq n^2 \leq \cdots \leq n^d$ und daher

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \leq (|a_0| + |a_1| + \cdots + |a_d|) n^d$$

Mit $c = |a_0| + |a_1| + \cdots + |a_d|$ und $n_0 = 1$ gilt $f(n)\ in\ O(n^d)$

Beispiele

- $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$
  - für $c = 15$ und $n_0 = 1$ daher $in\ O(n^4)$
- $5n^2 + 3n \log n + 2n + 5 \leq (5 + 3 + 2 + 5)n^2 = cn^2$
  - für $c = 15$ und $n_0 = 1$ daher $in\ O(n^2)$
  - Hinweis: $\log n \leq n$ für $n \geq 1$
- $20n^3 + 10n \log n + 5 \leq 35n^3$ für $n_0 = 1$ daher $in\ O(n^3)$
- $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$
  - für $c = 4$ und $n_0 = 1$ daher $in\ O(2^n)$

Kürzere Variante

Addition

- Addition

$$O\big(f(n)\big) + O\big(g(n)\big)\ ist\ O\big(f(n) + g(n)\big)$$

  - Bei sequentiell aufeinanderfolgenden Teilen
  - Beispiel

```
for (int i = 0; i < n; i++) {
    System.out.print("a");
}
for (int i = 0; i < n * n; i++) {
    System.out.print("b");
}
```

$$O(n) + O(n * n) = O(n + n^2) = O(n^2)$$

Multiplikation

- Multiplikation

$$O\big(f(n)\big) * O\big(g(n)\big)\ ist\ O\big(f(n) * g(n)\big)$$

- Bei verschachtelten Teilen (Schleifen), die beide von *n* abhängen
- Beispiel

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.print("a");
    }
}
```

$$O(n) * O(n) \ = \ O(n * n) \ = \ O(n^2)$$

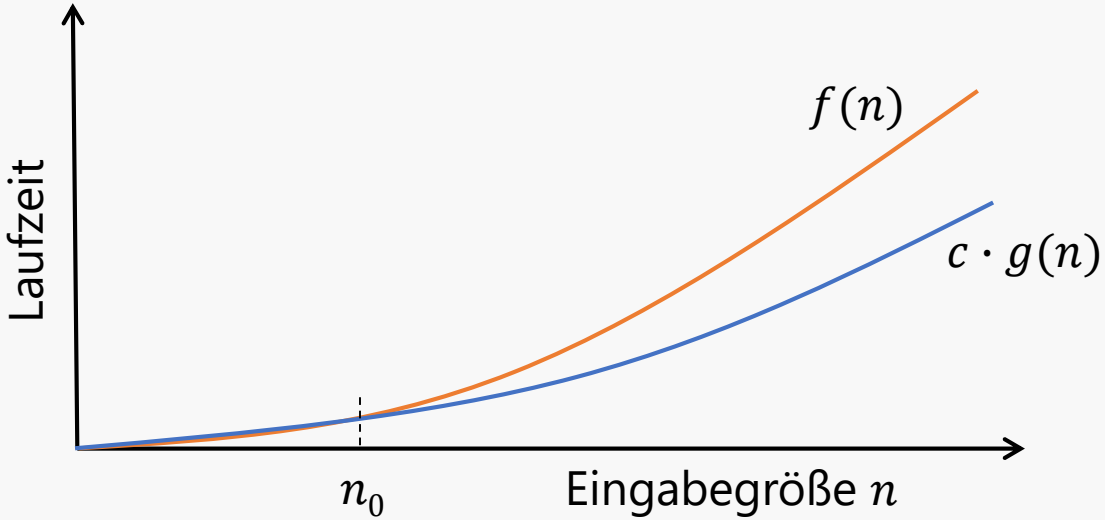
Weitere Schranken – untere Schranke

- Es gilt für zwei Funktionen $f(n)$ und $g(n)$

$$f(n) \ \ ist\ in \ \ \Omega\big(g(n)\big)$$

- Falls

$$\exists c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 \ \ c \cdot g(n) \leq f(n)$$



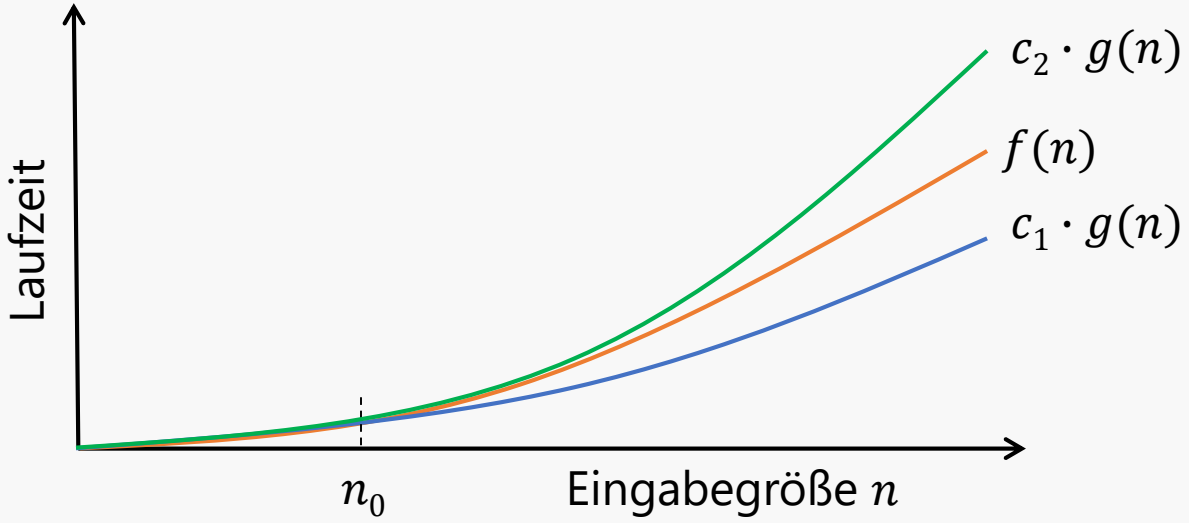
Weitere Schranken – scharfe Schranke

- Es gilt für zwei Funktionen $f(n)$ und $g(n)$

$$f(n) \ \ ist\ in \ \ \Theta\big(g(n)\big)$$

- Falls

$$\exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0\ \forall n \geq n_0\, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



Typische Komplexitätsfunktionen

Beispiele für Komplexitätsfunktionen

	Bezeichnung
$O(1)$	Konstant
$O(\log n)$	Logarithmisch
$O(n)$	Linear
$O(n \log n)$	Log-Linear
$O(n^2)$	Quadratisch
$O(n^3)$	Kubisch
$O(k^n)$	Exponentiell

Konstant – O(1)

- Unabhängig von der Eingabegröße *n* ist der Aufwand für eine Operation konstant
- Konstant auf einem bestimmten System
  - Aber unterschiedlich (konstant) bei verschiedenen Systemen
- Beispiele
  - Bestimmen der Länge eines Arrays
    - Wenn die Länge beim Array gespeichert wird (wie z. B. in Java)

Logarithmisch – O(log n)

- Wenn die Eingabegröße *n* in jedem Schritt um einen bestimmten Faktor (z. B. 2 oder 10) verkleinert wird
- Beispiel (ganze Zahl in String konvertieren)

```
private static String intToString(int n) {
    if (n == 0) {
        return "0";
    }
    String result = "";
    while (n > 0) {
        result = n % 10 + result;
        n = n / 10;
    }
    return result;
}
```

Entscheidender Bereich (abhängig von der Größe von n)
Wie oft kann n durch 10 dividiert werden?
$\log_{10}(n)$ mal (Wert noch aufrunden) und daher $O(\log n)$

Logarithmisch – Hinweis

- Basis ist egal
- Logarithmen zu verschiedenen Basen unterscheiden sich nur um einen konstanten Faktor voneinander
  - Daher kann die Basis weggelassen werden

Linear – O(n)

- Laufzeit wächst linear mit der Eingabegröße
- Beispiel (Summe aller Zahlen eines Arrays bestimmen)

```
private static int addNumbers(int[] input) {
    int sum = 0;
    for (int i = 0; i < input.length; i++) {
        sum += input[i];
    }
    return sum;
}
```

Jedes Element muss betrachtet werden

- Array muss komplett durchlaufen werden
  - Mit wachsender Anzahl der Elemente im Array wächst auch die Laufzeit

Linear – Rekursionsbeispiel

- Beispiel Fakultätsberechnung

```
// n >= 0
private static long factorial(int n) {
    return n <= 0 ? 1 : n * factorial(n - 1);
}
```

- Abfolge von Aufrufen
  - `factorial(n) -> factorial(n-1) -> factorial(n-2) -> … -> factorial(1) -> factorial(0)`
  - Daher linear
- Aber
  - Benötigt auch mehr Speicher – pro Aufruf ein neuer Frame
  - Speicherbedarf wächst linear mit der Eingabegröße

Log-Linear – O(n log n)

- Eine wichtige Klasse mit vielen Algorithmen
- Beispiele
  - Schnelle Sortieralgorithmen
  - Werden noch ausführlich besprochen

Quadratisch – $O(n^2)$

- Beispiel (Array auf Duplikate überprüfen)

```
private static boolean allEntriesDifferent(int data[]) {
    for (int i = 0; i < data.length; i++) {
        for (int j = i + 1; j < data.length; j++) {
            if (data[i] == data[j]) {
                return false;
            }
        }
    }
    return true;
}
```

Äußere Schleife abhängig von der Länge von data

Innere Schleife abhängig von äußerer Schleife. Anzahl der Durchläufe nimmt mit ansteigendem j ab (n – 1, n – 2, …).

$$n - 1 + n - 2 + \cdots + 2 + 1 = \frac{n * (n-1)}{2} = \frac{n^2 - n}{2}$$

$$\frac{n^2 - n}{2} \; ist \; in \; O(n^2)$$

Kubisch – $O(n^3)$

- Z. B. Matrizenmultiplikation

```
private static int[][] multiply(int[][] first, int[][] second) {
    int m = first.length;
    int n = second[0].length;
    int p = second.length;
    int[][] result = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < p; k++) {
                result[i][j] += first[i][k] * second[k][j];
            }
        }
    }
    return result;
}
```

Drei verschachtelte for-Schleifen
Worst-Case – zwei quadratische Matrizen multiplizieren (dann sind m, n und p gleich groß)

Polynomiell allgemein – $O(n^k)$

- Eigentlich
  - Viele bisherige Klassen enthalten (z. B. linear, quadratisch, kubisch)
- Bei größerem k
  - Werden Algorithmen in $O(n^k)$ unbrauchbar bzw. sind nur mehr auf kleinen Eingabegrößen sinnvoll
  - Daher ist eine feinere Unterteilung (linear, quadratisch, kubisch) im unterem Spektrum sinnvoll

Exponentiell – $O(k^n)$

- Beispiel (Türme von Hanoi)

```
private static void solveHanoi(int n, String a, String b, String c) {
    if (n > 0) {
        solveHanoi(n - 1, a, c, b);
        System.out.println(a + " --> " + c);
        solveHanoi(n - 1, b, a, c);
    }
}
```

- $2^n - 1$ Ausgaben bei n Scheiben und daher in $O(2^n)$

Exponentiell – Türme von Hanoi

- Beispiele für $n = 2$, $n = 3$ und $n = 4$

```
Tower 1 --> Tower 2
Tower 1 --> Tower 3
Tower 2 --> Tower 3
```

```
Tower 1 --> Tower 3
Tower 1 --> Tower 2
Tower 3 --> Tower 2
Tower 1 --> Tower 3
Tower 2 --> Tower 1
Tower 2 --> Tower 3
Tower 1 --> Tower 3
```

```
Tower 1 --> Tower 2
Tower 1 --> Tower 3
Tower 2 --> Tower 3
Tower 1 --> Tower 2
Tower 3 --> Tower 1
Tower 3 --> Tower 2
Tower 1 --> Tower 2
Tower 1 --> Tower 3
Tower 2 --> Tower 3
Tower 2 --> Tower 1
Tower 3 --> Tower 1
Tower 2 --> Tower 3
Tower 1 --> Tower 2
Tower 1 --> Tower 3
Tower 2 --> Tower 3
```

- Achtung
  - Z. B. 1023 Ausgaben bei $n = 10$
  - Z. B. 1.048.575 Ausgaben bei $n = 20$
  - Wächst sehr schnell an!

Hinweise zur O-Notation

- Beschreibt asymptotisches Verhalten
  - Für kleine Eingabegrößen kaum aussagekräftig
- Kleinste obere Schranke wählen
  - Matrizenmultiplikation (eigentlich $O(n^3)$) ist z. B. auch in $O(n^{10})$
    - Das hilft aber nicht viel bei der Auswahl von Algorithmen
- Für große Eingabegrößen
  - Algorithmus mit kleinerer oberer Schranke wählen
    - Z. B. $O(n \log n)$ gegenüber $O(n^2)$ bevorzugen

Maximale Abschnittssumme – Wiederholung (1)

- Erste Variante ist ein kubischer Algorithmus

```
private static int maxSum1(int[] input) {
    int maxSum = input[0];
    for (int left = 0; left < input.length; left++) {
        for (int right = left; right < input.length; right++) {
            int sum = 0;
            for (int k = left; k <= right; k++) {
                sum += input[k];
            }
            if (sum > maxSum) {
                maxSum = sum;
            }
        }
    }
    return maxSum;
}
```

Drei verschachtelte Schleifen!
Alle Obergrenzen von der Länge von input abhängig – $O(n^3)$

Maximale Abschnittssumme – Wiederholung (2)

- Zweite Variante ist ein quadratischer Algorithmus

```
private static int maxSum2(int[] input) {
    int maxSum = input[0];
    for (int left = 0; left < input.length; left++) {
        int sum = 0;
        for (int right = left; right < input.length; right++) {
            sum += input[right];
            if (sum > maxSum) {
                maxSum = sum;
            }
        }
    }
    return maxSum;
}
```

Zwei verschachtelte Schleifen!
Alle Obergrenzen von der Länge von input abhängig – $O(n^2)$

Maximale Abschnittssumme – Wiederholung (3)

- Dritte Variante ist ein linearer Algorithmus

```
private static int maxSum3(int[] input) {
    int maxSum = input[0], endSum = maxSum;
    for (int pos = 1; pos < input.length; pos++) {
        System.out.println("pos = " + pos);
        endSum = Math.max(endSum + input[pos], input[pos]);
        maxSum = Math.max(maxSum, endSum);
    }
    return maxSum;
}
```

Eine Schleifen über alle Elemente des Arrays – $O(n)$
Hinweis: Die Laufzeit von `Math.max` ist in $O(1)$

Kann das noch verbessert werden?

```
private static int maxSum3(int[] input) {
    int maxSum = input[0], endSum = maxSum;
    for (int i = 1; i < input.length; i++) {
        endSum = Math.max(endSum + input[i], input[i]);
        maxSum = Math.max(maxSum, endSum);
    }
    return maxSum;
}

private static int maxSum4(int[] input) {
    int maxSum = input[0], endSum = maxSum;
    for (int i = 1; i < input.length; i++) {
        int help = endSum + input[i];
        endSum = help > input[i] ? help : input[i];
        if (endSum > maxSum) {
            maxSum = endSum;
        }
    }
    return maxSum;
}
```

Variante ohne Methodenaufrufe

Ergebnisse

Eingabegröße n	maxSum3 (sec)	maxSum4 (sec)	Faktor
20 000 000	0.023	0,014	1.64
40 000 000	0.036	0,021	1,71
80 000 000	0.070	0,042	1.66
160 000 000	0.139	0,076	1.82
320 000 000	0.267	0,155	1.72
640 000 000	0.529	0,296	1.78

- Laufzeit **linear** für beide Implementierungen
  - `maxSum3` braucht länger als `maxSum4`
  - Faktor (Zeit für `maxSum3` / Zeit für `maxSum4`) schwankt aber nur um einen bestimmten Wert

Zusammenfassung

- Algorithmen allgemein
- Einführendes Beispiel
- Komplexitätsanalyse von Algorithmen
- Typische Komplexitätsfunktionen