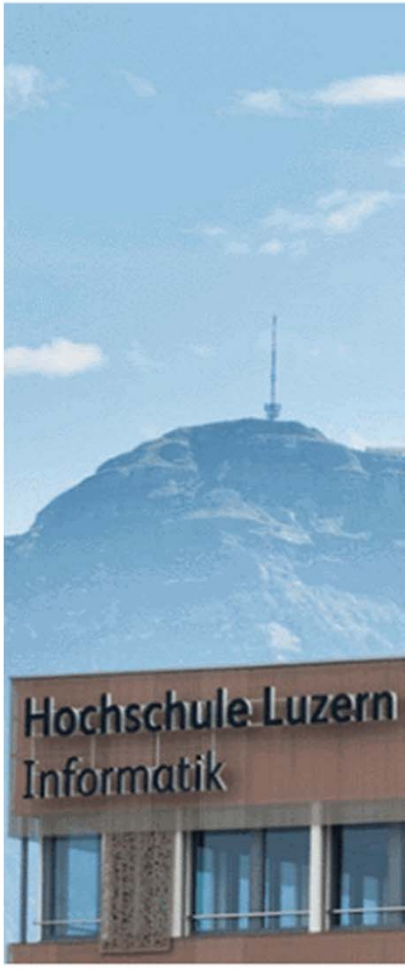
Algorithmen & Datenstrukturen

Einstieg

Algorithmen, Datenstrukturen und Komplexität

Hansjörg Diethelm

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
INFORMATIK



Inhalt

- Algorithmen und Datenstrukturen
- Komplexität eines Algorithmus

Lernziele

Sie ...

- können beschreiben, was ein Algorithmus ist.
- können erläutern, was gleichwertige Algorithmen bedeuten.
- können erläutern, weshalb Algorithmen und Datenstrukturen eng zusammenhängen.
- können beschreiben, was Komplexität bei einem Algorithmus meint.
- können für einfache Funktionen deren Ordnung bestimmen.
- dito für einfache Code-Fragmente betreffend Zeitkomplexität.
- kennen die wichtigsten Ordnungsfunktionen im Vergleich.
- kennen wichtige Aspekte bei der Interpretation einer Ordnung.
- wissen, welche Ordnungen praktisch versagen!

Algorithmen und Datenstrukturen

Definition Algorithmus

- Ein Algorithmus ist ein **präzise festgelegtes Verfahren zur Lösung eines Problems**; genauer gesagt von einer Problem-Klasse beinhaltend (unendlich) viele gleichartige Probleme.

- Etwas salopp:
  Algorithmus = Lösungsverfahren (Rezept, Anleitung)

- Probleme bzw. Problem-Klassen , die mit Algorithmen gelöst werden können, heissen ➔ **berechenbar**.

Beispiele

- Berechnung des grössten gemeinsamen Teilers (**ggT**) für zwei beliebige natürliche Zahlen.
- Zeichnen der **Verbindungslinie**, welche zwei Punkte verbindet.
- **Sortierung** von zufällig vorliegenden ganze Zahlen.
- Finden des **kürzesten Weges** zwischen zwei Knoten in einem zusammenhängenden Graphen.
- Entscheiden, ob es sich bei einer vorliegenden natürlichen Zahl um eine **Primzahl** handelt.
- Berechnung des **Integrals** bei vorliegenden Funktionswerten in einem bestimmten Bereich.
- Finden einer **Lösung** in einem vorgegebenen Lösungsraum.

Eigenschaften eine Algorithmus

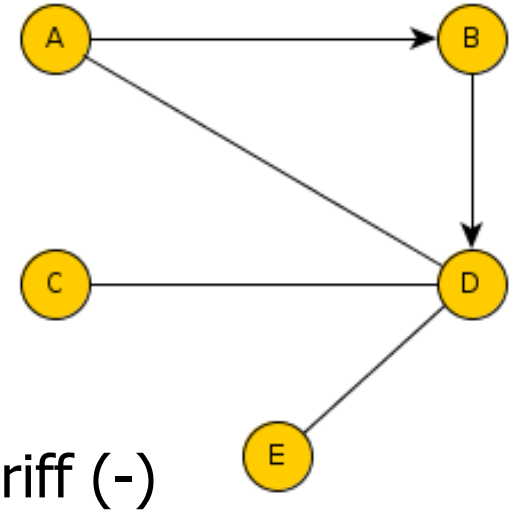
- schrittweises Verfahren
- ausführbare Schritte
- eindeutiger nächster Schritt (➔ **determiniert**)
- endet nach endlich vielen Schritten (➔ **terminiert**)

Algorithmen vs. Informatik

- Der Computer ist **DER** «Algorithmen-Cruncher», vgl.
  - arbeitet schrittweise, gemäss Programm
  - Anweisung für Anweisung, jede Anweisung korrespondiert mit einem ausführbaren Befehl
  - arbeitet präzise und schnell
- Algorithmen sind zentrales Thema in der Informatik und in der Mathematik, vgl.
  - **Algorithmentheorie:** Guter Lösungsalgorithmus für bestimmte Problemstellung?
  - **Komplexitätstheorie:** Ressourcenverbrauch von Rechenzeit und Speicherbedarf?
  - **Berechenbarkeitstheorie:** Was ist mit einer Maschine grundsätzlich lösbar bzw. nicht lösbar?

Definition Datenstruktur

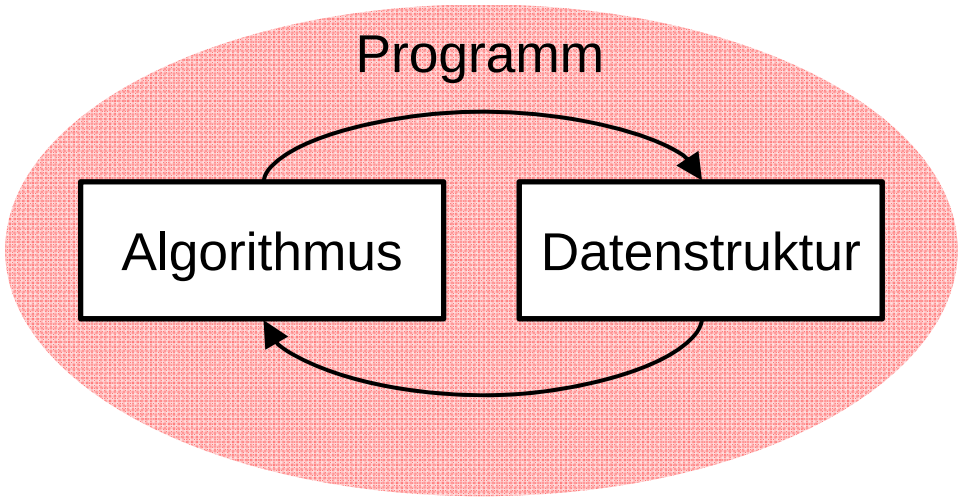
- Eine Datenstruktur ist ein **Konzept zur Speicherung und Organisation von Daten**. Es handelt sich um eine ➔ **Struktur**, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung möglichst effizient zu ermöglichen.
  Datenstrukturen sind daher insbesondere auch durch die ➔ **Operationen** charakterisiert, welche Zugriff und Verwaltung realisieren.

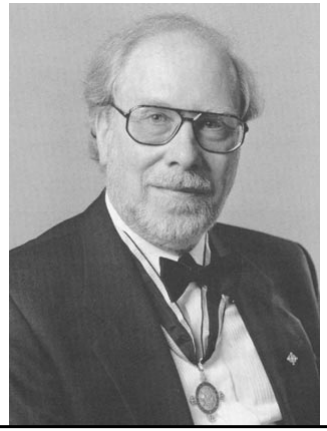


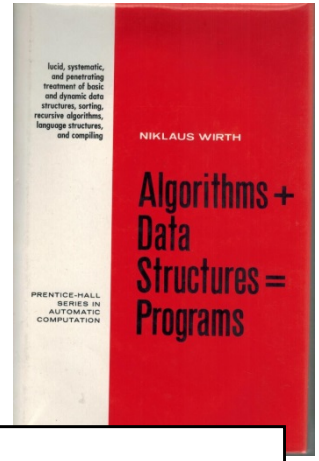
- Beispiele:
  - **Array**: direkter Zugriff (+), fixe Grösse (-)
  - **Liste**: flexible Grösse (+), sequentieller Zugriff (-)

Algorithms + Data Structures = Programs

- Bei vielen Algorithmen hängt der ➔ **Ressourcenbedarf**, d.h. sowohl die benötigte Laufzeit als auch der Speicherbedarf, von der Verwendung geeigneter Datenstrukturen ab.
- **Algorithmen operieren auf Datenstrukturen** und **Datenstrukturen bedingen spezifische Algorithmen**. Beides ist untrennbar miteinander verbunden!







Niklaus Wirth
Algorithms + Data Structures = Programs
Prentice Hall, 1976

Schweizer Informatiker
Turing Award Preisträger 1984

Fragen, die sich stellen ...

- betreffend Algorithmen und Datenstrukturen:
  - Adäquat für kleine Probleme? Adäquat für grosse Probleme?
  - Selber entwickeln oder aus Bibliothek wählen?
  - Einfach zu verstehen, zu implementieren, zu warten oder schwierig?
  - Schneller Algorithmus mit grossem Speicherbedarf oder umgekehrt?
- Extrembeispiel: Ist n eine Primzahl, ja/nein?
  - Prüfen, ob n durch 2, 3, 4, 5, 6, … , sqrt(n) ganzzahlig teilbar.
    → wenig Speicher, lange Rechenzeit
  - Alle Primzahlen 2, 3, 5, 7, 11, … im Voraus in einer Tabelle abspeichern. → viel Speicher, schnell

    Natürlich kann man dies praktisch nur für eine endliche Anzahl Primzahlen tun.

Euklidische Algorithmus zur ggT-Berechnung (300 v. Chr.)

- Ermittlung des grössten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen A und B:

1. Sei A die grössere der beiden Zahlen A und B. (Vertauschen, falls dies nicht so ist.)
2. Setze A = A – B
3. Wenn A und B ungleich sind, dann fahre fort mit Schritt 1, wenn sie gleich sind, dann beende den Algorithmus. Diese Zahl ist der ggT.

Euklidischer Algorithmus «von Hand» ausgeführt

- ggT von 14 und 8:

Schritt	A	B	A – B
1.	14	8	6
2.	8	6	2
3.	6	2	4
4.	4	2	2
5.	2	2	gleich → 2 ist ggT

Iterative Implementation (1)

```
public static int ggtIterativ1(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Iterative Implementation (2)

```
public static int ggtIterativ2(int a, int b) {
    while ((a != 0) && (b != 0)) {
        if (a > b) {
            a = a % b;
        } else {
            b = b % a;
        }
    }
    return (a + b); // Eine Zahl ist 0!
}
```

Rekursive Implementation

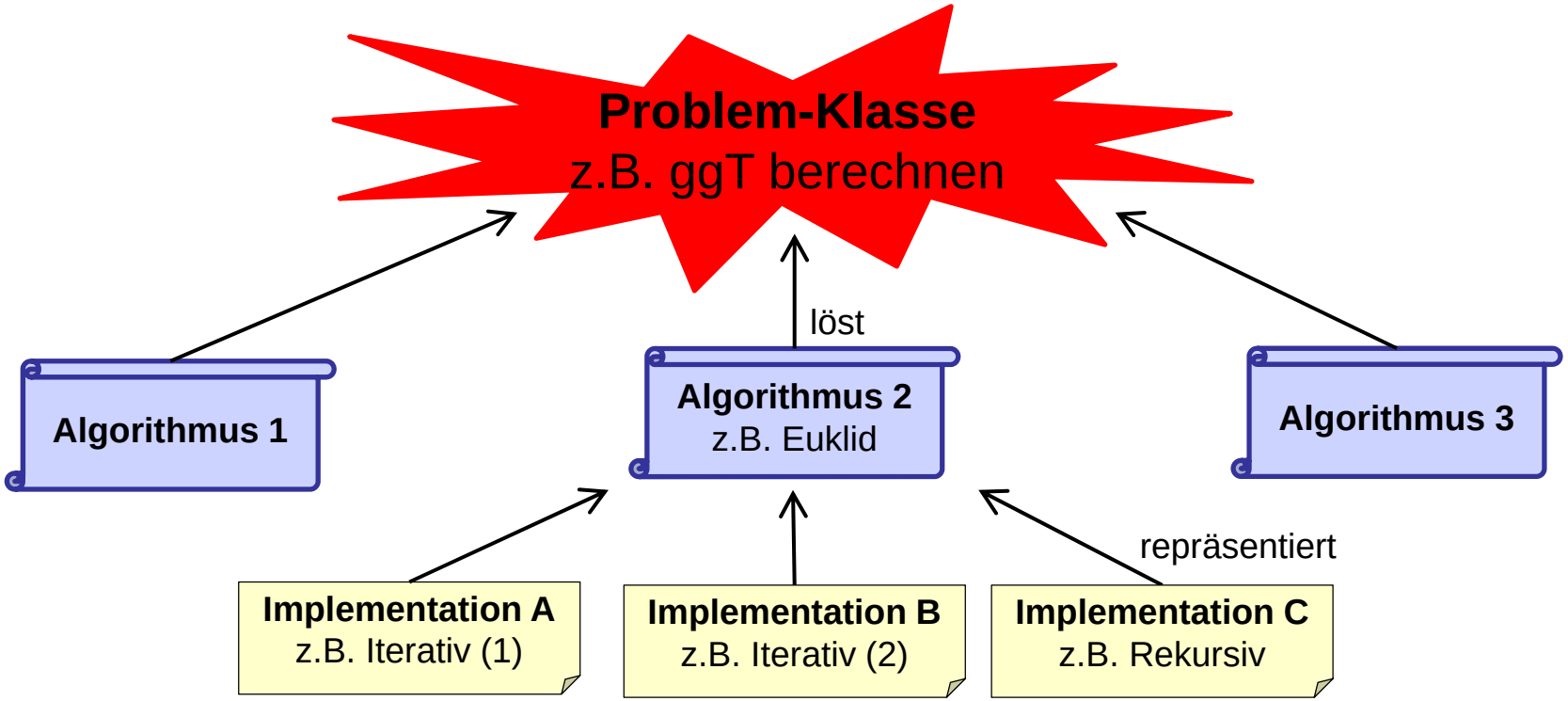
```
public static int ggtRekursiv(final int a, final int b) {
    if (a > b) {
        return ggtRekursiv(a - b, b);
    } else {
        if (a < b) {
            return ggtRekursiv(a, b - a);
        } else {
            return a;
        }
    }
}
```

Fazit der unterschiedlichen Implementierungen

- Alle Implementierungen basieren auf Euklid.
- unterschiedliche Ansätze: Schleifen vs. Rekursion
- «Abkürzungen»: Modulo-Operation anstelle x-mal A-B
- unterschiedliche Anzahl Schleifendurchläufe
  → Ausführungsgeschwindigkeit
- unterschiedliche Anzahl Methodenaufrufe
  → unterschiedlicher Speicherbedarf

- Alle Implementierungen liefern die gleichen Resultate (gemäss Testing).
- Alle Implementierungen sind → **gleichwertig**.

Gleichwertige Algorithmen



- Die Gleichwertigkeit von Algorithmen allgemein zu beweisen ist ein ➔ **unlösbares Problem**. Dies bedeutet, dass es kein Java-Programm geben kann, das den Quellcode von zwei Implementationen einliest und feststellt, ob sie gleichwertige Algorithmen repräsentieren oder nicht!

Komplexität eines Algorithmus

Definition Komplexität

- Komplexität (auch Aufwand oder Kosten) eines Algorithmus:

Ressourcenbedarf = f (Eingabedaten)

D.h. «Wie hängt der Ressourcenbedarf von den Eingabedaten ab?»

- Ressourcenbedarf:
  - Rechenzeit ➔ **Zeitkomplexität**
  - Speicherbedarf ➔ **Speicherkomplexität**
- Eingabedaten:
  - Grösse der Daten**menge** (z.B. 100 vs. 1'000'000'000 zu sortierende Elemente)
  - Grösse eines Daten**wertes** (z.B. 10! vs. 1'000'000'000!)

Was interessiert uns?

- **Wie wächst der Ressourcenbedarf**, wenn mehr Daten zu verarbeiten sind? Z.B.
  - Bleibt der Ressourcenbedarf gleich, wenn wir den ggT von zwei sehr grossen Zahlenwerten berechnen wollen?
  - Verdoppelt oder vervierfacht sich der Ressourcenbedarf für das Sortieren der doppelten Datenmenge?
- Es interessiert an dieser Stelle **NICHT** der exakte/absolute Ressourcenbedarf !
  - Z.B., die ggT-Berechnung von 1'000'000'489 und 9'123'000'124 auf dem Computer XY mit der Konfiguration Z dauert 0.42 Sek.
  - Vgl., solche Rechenzeiten sind für jeden Computer anders. Möchte man die Rechenzeit reduzieren, so lässt sich jederzeit ein schnellerer Computer kaufen!

«Aha-Beispiel» zur Zeitkomplexität: Rechenzeit T = f(n) ?

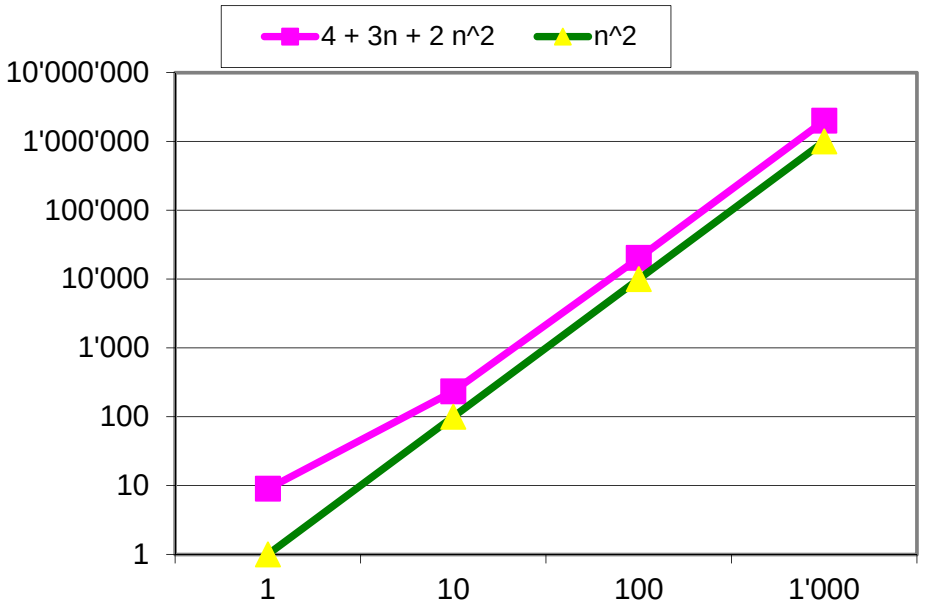
```
public static void task(final int n) {
    task1(); task1(); task1(); task1(); // T ~ 4
    for (int i = 0; i < n; i++) {       // äussere Schleife: n-mal
        task2(); task2(); task2();      // T ~ n· 3
        for (int j = 0; j < n; j++) {   // innerer Schleife: n-mal
            task3(); task3();           // T ~ n· n· 2
        }
    }
}
```

- Annahmen:
  - Die Methoden `task1()`, `task2()` und `task3()` besitzen in etwa dieselbe Rechenzeit.
  - Die Schleifensteuerungen beanspruchen im Vergleich vernachlässigbare kleine Ausführungszeiten.

→ Rechenzeit T von `task(n)`: $T = f(n) \sim 4 + 3 \cdot n + 2 \cdot n^2$

Zeitkomplexität für grosse n

n	$4 + 3 \cdot n + 2 \cdot n^2$	n^2
1	9	1
10	234	100
100	20‘304	10‘000
1‘000	2‘003‘004	1‘000‘000
10‘000	200‘030‘004	100‘000‘000



- Für grosse n dominiert der Anteil von **n^2**.
- Für grosse n verlaufen die Funktionen parallel, d.h. unterschieden sich nur durch einen konstanten Faktor (vgl. logarith. Massstäbe!).

Wir sagen:

- f(n) ist von der ➔ **Ordnung $O(n^2)$** bzw. die
- Rechenzeit von `task(n)` verhält sich gemäss **Ordnung $O(n^2)$**.

Interpretation von $O(n^2)$

Die Rechenzeit von `task(n)` verhält sich gemäss **Ordnung $O(n^2)$** bedeutet:

- **Verdoppelung** von n
  → 2^2 = 4-fache Rechenzeit $\quad \frac{(\mathbf{2}\cdot n)^2}{(n)^2} = 4$
- **Verdreifachung** von n
  → 3^2 = 9-fache Rechenzeit $\quad \frac{(\mathbf{3}\cdot n)^2}{(n)^2} = 9$
- **Verzehnfachung** von n
  → 10^2 = 100-fache Rechenzeit $\quad \frac{(\mathbf{10}\cdot n)^2}{(n)^2} = 100$

Definition «Big-O-Notation»

- Die ➔ **Big-O-Notation** (landau'sches Symbol) bringt zum Ausdruck, dass eine Funktion f(n) höchstens so schnell wächst wie eine andere Funktion g(n). **g(n) ist die obere Schranke** für f(n).
- Die Funktion f(n) ist in der Menge O(g(n)), falls es Konstanten $\mathbf{c} \in R$, $\mathbf{n_0} \in N$ gibt, sodass für alle $n \in N$, $n > \mathbf{n_0}$ gilt:

$$f(n) \leq \mathbf{c} \cdot g(n)$$

- Anders ausgedrückt:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq \mathbf{c}$$

- Vorheriges Beispiel: `task(n)` mit $T = f(n) = K \cdot (4 + 3 \cdot n + 2 \cdot n^2)$
  → $g(n) = n^2$, d.h. $f(n) \in O(n^2)$ bzw. « f(n) hat die Ordnung $O(n^2)$»

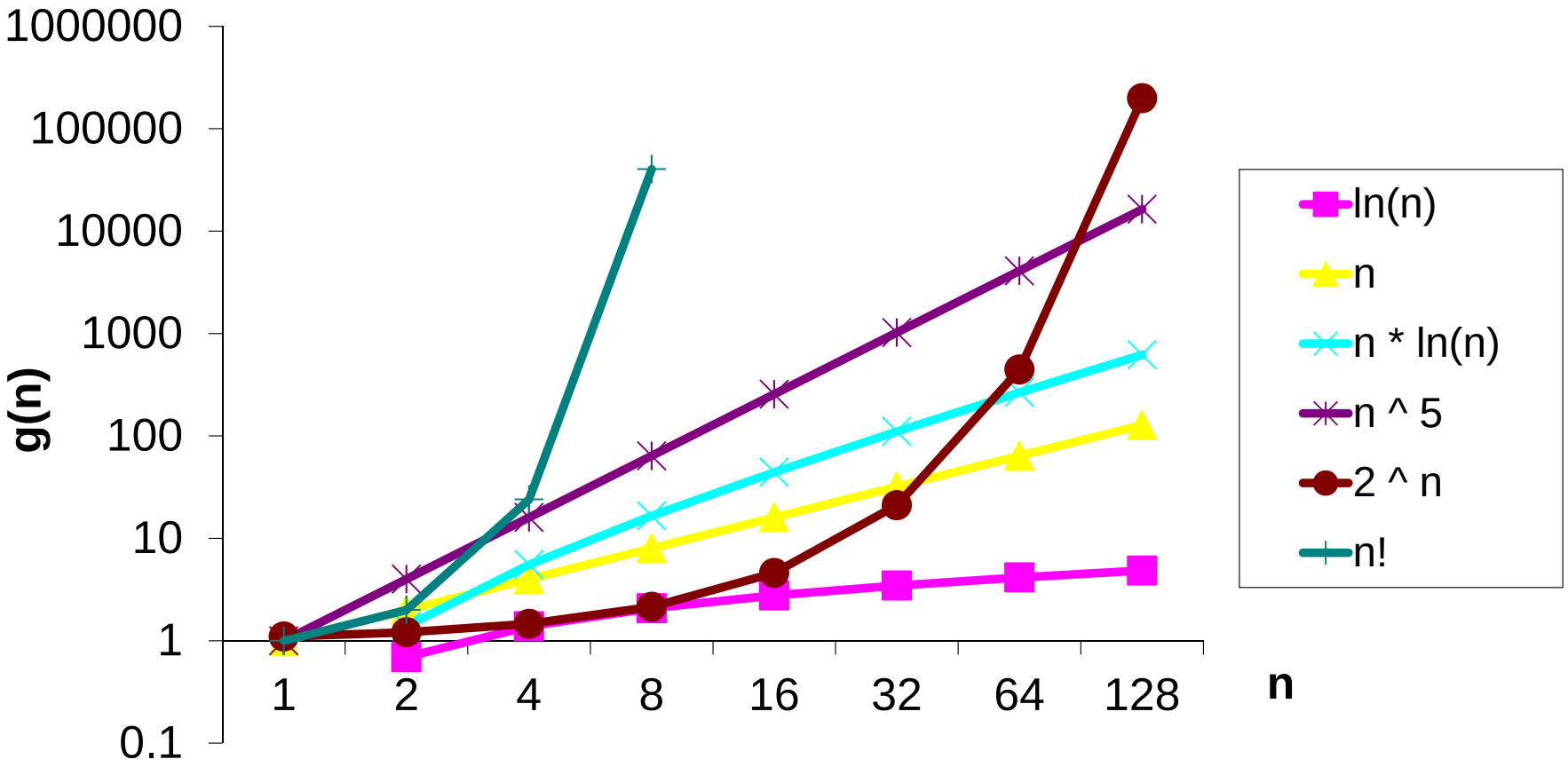
Wichtige Ordnungsfunktionen

Aufgelistet gemäss Wachstumsrate:		Beispiele:
▪ Konstant	O(1)	Hashing
▪ Logarithmisch	O(ln(**n**))	binäres Suchen
▪ Linear	O(**n**)	Suchen in Texten
▪ n log n	O(**n** · ln(**n**))	schlaues Sortieren
▪ Polynomial ①	O($\mathbf{n}^m$)	einfaches Sortieren
▪ Exponential	O($d^{\mathbf{n}}$)	Optimierungen
▪ Fakultät	O (**n** !)	Permutationen, TSP ②

① In der Praxis anwendbare Algorithmen haben typisch Ordnungsfunktionen $\leq \mathbf{n^2}$ oder maximal $\mathbf{n^3}$.

② TSP = «Travelling Salesman Problem»

Funktionsverläufe zur Veranschaulichung



Rechnen / Handhabung der Big-O-Notation

- $f(n) = 0.001 \cdot n^3$ → $O(n^3)$
  D.h. konstante Faktoren ignorieren bzw. als 1 setzen.

- $f(n) = ld_2(10 \cdot n)$ → $O(ld_2(n)) = O(\log_{10}(n)) = O(ln_e(n))$
  D.h. bei der Ordnung spielt die Logarithmus-Basis keine Rolle.

- $f(n) = 2 + 37 \cdot n^3 + 0.01 \cdot n^4 + 0.1 \cdot 2^n$ → $O(2^n)$
  D.h. nur der am stärksten wachsende Summand zählt.

Abschliessende Bemerkungen

- Die Ordnung ignoriert **konstante Faktoren**. Diese können aber praktisch relevant sein!
- Die Ordnung berücksichtigt das **Verhalten bei kleinen n** nicht. Langsamere Algorithmen sind aber bei kleinen n oft auch brauchbar oder gar die bessere Wahl!
- Die **exakte mathematische Analyse** von vielen Algorithmen ist schwierig oder sogar unmöglich!
- Häufig muss man bei der Analyse **differenzieren**:
  - bester Fall (best Case)
  - schlimmster Fall (worst Case)
  - mittlerer Fall (average Case)

  Häufig ist der mittlere Fall ein besseres Mass für die Leistung eines Algorithmus, als der schlimmste.

Zusammenfassung

- Ein Algorithmus ist ein Lösungsverfahren für eine Problemklasse.
- Algorithmen und Datenstrukturen sind untrennbar miteinander verbunden.
- Für eine Problemklasse gibt es typisch verschiedene Lösungs-Algorithmen, die ihrerseits in Variationen implementiert werden können.
- Unter der Komplexität eines Algorithmus versteht man sein Ressourcenbedarf in Abhängigkeit der Eingabedaten.
- Bei den Ressourcen geht es um Rechenzeit und Speicherbedarf.
- Es interessiert insbesondere, wie sich der Ressourcenbedarf bei einem bestimmten Algorithmus bei grossen Problemen entwickelt.
- Die Big-O-Notation ermöglicht vielfach eine adäquate Qualifizierung.

Fragen?