Lukas Müller

Programmieren für Studierende der Naturwissenschaften

V9 – Algorithmen

Heute: Algorithmen

- Wie kann man Algorithmen auf Papier schreiben?
- Mehr über die Komplexität

„ Algorithm design - that's the field where people talk about programs and prove theorems about programs instead of writing and debugging-programs."
An Introduction to Algorithm Design, Jon Louis Bentley Carnegie-Mellon University, 1979

Rückblick auf die erste Vorlesung

1. Problem beschreiben und analysieren
2. Auswahl, Entwicklung und Beschreibung der benötigten Algorithmen
3. Übertragung / Umsetzung in eine Programmiersprache
4. Test des Programms

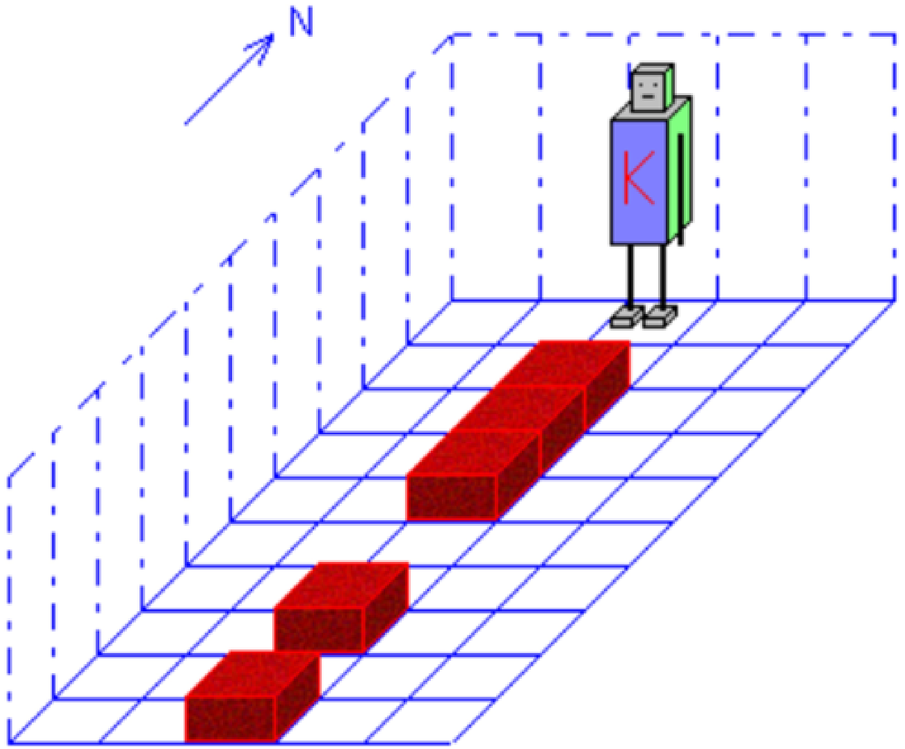
Das alles ist augenscheinlich nicht trivial!

Möglichkeiten vor der Implementierung

- Natürlichsprachlich (Text/Stichworte)
- Pseudocode (Angelehnt an Code, es gibt keine festen Regeln)
- Grafisch (Ablaufdiagramme, Zustandübergangsdiagramme)

- In der Praxis oft eine Mischung.

Beispiel:



Quelle: http://www.inf-schule.de/algorithmen/algorithmen/algorithmusbegriff/exkurs_darstellung, 16.09.2017

Roboter-Beispiel

- Solange die Wand noch nicht erreicht ist, wiederhole Folgendes: Wenn ein Ziegel im Weg liegt, dann hebe ihn auf und gehen einen Schritt weiter. Ansonsten gehe direkt einen Schritt weiter. Drehe dich um 180° Grad (???). Solange die Wand noch nicht erreicht ist, gehe einen Schritt weiter. Drehe dich um 180° Grad.

Roboter-Beispiel

- Solange die Wand noch nicht erreicht ist, wiederhole Folgendes:
- Wenn ein Ziegel im Weg liegt:
  - dann hebe ihn auf
  - und gehe einen Schritt weiter.
- Ansonsten:
  - gehe direkt einen Schritt weiter.
- Drehe dich um 180° Grad.
- Solange die Wand noch nicht erreicht ist:
  - gehe einen Schritt weiter
- Drehe dich um 180° Grad.

Beispiel (nicht mehr Roboter)

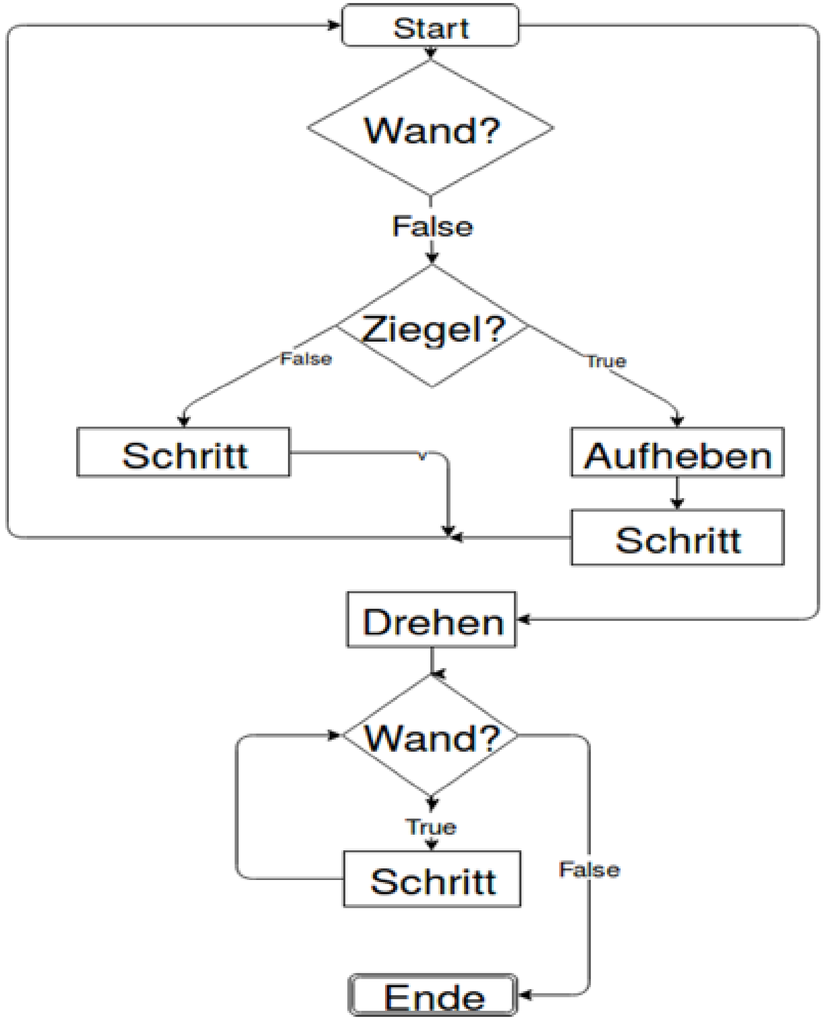
Division

Algorithm 2: Division

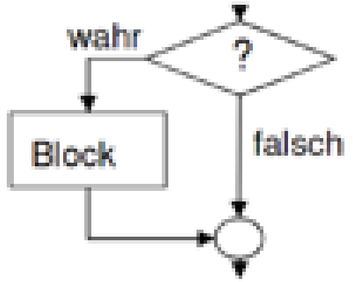
```
1  function divide (x, y);
   Input: Two n-bit integers x and y, where y ≥ 1
   Output: The quotient and remainder of x divided by y
2  if x = 0 then
3  |   return (q, r) = (0, 0)
4  else
5  |   set (q, r) = divide(⌊x/2⌋, y);
6  |   q = 2 × q, r = 2 × r;
7  |   if x is odd then
8  |   |   r = r + 1
9  |   end
10 |   if r ≥ y then
11 |   |   r = r − y, q = q + 1
12 |   end
13 |   return (q, r)
14 end
```

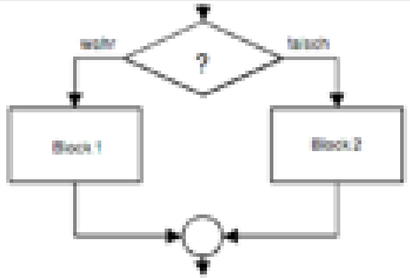
Roboter-(Negativ-)Beispiel

- Das sieht toll aus,
- Ist jedoch nicht richtig

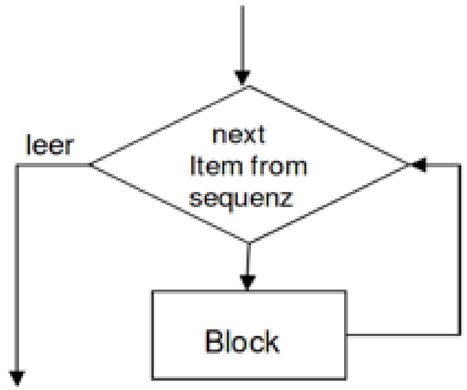


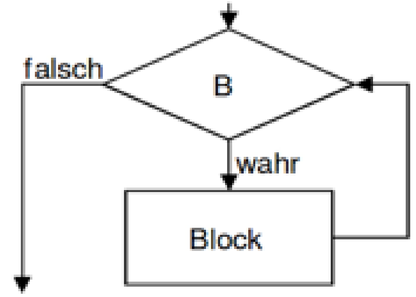
Programmablaufdiagramme und Pseudocode

Pseudocode	Ablaufplan
if ‹Bedingung› **then** ‹Block › **endif**;	

Pseudocode	Ablaufplan
if ‹Bedingung› **then** ‹Block 1›; **else** ‹Block 2›; **endif**;	

Programmablaufdiagramme und Pseudocode

Schleifenart	Ablaufplan
foreach - Schleife **For** <Iteratinsvariable> **in** sequenz do Block Meint:"für jedes Element in der Sequenz führe "Block" genau einmal aus."	

Schleifenart	Ablaufplan
vorprüfend (Kopfgesteuert) **while** (B) **do** Block	

Kritik an Programmablaufplänene

1. Schon bei mittelgroßen Algorithmen schnell unübersichtlich
   - Erfordert viel Platz
2. Verführt zur Verwendung von expliziten Sprunganweisungen
3. Korrigiert man einen Denkfehler, müsste gegebenenfalls vieles im Ablaufplan „nachgezogen“ werden

In der Praxis selten anzutreffen

Aber: sind gut geeignet, elementare Strukturen der Programmierung zu verdeutlichen und den Einstieg zu erleichtern!

Kurzer Exkurs zur Komplexität

- Messen, wie „effizient“ ein Programm arbeitet
  - Unsinnige Anweisungen oder ineffiziente Strukturen können viel Rechenzeit und/oder Speicher kosten

- Optimierung:
  - Oft Zeitgewinne mit größeren Speichermengen oder Speicheroptimierung mit längeren Rechenzeiten
  - Nicht selten sind Zeitgewinne auf Kosten der Genauigkeit. Eine approximierte Lösung in einer Stunde ist manchmal besser, als eine exakte in 40.000 Jahren

- Konsequenz:
  - Wenn eine solide Effizienzoptimierung angestrebt wird, dann bereits in der konzeptuellen Phase
  - Nachträgliche Effizienzsteigerung ist oft mit hohem Aufwand verbunden.

Wenn ein Algorithmus erst einmal gefunden wurde...

Eigenschaften von Algorithmen

- Korrektheit
- Terminierbarkeit
- **Komplexität**
  - Rechenzeit
  - Speicherplatz

Analyse von Algorithmen

- **Rechenzeit**
  - Anzahl der durchgeführten Elementaroperationen in Abhängigkeit von der Eingabegröße
- **Speicherplatz**
  - Der maximale Speicherverbrauch während der Ausführung des Algorithmus in Abhängigkeit von der Komplexität der Eingabe
- **Datentransfer**
  - Wie groß sind die Daten, die übertragen werden müssen

Rechenzeit

- **1. Ansatz**
  - Direktes Messen der Laufzeit (z.B. in ms): Abhängig von vielen Parametern, wie Rechnerkonfiguration, Rechnerlast, Compiler, Betriebssystem…
  - Deshalb: kaum übertragbar und ungenau

- **2. Ansatz:**
  - Zählen der benötigten Elementaroperationen des Algorithmus in Abhängigkeit von der Größe n der Eingabe
  - Das algorithmische Verhalten wird als Funktion der Anzahl der benötigten Elementaroperationen dargestellt.
  - Kategorisierung in verschiedene Mengen

In Abhängigkeit von den Eingabedaten

- **Eingabedaten** charakterisieren

- Meistens ist es sehr schwer eine genaue Verteilung der Daten zu finden, die dem realen Fall entspricht. Deswegen müssen wir in der Regel **den schlimmsten Fall betrachten** und eine obere Schranke für die Laufzeit finden

- Wenn diese **obere Schranke** korrekt ist, garantieren wir, dass die Laufzeit unseres Algorithmus für beliebige Eingabedaten immer kleiner oder gleich dieser Schranke ist

- **Beispiel:**

  - Ich möchte eine Liste sortieren. Wie viele Schritte benötige ich – in Abhängigkeit von der Eingabeliste - maximal?

Welche Operationen werden betrachtet?

- Kleine Operationen, die mehrere kleinere Operationen zusammenfassen, welche einzeln in konstanter Zeit ausgeführt werden, aber den gesamten Zeitaufwand des Algorithmus durch ihr häufiges Vorkommen wesentlich mitbestimmen.

- Laufzeit einzelner Elementar-Operation ist dann abhängig von der eingesetzten Rechner-Hardware

- **Beispiel:**
  - Bei Sortieralgorithmen: # Vergleiche
  - Bei anderen Algorithmen:
    - Speicherzugriffe
    - Anzahl der Multiplikationen
    - Anzahl der Bitoperationen
    - Anzahl der Schleifendurchgänge
    - Anzahl der Funktionsaufrufe

Die O-Notation

- Mit der O-Notation haben Informatiker einen Weg gefunden, die asymptotische Komplexität (bzgl. Laufzeit oder Speicherplatzbedarf) eines Algorithmus zu charakterisieren.

- Die O-Notation erlaubt es, Algorithmen auf einer höheren Abstraktionsebene miteinander zu vergleichen (unabhängig von Implementierungsdetails, wie Programmiersprache, Compiler und Hardware-Eigenschaften)

- O(...) bezeichnet die Menge aller Funktionen, die bezüglich der Eingabe die gleiche Komplexitätsklasse (d.h. bis auf einen konstanten Faktor) besitzen

$$2 \cdot n \in O(n)$$

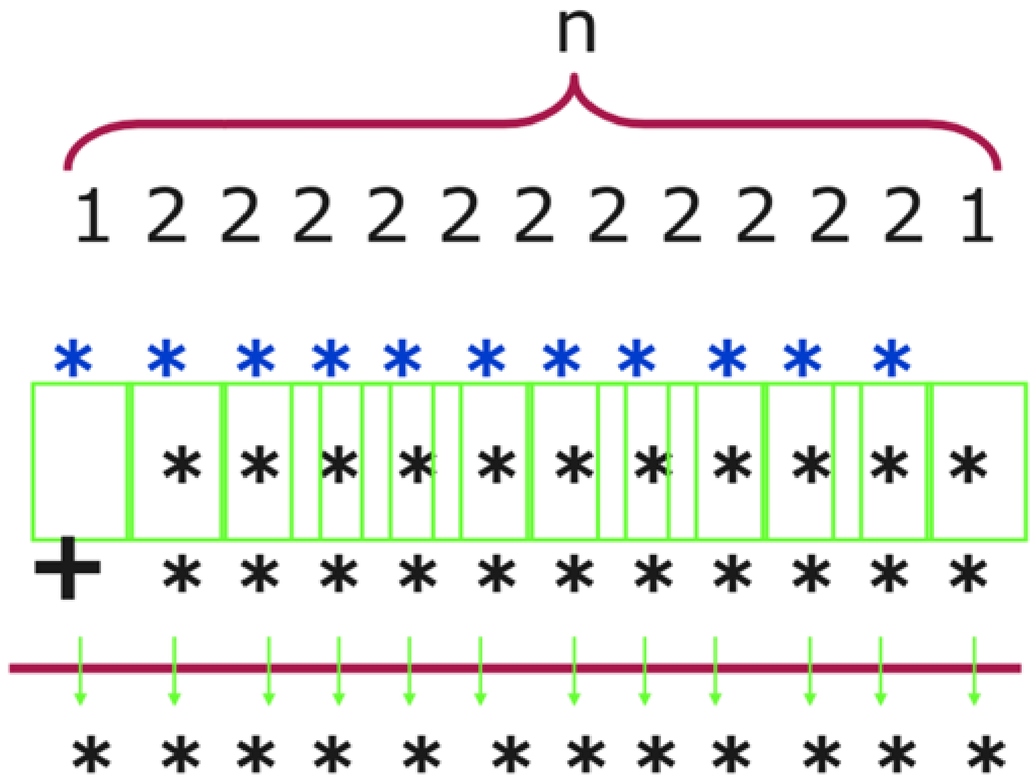
$$n/2 + 1 \in O(n)$$

An dieser Stelle keine Definition, sondern ein Beispiel

Beispiel: Schriftliche Addition und Multiplikation

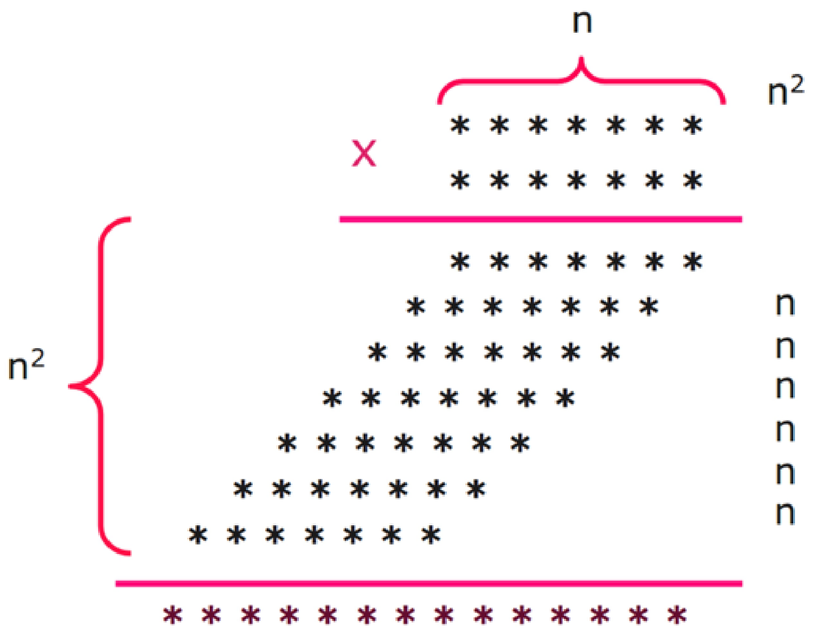
- **Eingabegröße:** n = Stellen der Zahl

- **Berechnungsschritt:**
  - Addition von 2 Ziffern
  - Ggf. Übertrag
- **Komplexitätsanalyse:**
  - T(n) = Anzahl der Berechnungsschritte, um zwei Zahlen mit n Ziffern zu addieren
- **Worst case:**
  - T(n) = 2n (Additionen)
  - → linear → Klasse O(n)



Beispiel: Schriftliche Addition und Multiplikation

- Eingabegröße: n = Stellen der Zahl
- Berechnungsschritt:
  - Multiplikation von 2 Ziffern
  - Addition der Ergebnisse
- Worst case:
  - T(n) = n Multiplikationen+ n^2 +n Additionen
  - → quadratisch → Klasse O(n^2)



Komplexitätsklassen

	Sprechweise	Typische Algorithmen / Operationen
$O(1)$	konstant	Addition, Vergleichsoperationen, rekursiver Aufruf, …
$O(\log n)$	logarithmisch	Suchen auf einer sortierten Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		gute Sortierverfahren
$O(n \cdot \log^2 n)$		
…		
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k)$, k≥2	polynomiell	
…		
$O(2^n)$	exponentiell	Ausprobieren von Kombinationen

Noch ein Beipsiel:

```
def sum(n):
    summe =0
    for i in range(n+1):
        summe += i
    return summe
```

```
def  sum(n):
    return n*(n+1)/2
```

- Schleife durchläuft n-Mal.
  - Darin: konstante Operationen (Addition und Zuweisung).
  - → n * 1 Operationen
  - → O(n)

```
def sum(n):
    summe =0
    for i in range(n+1):
        summe += i
    return summe
```

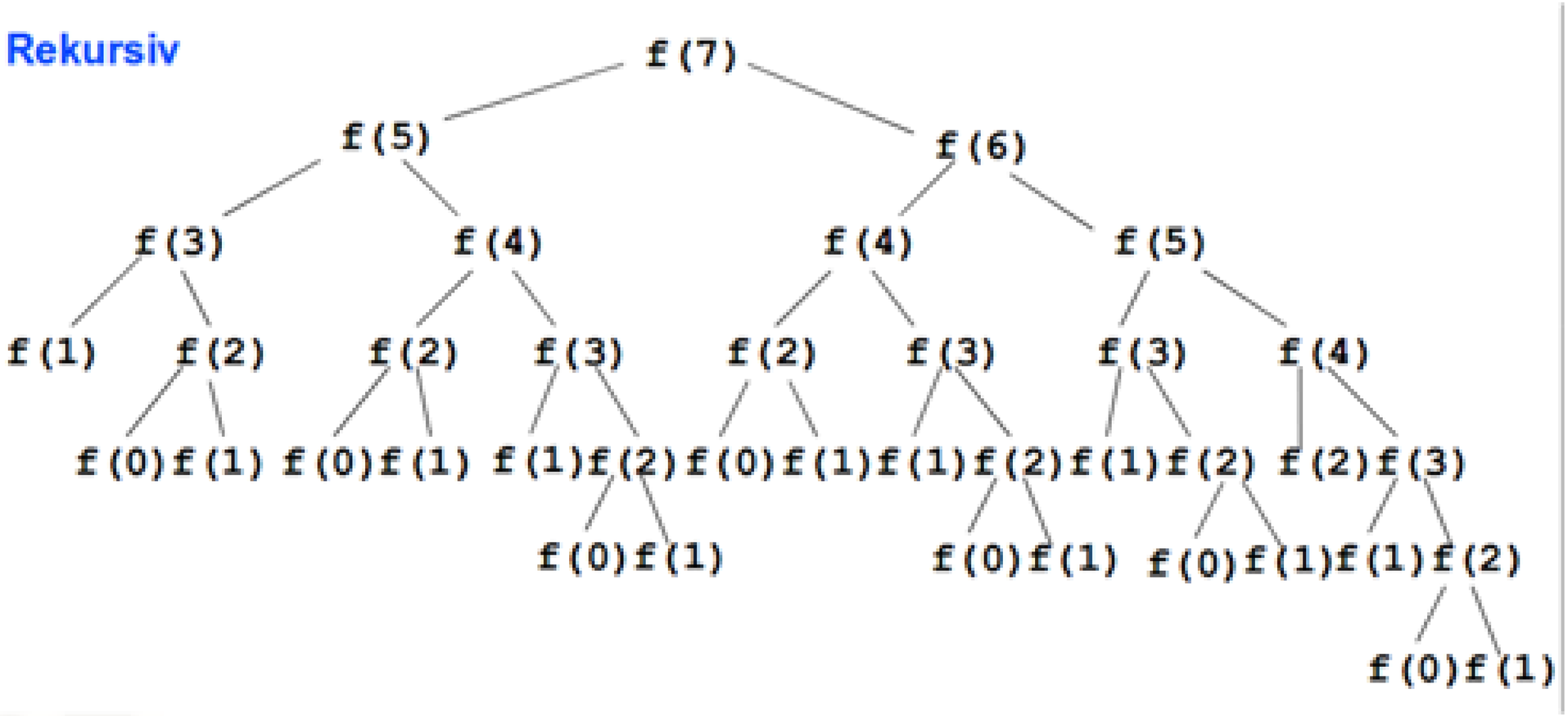
- Nach Gauß
  - → 3 konstante Operationen
  - → O(1)

```
def  sum(n):
    return n*(n+1)/2
```

- Fibonacci-Folge: 1,1,2,3,5,8,13,21,34...
  - Das nächste Folgenglied ist immer die Summe der beiden vorausgehenden.
  - Taucht häufig in der Natur auf (z.B. Goldener Schnitt, Wachstumsraten bei Pflanzen, Populationswachstum etc.)
  - Wird häufig rekursiv (also als Funktion, die sich selbst aufruft) beschrieben:

```
fib(n) = fib(n-1)+fibn(n-2)
```

Und noch ein Beispiel



Fazit

- Die Wahl des Algorithmus kann Einfluss auf Laufzeit des Programms haben
- Die Informatik hat ein System gefunden, diese Unterschiede zu Kategorisieren.
- Bei kleinen Programmen merkt man vielleicht keine Unterschiede, sobald Probleme allerdings größer werden, kann Laufzeit von Bedeutung werden