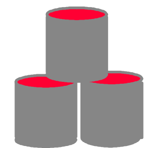
Grundlagen der Informatik
– Algorithmen und Komplexität –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



Inhalt

- Einleitung
- Problemreduktion durch Rekursion
- Komplexität

Einleitung

- Der Entwurf von Algorithmen und damit von Programmen ist eine konstruktive und kreative Tätigkeit.
  - ⇨ Neben der reinen Funktionalität sind auch Fragen der Performance wie Laufzeit und benötigter Speicher zu berücksichtigen.
- Eine automatische Ableitung eines optimalen Algorithmus aus einer Beschreibung der Anforderung ist prinzipiell nicht automatisierbar.
  - ⇨ Daher wird in der Praxis i.d.R. an Hand von Heuristiken eine geeignete algorithmische Lösung konstruiert.
  - ⇨ Solche Heuristiken können als „*best practice*" Beispiele interpretiert werden
    - Sie helfen Entwicklern, brauchbare Erfahrungen für die Lösung neuer Probleme zu nutzen.
  - ⇨ Neben allgemeinen Prinzipien wie etwa das der schrittweisen Verfeinerung als Entwurfsmethode oder Rekursion / Iteration als Lösungsstrategie zur Problemreduktion unterscheidet man spezielle Algorithmenmuster.

Schrittweise Verfeinerung

- In Kapitel „Prozedurale Programmierung" wurde bereits der Top-Down Entwurf als Spezialfall der schrittweisen Verfeinerung kennen gelernt im Zusammenhang mit einem ersten intuitiven Algorithmusbegriff.
- Verfeinerung basiert auf dem Ersetzen von Pseudocodeteilen durch verfeinerten Pseudocode und letztendlich durch konkrete Algorithmenschritte bzw. Progammcode (z.B. in Java):
  - ⇨ Zerlege die Aufgabe in einfache Teilaufgaben und
  - ⇨ betrachte jede Teilaufgabe unabhängig von den anderen für sich allein.
  - ⇨ Falls eine Teilaufgabe zu komplex ist, um eine Lösung zu finden, wird die schrittweise Verfeinerung auf die Teilaufgabe angewendet.
  - ⇨ Nach erfolgreicher Zerlegung gibt es nur noch so einfache Teilaufgaben, dass sie mit Hilfe von elementaren Programmcode ausgedrückt werden können.
  - ⇨ Die Gesamtheit der Teil-Lösungen zusammen mit einer Vorschrift über ihr Zusammenwirken bilden die Lösung des Gesamtproblems.

Problemreduktion durch Rekursion

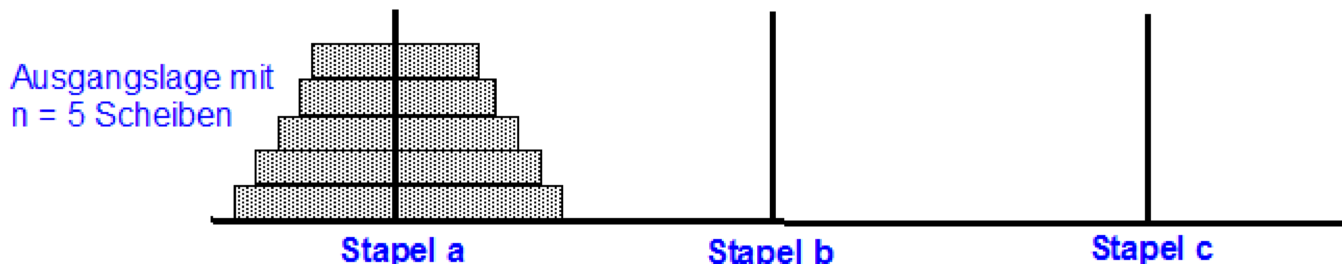
- Rekursive Algorithmen wie in der Informatik kommen in den klassischen Ingenieurwissenschaften nicht vor:
  - ⇨ Ein mechanisches Bauteil kann physikalisch nicht sich selbst als Bauteil enthalten,
  - ⇨ während ein Programm sich durchaus selbst aufrufen kann
- Rekursive Programmierung ist daher
  - ⇨ i.d.R. nicht aus dem Alltagswissen ableitbar, sondern muss als Technik erlernt und geübt werden.
  - ⇨ besonders für mathematische Probleme geeignet, die sich sehr elegant, exakt und kompakt mit Hilfe der Rekursion formulieren und lösen lassen.

Problemreduktion durch Rekursion

- Der folgende Sachverhalt kann in der theoretischen Informatik nachgewiesen werden:
  - ⇨ Jede berechenbare Funktion kann mit Hilfe der Rekursion dargestellt werden
    - also ohne die Verwendung von lokalen Variablen
- Es gilt also: Die Mächtigkeit der Rekursion entspricht der Mächtigkeit der imperativen Programmierung.
  - ⇨ Zur Veranschaulichung kann aus Kapitel „Prozedurale Programmierung“ das Fakultäts-Beispiel herangezogen werden.

Rekursion Beispiel: Türme von Hanoi

- Ein geordneter Stapel von n der Größe nach sortierten Scheiben soll verschoben werden. Dazu
  - ⇨ darf immer nur eine Scheibe in einem Schritt bewegt werden und
  - ⇨ nie eine größere auf eine kleinere Scheibe abgelegt werden.
- Löse dieses Problem mit insgesamt drei Ablageplätze a, b und c,
  - ⇨ wobei der Stapel zu Beginn bei a steht und nach b verschoben werden soll.
  - ⇨ Es darf also ein Hilfsstapel (c) verwendet werden.



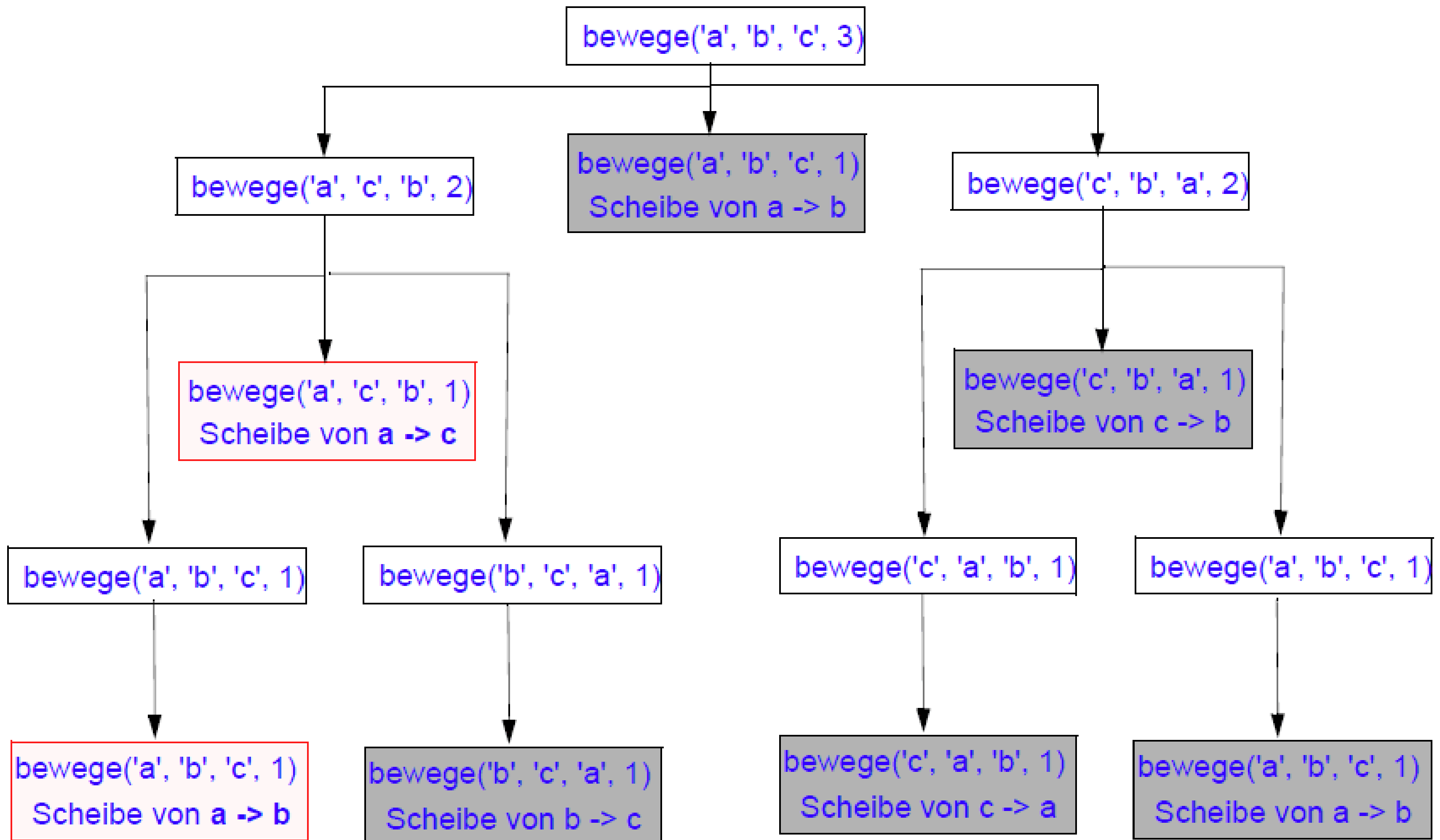
Lösungsidee zum Beispiel: Türme von Hanoi

- Für Stapel mit 2 Scheiben ist das Problem trivial
  - ⇨ Die kleine Scheibe kommt auf den Hilfsstapel, die größere auf das Ziel
  - ⇨ Stapel mit 1 Scheibe ist natürlich auch trivial
- Wie sieht es mit 3 Scheiben aus?
  - ⇨ 2 Scheiben müssen auf den Hilfsstapel, die letzte auf das Ziel
  - ⇨ Das Problem kann also auf die Lösung mit 2 Scheiben zurückgeführt werden!

Implementierungsansatz Türme von Hanoi

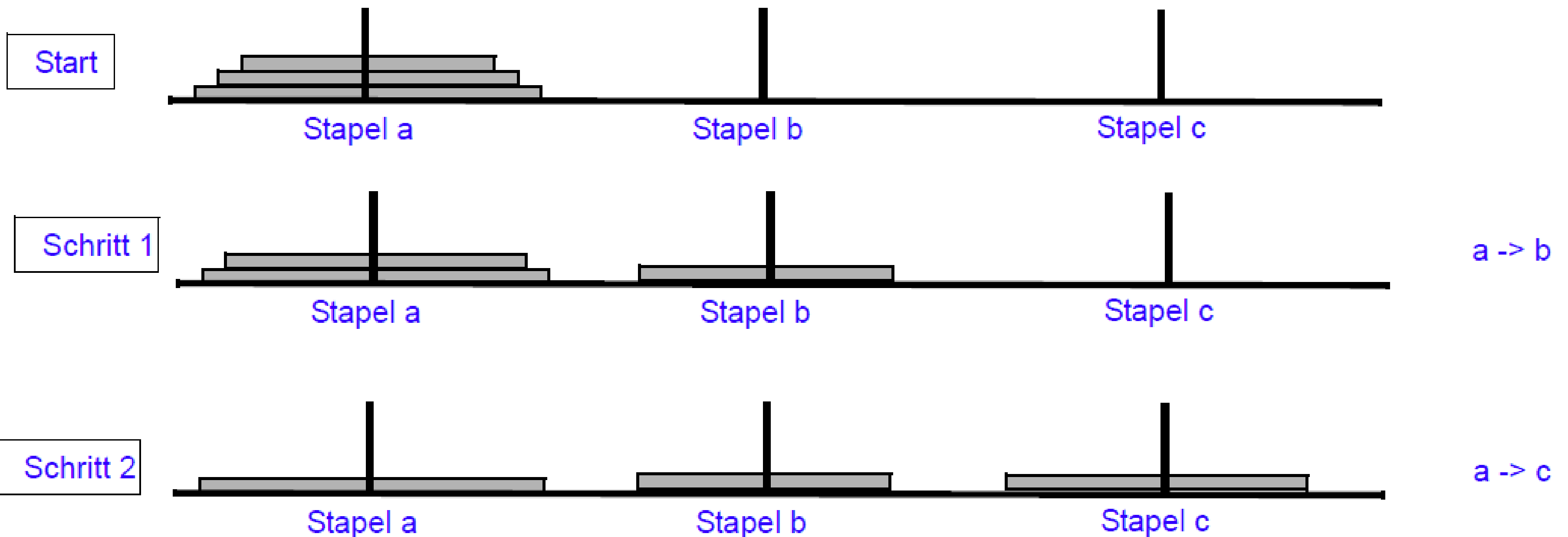
```java
class TuermeVonHanoi {
   // Bewegt n Scheiben von Turm a nach Turm b und
   // benutzt als Zwischenspeicher Turm z.
   static void bewege (char a, char b, char z, int n) {

      if (n == 1) {
         System.out.println("Bewege Scheibe von "+ a +" auf "+ b);
      } else {
         bewege (a, z, b, n-1);   // die oberen n-1 Scheiben von a nach c
         bewege (a, b, z, 1);     // Bewege größte Scheibe von a nach b
         bewege (z, b, a, n-1);   // die oberen n-1 Scheiben von c nach b
      }
   } // bewege

   public static void main (String[] args) {
      // Gib die notwendigen Züge für einen Stapel der Höhe 5 aus
      bewege('a', 'b', 'z', 5);
   } // main
} // TuermeVonHanoi
```

Was passiert beim Aufruf von bewege ('a', 'b', 'c', 3)?

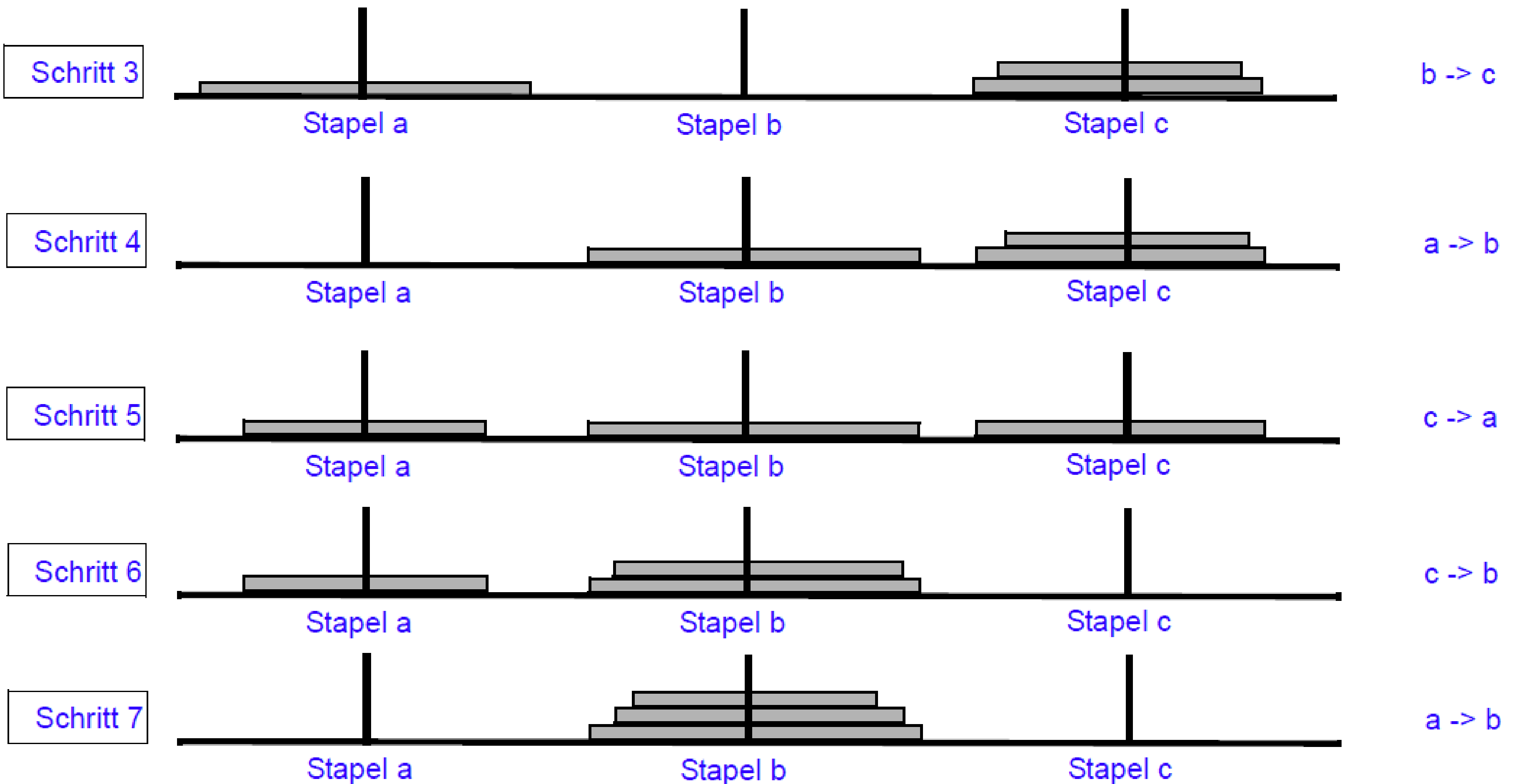


Was passiert beim Aufruf von bewege ('a', 'b', 'c', 3)?

- Der Ablauf kann entweder durch die print-Befehle im Programm veranschaulicht werden oder durch „Nachdenken“:



Was passiert beim Aufruf von bewege ('a', 'b', 'c', 3)?



Rekursion vs. Iteration

- Warum sollte eine iterative Lösung bevorzugt werden?
  - ⇨ Sie beansprucht deutlich weniger Speicherplatz als die rekursive Variante
    - die Variablen müssen nicht für jeden Rekursionsschritt angelegt werden
  - ⇨ Die rekursive Lösung erfordert sehr viele Unterprogramm-Aufrufe
    - großen Aktivität im Programm-Stack (viele Kontextwechsel)
- Wenn es geht, sollte daher ein Problem iterativ (induktiv) gelöst werden
  - ⇨ nur unter Verwendung der gängigen Schleifenkonstrukte!
- In zeitkritischen Anwendungen sollte Rekursion vermieden werden.
  - ⇨ eine optimale rekursive Lösung kann auch in einem anschließenden Optimierungsschritt in eine iterative umgewandelt werden.
- Wie könnte eine iterative Lösung der Fakultäts-Funktion in Java aussehen?

Elimination von Rekursion

- Mittels Rekursion lassen sich Spezifikationen recht elegant und einfach implementieren.
- Die so erhaltenen Lösungen sind jedoch meist nicht sehr effizient, was den Speicherverbrauch und die Laufzeit betrifft.
- Jede primitive Rekursion lässt sich durch eine iterative Lösung mittels Schleifen darstellen
  - ⇨ Gesucht ist ein strukturierter Weg, um eine rekursive Lösung in eine Iterative zu überführen.
- Ein erster Schritt hierzu ist die ***Endrekursion***, die eine effizientere Implementierung gestattet.

Endrekursion

- Ein Algorithmus f ist endrekursiv,
  - ⇨ falls ein rekursiver Aufruf die letzte Anweisung ist und keine weiteren Berechnungen stattfinden.
  - ⇨ Das Ergebnis des rekursiven Aufrufs ist das Ergebnis der gesamten Funktion.
- Entsprechender Algorithmus f kann wie folgt ausgedrückt werden, wobei s und r beliebige von f unabhängige Funktionen sind und R die Abbruchbedingung darstellt:

$$f(x) = \begin{cases} s(x) & \text{falls } R(x) \\ f(r(x)) & \text{sonst} \end{cases}$$

Endrekursion

- In diesem Fall ist streng genommen keine Rekursion notwendig, da eine gute Programmierumgebung die Rekursion leicht in eine Iteration umwandeln kann.
  - ⇨ In diesem Fall entfällt also der Overhead, der sich durch die Dynamik der rekursiven Stack-Bearbeitung ergibt.
- Die Frage ist also, lassen sich alle rekursiven Algorithmen in eine endrekursive Form überführen? – Nein
- Das folgende gilt:
  - ⇨ Endrekursive Algorithmen lassen sich einfach in eine iterative Variante überführen.
  - ⇨ Die Umwandlung nicht-endrekursiver Algorithmen ist deutlich schwieriger!

Beispiel: Umwandlung Endrekursion

```
// rekursiv
int fak (int n)  {
   if (n <= 1)
     return 1;
   else
     return n * fak(n-1);
}
```

Wg. Multiplikation nicht endrekursiv!

```
// endrekursiv
int fak (int n, int akk)  {
   if (n <= 1)
     return akk;
   else
     return fak(n-1, akk * n);
}

int fak (int n)  {
   return fak (n, 1);
}
```

```
// iterativ
int fak (int n)  {
   int akk = 1;
   while (n > 1)   {
      akk = akk * n;
      n --;
   }
   return akk;
}
```

Komplexität

- Für die algorithmische Lösung eines Problems unerlässlich:
  - dass der gefundene Algorithmus das Problem korrekt löst.
  - Wünschenswert: möglichst geringer Aufwand
  - Dies betrifft also die Effizienz bzgl. Rechenaufwand und Speicherplatz.
- Die Komplexitäts-Theorie liefert Aussagen, um den Aufwand von Algorithmen abzuschätzen. Unterscheidung in:
  - Rechenzeit-Aufwand: *Zeit-Komplexität*, oft in Rechenschritten gemessen.
  - Speicherplatz-Bedarf: *Speicher-Komplexität*, bestimmt den Umfang des zu reservierenden Speicherplatzes.
- Untersucht wird dabei der algorithmische Aufwand zur Lösung eines bestimmten Problems in Abhängigkeit der Eingabedaten, welche i.d.R. durch eine Problemgröße bestimmt wird.

Beispiel: Quadratzahl

- Zu einer gegebenen natürlichen Zahl n ∈ **N** soll das Quadrat n^2 berechnet werden, ohne die Multiplikation zu verwenden. Dazu verwenden wir die beiden (ähnlichen) Algorithmen, die sofort in einer Java-Anwendung codiert werden könnten:
  - ⇨ Algo1: $n^2 = \sum_{i=1}^{n} n$
  - ⇨ Algo2: $n^2 = \sum_{i=1}^{n} \sum_{i=1}^{n} 1$

- Frage: Wie viele Rechenoperationen/Rechenschritte müssen jeweils ausgeführt werden?
  - ⇨ Für Algorithmus Algo1 werden n Additionen durchgeführt.
  - ⇨ Für Algorithmus Algo2 werden n^2 Additionen durchgeführt.

Beispiel: Quadratzahl

- In diesem Fall wird das Problem durch die Zahl n festgelegt (Problemgröße), die relevanten Rechenschritte ergeben sich aus der Anzahl der notwendigen Additionen.
- Annahme: die Zeit, die ein Computer benötigt, um die Algorithmen auszuführen, ist proportional zur Anzahl der Rechenschritte
- Die Algorithmen zeigen somit prinzipiell unterschiedliches Verhalten:
  - ⇨ Die benötigte Rechenzeit für Algo1 steigt *linear* mit Problemgröße n.
  - ⇨ Die benötigte Rechenzeit für Algo2 steigt *quadratisch* mit Problemgröße n.

Programmlaufzeiten vs. Zeit-Komplexität

- Programmlaufzeiten hängen von verschiedenen Faktoren ab, wie:
  - ⇨ Eingabewerten (Problemgröße)
  - ⇨ Qualität des vom Compilers übersetzen Programms und des gebundenen Objektprogramms.
  - ⇨ Leistungsfähigkeit der Hardware
  - ⇨ Zeitkomplexität des angewendeten Algorithmus
  - ⇨ . . .
- Der Anwender/Entwickler hat i.d.R. bei der Erstellung der Software nur Einfluss auf den Algorithmus!
- Die Frage lautet nun:
  - ⇨ Wie kann die Zeit-Komplexität formal beschrieben werden?
  - ⇨ Gibt es genau eine Definition dafür, oder bieten sich verschiedene an?

Beispiel: Sequentielle Suche

- Gegeben seien die folgenden Werte:
  - ⇨ Eine Zahl $n \geq 1$,
  - ⇨ n Zahlen $a_1, \ldots, a_n$ die alle verschieden sind
  - ⇨ Und eine Zahl b.
- Gesucht wird
  - ⇨ Der Index $i = 1, 2, \ldots, n$, so dass $b = a_i$ ist, sofern ein solcher Index existiert.
  - ⇨ Sonst soll $i = n + 1$ ausgegeben werden.

Beispiel: Sequentielle Suche

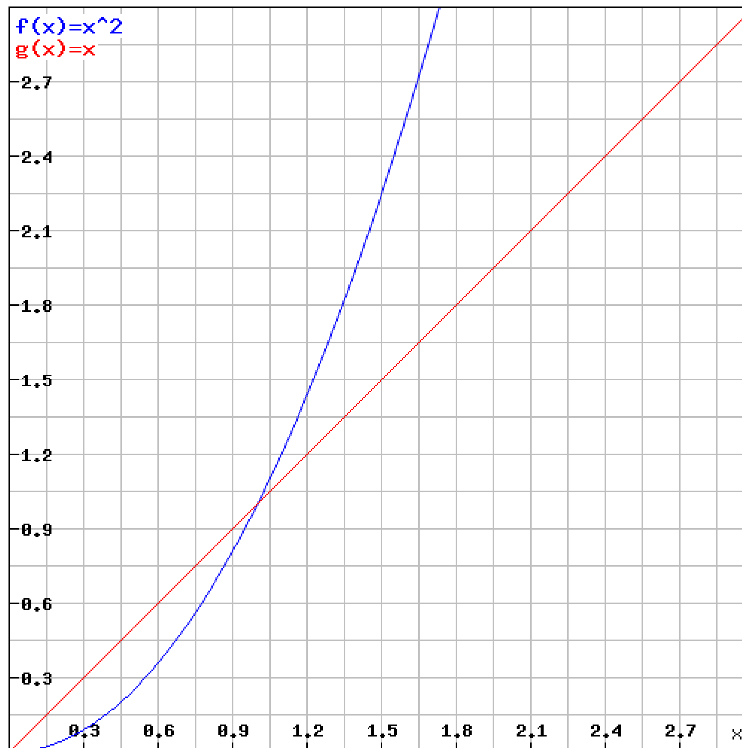
- Ein einfacher Algorithmus für dieses Suchproblem sieht wie folgt aus:
  - ⇨
    ```
    i := 1;
    while ( (i ≤ n) ∧ (b ≠ a_i) ) {
       i := i + 1;
    }
    ```
- Wie sieht der Aufwand der Suche aus, d.h. die Anzahl der Rechenschritte?
  - ⇨ Bei erfolgreicher Suche, wenn $b = a_i$ ist, werden i Schritte benötigt.
  - ⇨ Bei erfolgloser Suche werden n + 1 Schritte benötigt.
- Unterscheidung ist häufig sinvoll in:
  - ⇨ Bester Fall: Wie arbeitet der Algorithmus im günstigsten Fall?
  - ⇨ Schlechtester Fall: Wie arbeitet der Algorithmus im schlimmsten Fall?
  - ⇨ Durchschnittlicher Fall: Wie arbeitet der Algorithmus im Mittel?

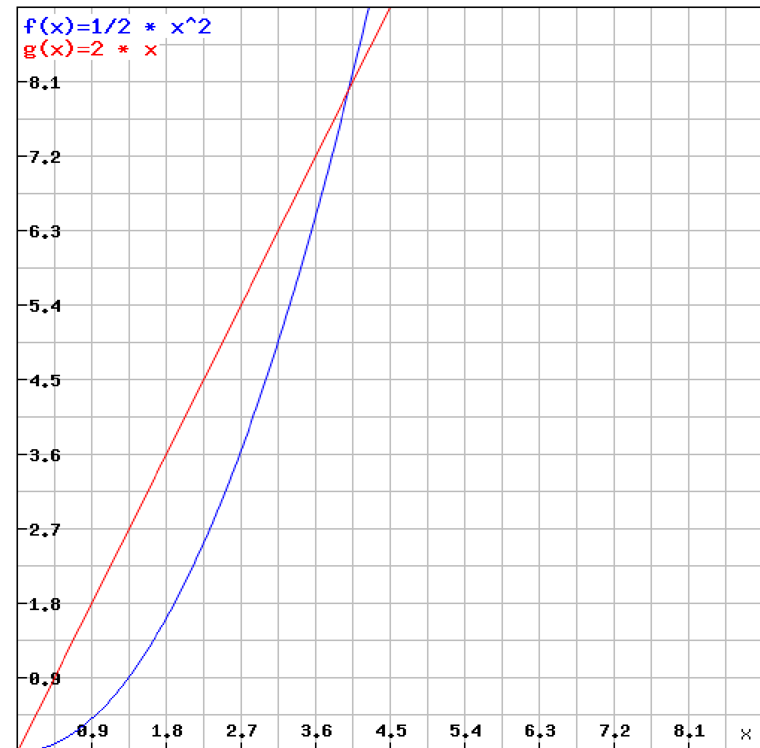
Obere Schranke: O-Notation

- Viele Details gehen zur Ermittlung der Programmlaufzeit ein
  - ⇨ Oft reicht es aus, die Zeit-Komplexität nur mit dem groben Skalierungsverhalten zu berücksichtigen.
- Zum Vergleich von Algorithmen eignet sich im Allgemeinen die *worst-case* Semantik am besten
- O-Notation definiert Klassen von Algorithmen, die sich im asymptotischen Fall identisch bzgl. ihrer Zeit-Komplexität verhalten:
  - ⇨ $f(n) \in O(g(n))$ g.d.w. $\exists$ **c, n_0** so dass $\forall$ **$n \geq n_0$**: $f(n) \leq c \cdot g(n)$
  - ⇨ Anschaulich bedeutet obiger Sachverhalt,
    dass die Funktion f nicht stärker wächst als die Funktion g.
  - ⇨ Dabei sind f und g zwei Funktionen von **N** → **N**.
- Für das Quadrat-Beispiel gelten die folgenden Aussagen:
  - ⇨ Algo1 $\in O(n)$ und Algo2 $\in O(n^2)$

Wachstum im Vergleich

- Es gibt immer ein n_0, ab dem eine Funktion $f \in O(n^2)$ stärker wächst als eine Funktion $g \in O(n)$





Untere Schranke

- Aus der Definition der O-Notation (obere Schranke) folgt, dass die Komplexität auch stets „schlechter“ angegeben werden kann:
  - Falls z.B. $f \in O(n^3)$, so gilt auch $f \in O(n^4)$.
- Um sinnvolle Aussagen über unsere Algorithmen zu bekommen, sind wir aber an einer möglichst guten Abschätzung interessiert!
- Im Gegensatz zur O-Notation gibt die o-Notation eine untere Schranke an:
  - $f(n) \in o(g(n))$ g.d.w. $\exists\ c, n_0$ so dass $\forall\ n \geq n_0$: $f(n) \geq c \cdot g(n)$
  - Anschaulich bedeutet obiger Sachverhalt,
dass die Funktion f mindestens so stark wächst wie die Funktion g.
- o-Notation auch manchmal mit Ω bezeichnet

Genaue Ordnung

- Damit kann insgesamt eine genaue Ordnung angegeben werden, die sinnvolle Aussagen unserer Algorithmen erlaubt:
  - ⇨ $\Theta(g) = O(g) \cap o(g)$
  - ⇨ Falls $f \in \Theta(g)$, sind f und g der gleichen Ordnung, d.h.
  - ⇨ Es gilt $f \in O(g)$ und $f \in o(g)$.

- Anmerkung: O, Ω, Θ werden manchmal auch als Landau'sche Symbole bezeichnet
  - ⇨ nach dem Zahlentheoretiker Edmund Landau (1877-1938)

Komplexitätsklassen

Aufwand	Sprechweise	Problemklasse
$O(1)$	konstant	Einige Tabellen-Suchverfahren (Hashing)
$O(\log n)$	logarithmisch	Allgemeine Tabellen-Suchverfahren, binäre Suche
$O(n)$	linear	Sequentielle Suche, Suche in Texten, syntaktische Analyse (best case)
$O(n \cdot \log n)$		Gute Sortierverfahren
$O(n^2)$	quadratisch	Einfache Sortierverfahren, einige dynamische Optimierungsprobleme
$O(n^3)$	kubisch	Einfache Matrizen-Multiplikation
$O(2^n)$	exponentiell	Viele Optimierungsprobleme, Bestimmung aller Teilmengen einer Menge, …

Hinweis: Konstanten entfallen bei der Bestimmung v. Komplexitätsklassen idR komplett

Einige Rechenregeln

Hierarchie

- $O(1) \subset O(\log_2 n) \subset O(n) \subset O(n \cdot \log_2 n) \subset O(n^k) \subset O(2^n)$
- $O(\log_k n) = O(\log_2 n)$, für $k \in \mathbf{N}$

Die Basis spielt keine Rolle

- $O(n^k) \subset O(2^l)$, für $k,l \in \mathbf{N}$, $k < l$
- $O\,(f(n) + g(n)) = O\,(\max\,(f(n), g(n)))$
- $O\,(f(n) \cdot g(n)) = O\,(f(n)) \cdot O\,(g(n))$
- Aus $f \in O(g(n))$ und $g \in O(h(n))$, folgt $f \in O(h(n))$

Einige Faustregeln

- Zur Bestimmung der Zeit-Komplexität eines Algorithmus ist es sinnvoll, den entsprechenden Laufzeitaufwand für alle relevanten Programmkonstrukte zu kennen:
  - ⇨ **Schleife**: Anzahl der Schleifendurchläufe · Laufzeit der teuersten Schleifenausführung
  - ⇨ **Geschachtelte Schleife**: Produkt der Größen aller Schleifen · Laufzeit der inneren Anweisung
  - ⇨ **Nacheinander-Ausführung**: Zunächst Addition der Laufzeiten der Anweisungen der Sequenz. Dann werden konstante Faktoren weggelassen und nur der jeweils höchste Exponent berücksichtigt.
  - ⇨ **Fallunterscheidung**: Laufzeit der Bedingungsanweisung + Laufzeit der teuersten Alternative.
  - ⇨ **Rekursiver Prozeduraufruf**: Anzahl der rekursiven Aufrufe · Laufzeit der teuersten Funktionsausführung.

Beispiele: Komplexitätsklassen

- Sortieren Sie die folgenden durch ihre O-Komplexitätsklassen gegebenen Algorithmen aufsteigend nach ihrer Komplexität:
  - ⇨ $O(27n^2)$
  - ⇨ $O(27 \cdot \log_2 n)$
  - ⇨ $O(3^n + 27)$
  - ⇨ $O(26n^2 + 2727n + 753)$
  - ⇨ $O(2727n \cdot \log_2 n)$
  - ⇨ $O(2727n)$

Beispiel: Fibonacci-Folge

- Der italienische Mathematiker Leonardo von Pisa (Filius Bonacci) fragte sich eines Tages, **wie viele Kaninchen** in einem eingezäunten Gehege pro Jahr geboren werden, wenn man davon ausgeht das:
  - ⇨ jeden Monat ein Paar ein weiteres Paar erzeugt
  - ⇨ Kaninchen zwei Monate nach der Geburt geschlechtsreif sind
  - ⇨ alle Kaninchen unsterblich sind.
- Mit F_n wird die Anzahl der Kaninchen Paare nach n Monaten beschrieben. Für die entsprechende Folge gilt dann:
- $F_0 = 0$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$
- Diese Zahlen werden Fibonacci-Zahlen genannt.
  - ⇨ Die ersten Fibonacci-Zahlen lauten:
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

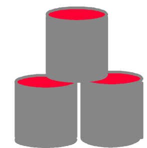
Fibonacci-Folge: Rekursive Lösung

- Aufwand?

```
public class Fibonacci {

  public static void main (String[] args) {
    int n = . . . ;

    long startzeit = System.currentTimeMillis(); // nur zur Zeiterfassung
    long ergebnis  = fibonacci(n);
    long endzeit   = System.currentTimeMillis(); // nur zur Zeiterfassung

    System.out.println("fibonaci (" + n + ") = " + ergebnis);
    System.out.println("Dauer war: " + (endzeit - startzeit) +" ms");
  } // main

  public static long fibonacci (int n) { // bestimme n-te Fibonacci Zahl
    return (n < 2)
           ? n
           : fibonacci(n - 1) + fibonacci(n - 2);
  }
} // Fibonacci
```

Grundlagen der Informatik
Veranschaulichung Komplexität

Prof. Dr. Bernhard Schiefer



Veranschaulichung Komplexitätstheorie

- Gegebener Rechner:
  - 1 Mio. Operationen/Sekunde (10^6 ops/sek)
- Gegeben seien 2 Algorithmen zur Lösung des Problems:
  - Variante 1: Laufzeit $1000 * n^2$
  - Variante 2: Laufzeit 2^n

Algorithmus 1

- Zeitkomplexität: $1000 * n^2$ Operationen (ops)
- Anwendung auf 10 Elemente:
  - ⇨ $1000 * 10^2 = 10^3 * 10^2 = 10^5$
  - ⇨ 10^5 ops / 10^6 ops/sek = 0,1 Sekunde Laufzeit
- Anwendung auf 100 Elemente:
  - ⇨ $1000 * 100^2 = 10^3 * 10^4 = 10^7$
  - ⇨ 10^7 ops / 10^6 ops/sek = 10 Sekunden Laufzeit
- Anwenden auf 1000 Elemente:
  - ⇨ $1000 * 1000^2 = 10^3 * 10^6 = 10^9$
  - ⇨ 10^9 ops / 10^6 ops/sek = 1000 Sekunden Laufzeit (~20 Minuten)

Algorithmus 2

- Zeitkomplexität: 2^n Operationen (ops)
- Anwendung auf 10 Elemente:
  - ⇨ $2^{10} = \sim 1000 = 10^3$
  - ⇨ 10^3 ops / 10^6 ops/sek = 0,001 Sekunden Laufzeit
- Anwendung auf 100 Elemente:
  - ⇨ $2^{100} = 2^{10*10} = \sim 1000^{10} = 10^{3*10} = 10^{30}$
  - ⇨ 10^{30} ops / 10^6 ops/sek = 10^{24} Sekunden Laufzeit
- Anwenden auf 1000 Elemente:
  - ⇨ $2^{1000} = 2^{10*100} = \sim 1000^{100} = 10^{300}$
  - ⇨ 10^{300} ops / 10^6 ops/sek = 10^{294} Sekunden Laufzeit

Vergleich bei n=10

- Zeitkomplexität: $1000 * n^2$ Operationen (ops)
  - ⇨ $1000 * 10^2 = 10^5$
  - ⇨ 10^5 ops / 10^6 ops/sek = 0,1 Sekunde Laufzeit

- Zeitkomplexität: 2^n Operationen (ops)
  - ⇨ $2^{10} = \sim 1000 = 10^3$
  - ⇨ 10^3 ops / 10^6 ops/sek = 0,001 Sekunden Laufzeit

Vergleich bei n=100

- Zeitkomplexität: $1000 * n^2$ Operationen (ops)
  - ⇨ $1000 * 100^2 = 10^7$
  - ⇨ 10^7 ops / 10^6 ops/sek = 10 Sekunden Laufzeit

- Zeitkomplexität: 2^n Operationen (ops)
  - ⇨ $2^{100} = 2^{10*10} = \sim 1000^{10} = 10^{30}$
  - ⇨ 10^{30} ops / 10^6 ops/sek = 10^{24} Sekunden Laufzeit

Vergleich bei n=1000

- Zeitkomplexität: $1000 * n^2$ Operationen (ops)
  - ⇨ $1000 * 1000^2 = 10^9$
  - ⇨ 10^9 ops / 10^6 ops/sek = 1000 Sekunden Laufzeit (~20 Minuten)

- Zeitkomplexität: 2^n Operationen (ops)
  - ⇨ $2^{1000} = 2^{10*100} = \sim 1000^{100} = 10^{300}$
  - ⇨ 10^{300} ops / 10^6 ops/sek = 10^{294} Sekunden Laufzeit

Zum Vergleich

- Ein Tag:
  - ⇨ 60 * 60 * 24 = 86.400 Sekunden = ~ 10^5 Sekunden
- Ein Jahr:
  - ⇨ 60 * 60 * 24 * 365 = 31.536.000 Sekunden = ~ 3 * 10^7 Sekunden
- Alter der Erde: ~4,5 Mrd. Jahre
  Alter des Universums: ~13,8 Mrd. Jahre
  - ⇨ 13,8 * 10^9 * 3 * 10^7 Sekunden = ~ 41,4 * 10^{16} Sekunden
- 10^{24} (bei n=100) entspricht also
  - ⇨ ~ 2.500.000 mal dem Alter des Universums