Definitionen für den Algorithmus

- **Don Knuth**
  muß terminieren, eindeutig interpretierbar, festgelegte Eingabegrößen, Ergebnis hängt von Eingabe ab, Mensch kann mit Bleistift nachmachen
- **Aho, Hopcroft, Ullman**
  endliche Folge Instruktionen, endliche Zeit pro Instruktion, eindeutig interpretierbar, enthalten Wiederholungsinstruktionen, terminiert
- **Bauer, Goos**
  präzise Beschreibung, in festgelegter Sprache, Verwendung elementarer Verarbeitungsschritte
- **Reichenberg**
  endliche Folge von Schritten mit jeweils Ein-/Ausgabe-Spezifikation

Der Algorithmusbegriff

- Algorithmen gab es schon lange vor den Computern (z.B: Ariadne, Euclid)
- Algorithmus ≠ Programm
- Churchsche These:
  „Für jeden Algorithmus, der intuitiv berechenbar ist, gibt es in jeder Programmiersprache (mit ausreichender Mächtigkeit) ein entsprechendes Programm.“
- Für jedes „Programm“ (egal ob in λ-Notation, Java, Turing-Maschine, Assembler, Gopher, Pseudocode) gibt es in in jeder anderen Notation eins, das dasselbe tut.

Was sind Algorithmen (nicht) ?

- **Algorithmus ≠ Programm**
  Für jeden Algorithmus gibt es unendlich viele Programme, die den Algorithmus ausführen
- **Algorithmus ≠ Problem**
  Für jedes (lösbare) Problem gibt es unendlich viele Algorithmen, die das Problem lösen.
- **Algorithmus =** Sammlung von exakt beschriebenen Aktionen und deren Ausführungsreihenfolge

  Also: kein Interpretationsspielraum, keine Aktionen wie: „nimm eine runde Zahl“

Was sind Algorithmen (nicht) ?

- **Ampelsteuerung**
  keine Eingabe, endlich viele Beschreibungsschritte, terminiert nicht, keine finale Ergebnisgröße
- **Briefe Verschicken**
  schlecht Mathematisch beschreibbar (was sind Ein-/Ausgabe-Größen?), einzelne Anweisungen relativ unklar
- **Textverarbeitungsprogramm**
  wohl eher eine Sammlung vieler einzelner Algorithmen
- **Matrix invertieren**
  endlich, klar beschreibbar, klare Ein-/Ausgabegrößen, eindeutig

Problem - Algorithmus - Programm

Problem: Berechne *s*, die Summe der ersten *n* natürlichen Zahlen

Algorithmus 1:

setze s=0; setze i=1;
solange i<=n: {setze s=s+i und i=i+1 }

Algorithmus 2:

setze s = (n+1)*n/2

Programm 1a:

```
int s=0;
for (int i=0; i<=n; i++) s+=i;
```

Programm 1b:

```
int i=1; s=0;
while (i<=n) s+=i;
```


Wozu Algorithmen?

Beispielproblem: Brief(e) verschicken

- einmalige Geschichte:
  - Brief ansehen, orientieren
  - Überlegen, wie er gefaltet werden muß
  - Brief falten
  - Umschlag in die Hand nehmen, orientieren
  - Brief hineinlegen
  - Umschlagdeckel ablecken, andrücken
- Verschicken von 1000 Briefen:
  - ggf. selben Schritt erst mal für alle Briefe ausführen
  - ggf. Briefe und Umschläge zurechtlegen (keine Orientierung)
  - ggf. Umschlagdeckel nicht selbst ablecken, u.s.w.

Wozu Algorithmen?

Im allgemeinen:

Algorithmen sind hilfreich:

- als Rezept (sonst wird beste Vorgehensweise vergessen, und nach jedem Schritt könnte neu geplant werden müssen)
- als Abstrakte Arbeitsbeschreibung (durchführbar von jemandem, der keine Ahnung hat, was er tut)
- als Methode, Prozesse zu optimieren (Zeitersparnis bei sich oft wiederholenden Vorgängen)

Ein paar Gedanken zu Algorithmen

Wie würden Sie einen Algorithmus entwerfen, um:

- 32 Spielkarten zu sortieren? (Und wie, wenn Sie blind wären?)
- 32 Karten mit den Zahlen 37, 11, 89, 112, 3, 45, 44, 28, 29, 289, ...
- *n*! (die Fakultät von *n*) zu berechnen?
- die physikalischen Abläufe einer Atombombe zu simulieren?
- das Wetter vorherzusagen?
- einen automatischen Psychotherapeuten zu programmieren?
- festzustellen, ob 1234567891 eine Primzahl ist?
- festzustellen, welche Wörter aus der Bibel bei Goethe vorkommen?
- ein Spiegelei zu braten?
- einen *general problem solver* zu programmieren?

Ein paar Gedanken zu Algorithmen

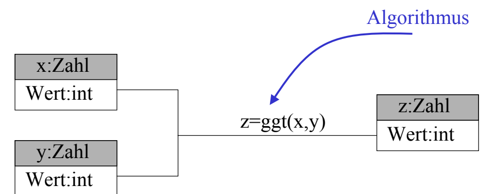
Wie würden Sie einen Algorithmus entwerfen, um:

- Spielkarten
- beliebige Karten
- *n*!
- Atombombe simulieren
- Wettervorhersage
- Psychotherapeut
- 1234567891 prim?
- Bibel / Goethe
- Spiegelei
- general problem solver

Algorithmendesign und OO-Design

Typischerweise ist mit Algorithmus ein Teil eines Softwaresystems gemeint, der beim OO-Design fast ganz außer acht gelassen wird.

Softwaresystem zur Berechnung des GGT:



Einige Eigenschaften von Algorithmen

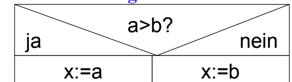
Definitionsbereich	Auf was für Daten arbeitet der Algorithmus?
zeitlicher Aufwand	Wie schnell / langsam läuft er?
räumlicher Aufwand	Wieviel Speicherplatz benötigt er?
Determinismus	Zufall? Mehrere Lösungen? Alle Lösungen?
Terminierung	Wann stoppt der Algorithmus? Tut er das?
Korrektheit	Liefert er immer das richtige Ergebnis?
Algorithmenschema	Nach welchem Muster läuft er ab?
Spezifikationen	Wie kann man sein Wirken exakt beschreiben?
Operationsweise	Funktional? Imperativ? Deklarativ?
Eleganz	Nachvollziehbarkeit? Verständlichkeit?

Darstellungen von Algorithmen

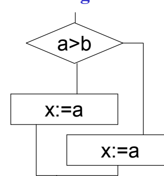
Umgangssprachlich

SET drücken;
solange MODE drücken,
bis SELECT erscheint;
zwei mal SET drücken

Struktogramme



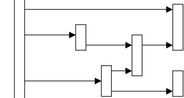
Flußdiagramme



Pseudocode

```
gib a und b ein
falls a < b
   x := a
sonst
   x := b
drucke x aus
```

UML-Szenario



Einige Designprinzipien für Algorithmen

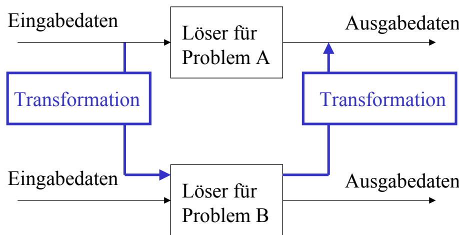
- **Induktion**
  schließe aus Lösung für *n*-1 auf Lösung für *n*
- **teile und herrsche**
  teile Problem in einfache Teile, kombiniere Ergebnisse
- **probabilistisch**
  Lösung zufällig erraten
- **brute force**
  alle Lösungskandidaten ausprobieren und nachprüfen
- **schrittweises Verfeinern**
  grobe Schritte => feine Schritte
- **branch and bound / backtracking**
  versuche verschiedene Lösungswege und verfolge die „guten“
- **greedy** (schrittweises Ausschöpfen)
  wähle unter allen möglichen Schritten, den „billigsten“
- **parallel**
  führe merere Schritte gleichzeitig (nebenläufig, parallel) aus
- **besondere**
  indeterministisch, genetisch, neuronal, vorberechnen, u.a.

Problemlösung durch Reduktion

- Problem 1:
  - Informatiker ist im Keller, hat Aufgabe: Spiegelei braten.
  - Problemlösung:
    1. Gehe Treppe hoch in die Küche
    2. Hole Pfanne aus dem Schrank, lege sie auf den Herd
    3. Schalte Herd ein - optional: etwas Öl in die Pfanne
    3. Hole Ei aus dem Kühlschrank, haue es in die Pfanne
    4. Wenn Ei schön, schalte Herd ab
- Problem 2:
  - Informatiker ist in der Küche
  - Problemlösung:
    1. Gehe in Keller und löse Problem 1

Problemlösung durch Reduktion

Im allgemeinen bedeutet Problemreduktion:



Problemlösung durch Brute Force

Beispiel:

gegeben: int x, y
gesucht: int z = ggt(x,y)

Lösung:

```
for (int i=1; i<x; i++)    if  (x % i == 0 && y & i == 0)   z=i;
```

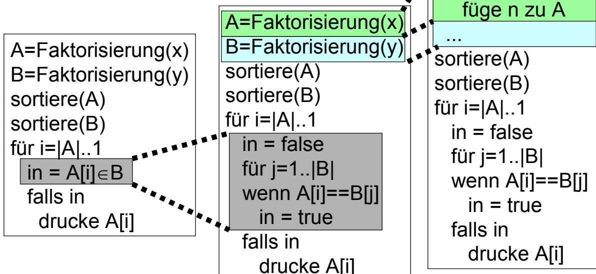
Rechnet offensichtlich viel zu lange für sehr große Werte von x

Brute Force Lösungen bieten sich nur selten an:
- wenn kein „richtiger“ Algorithmus bekannt
- wenn Entwicklung des Algorithmus „teurer“ als CPU-Ressourcen

Problemlösung durch schrittweises Verfeinern

Beispiel:

gegeben: int x, y
gesucht: int z = ggt(x,y)



Problemlösung mit Greedy Algorithmus

Beispiel:

gegeben: int x[] = { 7, 22, 39, 20, 9 }
Aufgabe: packe möglichst viel in Rucksack mit Kapazität 50.

Optimale Lösung: 22 + 20 + 7 = 49

Greedy Algorithmus:

1.	39	paßt	⇒	39 drin
2.	22	paßt nicht	⇒	39 drin
3.	20	paßt nicht	⇒	39 drin
4.	9	paßt	⇒	48 drin
5.	7	paßt nicht	⇒	48 drin (suboptimal)

Problemlösung durch Probabilistische Algorithmen

- probabilistische Algorithmen sind *deterministisch* (verwenden reproduzierbar generierte Zufallszahlenfolgen)
- liefern nicht immer optimales Ergebnis
- sinnvolle Einsatzgebiete:
  - bei Optimierungsaufgaben, bei denen Optimumsuche zu teuer ist, und Näherungslösung akzeptabel ist
  - wenn der probabilistische Algorithmus in der Regel schneller ist und nur in Spezialfällen viel langsamer

Beispiel für probabilistischen Algorithmus

gegeben: Zahlen x[0], x[1], X[2] (3 natürliche Zahlen)
gesucht: deren Maximum

Algorithmus:

```
i = Zufall(0,1,2);
j = Zufall(0,1,2);
if x[i] > x[j]
  return x[i]
else
  return x[j]
```

Mit welcher Wahrscheinlichkeit kommt das richtige Ergebnis?

(In-)Determinismus

probabilistische Algorithmen
- sind deterministisch (gleiche Eingabe ⇒ gleiche Ausgabe)
- verwenden reproduzierbare Zufallszahlenfolgen

indeterministische Algorithmen
- liefern verschiedene Ergebnisse bei gleicher Eingabe
- Auswahl des Ergebnisses ist zufällig
- Programmablauf ist nicht rekonstruierbar

komplexitätstheoretische nichtdeterministische Algorithmen
- sind Entscheidungsalgorithmen
- verfolgen verschiedene mögliche Berechnungswege
- liefern eindeutiges Ergebnis (immer 1 oder 0)

Beispiel für parallelen Algorithmus

gegeben: Zahlen x[0], x[1], ... X[n] (*n* natürliche Zahlen)
gesucht: deren Summe

Algorithmus:

```
a=Pozessor1(Summe(0,n/2))
b=Prozessor2(Summe(0,n/2))
return a+b
```

Parallele Algorithmen machen Rechenoperationen, die zeitlich nicht voneinander abhängen „gleichzeitig“ oder „nebenläufig“

Algorithmenentwurf durch Induktion oder Teile und Herrsche

Beide Methoden haben gemeinsam:

- Problem ist gelöst (z.B. vordefiniert) für triviale Fälle
- jeder nichttriviale Fall kann dadurch gelöst werden, daß seine Lösung eine „einfache“ Folgerung aus der Lösung eines einfacheren Falls ist
- jeder nichttriviale Fall wird schließlich bis zu den trivialen Fällen „heruntergebrochen“

Algorithmenentwurf durch Induktion oder Teile-und-Herrsche

Beispiel:
- gegeben: Zahlenfolge x[0], x[1], ... X[N] (natürliche Zahlen)
- gesucht: deren Maximum

Naheliegender Algorithmus:

```
m = 0;
for (i=1..N)
    if (x[i]>m)
       m=x[i];
return m;
```

Algorithmenentwurf durch Induktion

Beispiel:
- gegeben: Zahlenfolge x[0], x[1], ... X[N] (natürliche Zahlen)
- gesucht: deren Maximum

Induktiver Algorithmus: (Aufruf mit max(N))

```
Prozedur max(p)
        if (p==0)
                return x[0];
        m = max(p-1);
        if (m>x[p])
                return m
        else
                return x[p];
```

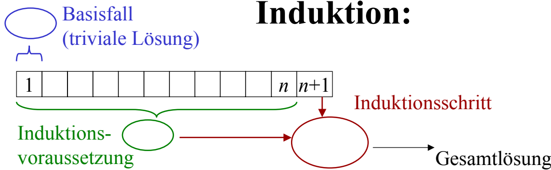

Algorithmenentwurf durch Teile und Herrsche

Beispiel:
- gegeben: Zahlenfolge x[0], x[1], ... X[N] (natürliche Zahlen)
- gesucht: deren Maximum

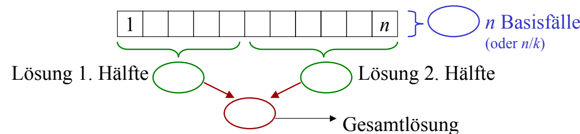
Teile-und-Herrsche-Algorithmus: (Aufruf mit max(0,N))

```
Prozedur max(l,r)
   if (l==r) return x[l];
   m1 = max ( l          ,  (l+r)/2   );
   m2 = max ( (l+r)/2+1  ,  r         );
   if (m1>m2)   return m1
   else         return m2
```


Induktion:



Teile-und-Herrsche:



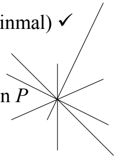
Vollständige Induktion?

Behauptung: n schiefe Geraden schneiden sich immer in einem Punkt

Beweis durch vollständige Induktion:

1. Basisfall: $n = 2$ (2 schiefe Geraden scheiden sich einmal) ✓

2. Induktionsschritt:
   Voraussetzung: die Geraden 1 bis n-1 scheiden sich in P
   aber auch: die Geraden 2 bis n scheiden sich in Q



   Da alle Geraden schief, muß gelten $P = Q$ (sonst wären 2 gleich)
   ⇒ auch n schiefe Geraden schneiden sich in einem Punkt

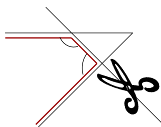
Vollständige Induktion?

Behauptung: es gibt keine Quadrate (Polygone mit nur rechten Winkeln)

Beweis durch Induktion über Anzahl Ecken

1. Induktionsanfang: n=3 (Dreiecke sind nicht quadratisch) ✓

2. Induktionsschritt: geben nicht-nurrechtwinkliges Polygon (n Ecken)



nehmen wir eine Ecke und schneiden sie ab
⇒ n+1-Eck, die beiden neuen Winkel können nicht beide 90º sein, (sonst war da keine Ecke)

⇒ es bleibt immer ein nicht rechter Winkel
⇒ es gibt keine Quadrate

Vollständige Induktion

Behauptung: $n! = n(n\text{-}1)!$

Beweis durch Vollständige Induktion

1. Induktionsanfang:
$1! = 1(1\text{-}1)! = 1 \cdot 1 = 1$ ✓

2. Induktionsschritt:
$(n\text{-}1)! = 1 \cdot 2 \cdots (n\text{-}1)$
$n(n\text{-}1)! = (n\text{-}1)! \cdot n = 1 \cdot 2 \cdots (n\text{-}1) \cdot n = n!$ ✓

3. Folgerung: weil die Behauptung für n=1 gilt, und
weil aus der Gültigkeit für n-1 die für n folgt,
gilt die Behauptung für alle n

Vollständige Induktion

Entwurf des Algorithmus:

```
Funktion Fakultät(n∈N) → N
    // Induktionsanfang
    falls n <= 1
        Funktionswert = 1
    // Induktionsschritt
    sonst
        Funktionswert = n·Fakultät(n-1)
    gib Funktionswert zurück
```


Verwendung von abstrakten Datentypen

Idee: Implementiere eine „Black-Box“, mit exakt spezifizierten Eingabe und Ausgaben und deren Zusammenhang

Eigentliche Implementierung ist für „Außenwelt“ unwichtig.

Vorteile: Implementierung kann verändert werden, ohne dass benutzende Programmteile beeinflusst werden

Programm kann in Baukasten-Manier aus ADTs (= fertige Bauklötze) zusammengebaut werden

Disziplinierung u. ä. Vorteile wie bei Kapselung

Nachteile: Ressourcenverschwendung durch Verwenden komplexer ADTs, wo einfacher Speziallösung möglich wäre

Beispiel für abstrakten Datentyp

Typ Stapel (wie in Info I gesehen):

bietet vier Operationen an:
`void push(int)` und `int pop()`
`boolean isempty()` und `void clear()`

Definition:

clear	leert den Stapel (enthält keine Daten)
push	legt ein Element auf den Stapel
isempty	gibt an ob der Stapel leer ist
pop	entfernt das zuletzt „gepushte“ Element und gibt es als Funktionswert zurück, oder Programmabsturz wenn Stapel leer

Noch ein Beispiel für ADT

Typ Binärbaum:

bietet vier Operationen an:
`void split (Node,Object,Object)` und `Node childL (Node)`
`Node childR (Node)` und `Node find (Object)`

Definition:

split	erzeugt zwei Nachfolgeknoten für einen Knoten
find	liefert Referenz auf gesuchten Knoten
childL	liefert Referenz auf linken Nachfolger zu geg. Kn.
childR	liefert Referenz auf rechten Nachfolger

Und noch ein Beispiel für ADT

Typ Schlange:

bietet drei Operationen an:
`boolean isEmpty ( )` und `void append (Object)`
`Object removeFirst ( )`

Definition:

isEmpty	gibt an, ob die Schlange leer ist
append	hängt ein Objekt ans Ende an
removeFirst	entfernt den Kopf der Schlange

Andere Beliebte ADTs

Liste, Tabelle, Baum, Graph, Matrix, Menge, ...

Bemerkung: In Java gibt es *abstrakte Klassen*. Das bedeutet nur, daß es keine Objekte gibt davon gibt. Die haben nichts mit Abstrakten Datentypen zu tun.

Oft sind die Methoden der abstrakten Klassen „leer“, müssen also in Unterklassen überschrieben werden.

Wie macht man Spezifikationen

- informell (umgangssprachlich, Pseudocode)
- gegeben-gesucht-Invarianten
- UML statische und dynamische Modelle, Zustandsautomaten
- Logik (Prädikatenlogik, Funktionen, Prädikate, Relationen)
- diverse Tools (produzieren ggf. sogar Code)
- Flußdiagramme, Struktogramme (ggf. informell)
- Grammatiken, Termersetzungssysteme
- mathematische Funktionsdefinitionen
- spezielle Spezifikationssprachen (teilw. domänenabhängig)

Wie macht man Spezifikationen

Beispiel für Spezifikation:

Algorithmus zur Lösung der Gleichung $a_2x^2+a_1x+a_0 = 0$

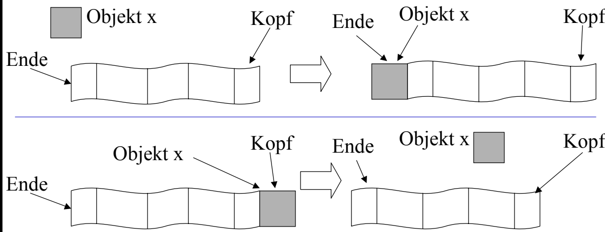
- verständlich für den typischen Informatiker,
- aber:

- nicht spezifiziert,was gegeben und was gesucht ist
- nicht spezifiziert, welche Typen a_i und x haben (int, float, ...)
- nicht spezifiziert, was die Ausgabe ist (eine Lösung?, beide?)
- nicht spezifiziert, was im Falle keiner Lösung zu tun ist
- nicht spezifiziert, in welchem Format Ein- und Ausgabe vorliegt

Ein ADT von Spezifikation bis Programm

1. Umgangssprachliche Beschreibung:

eine Klasse Schlange (FIFO) , deren Objekte folgendes können:
- beliebige Objekte können sich hinten anstellen
- das Objekt am Kopf der Schlange kann entfernt werden



Ein ADT von Spezifikation bis Programm

2. Formale Spezifikation:

2a: Signatur:

void anstellen(Objekt)	new:	$\rightarrow$ Schlange
Objekt abbauen( )	anstellen:	Schlange $\times$ Objekt $\rightarrow$ Schlange
	abbauen:	Schlange $\rightarrow$ Objekt $\times$ Schlange

Wir zerlegen die Operation abbauen in zwei einwertige Operationen:

abbauen( ) =		getKopf:	Schlange $\rightarrow$ Objekt
	und	ausketten:	Schlange $\rightarrow$ Schlange

Jetzt haben wir nur einwertige Funktionen und können mit Hilfe einfacher mathematischer Formeln spezifizieren:

Ein ADT von Spezifikation bis Programm

2b: mathematische Spezifikation

abbauen: Schlange $\rightarrow$ Objekt $\times$ Schlange

getKopf ⇢ Objekt, ausketten ⇢ Schlange

getKopf (ε) = null
getKopf (anstellen(x,ε)) = x
getKopf (anstellen(x,anstellen(y,s))) = getKopf (anstellen(y,s))

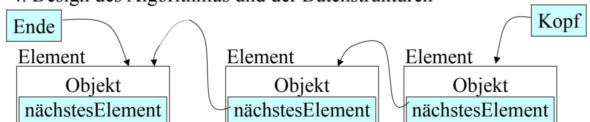
ausketten (ε) = null
ausketten (anstellen(x, ε)) = ε
ausketten (anstellen(x,anstellen(y,s))) = anstellen(x,ausketten(anstellen(y,s))

Ein ADT von Spezifikation bis Programm

3. Definition der Randbedingungen

- ggf. maximale Anzahl von Objekten in der Schlange
- Definition der Reaktion auf Fehler wie:
  - Objekt konnte nicht angestellt werden (z.B. weil kein Speicher)
  - Schlange konnte nicht abgebaut werden (z.B. weil leer)
- ggf. Reaktionszeiten und Ausnahmen spezifizieren

4. Design des Algorithmus und der Datenstrukturen



Ein ADT von Spezifikation bis Programm

5. Implementierung

```
class Schlange {

    class Element {
        Object object;
        Element next;
    }

    Element Kopf, Ende;

    public Schlange ( ) {
        Kopf = null;
        Ende = null;
    }

public void anstellen (Object x)
{
  if (Ende == null) {
    Ende = new Element( ); Kopf = Ende; }
  ...
}

public Object abbauen ( )
{
   if (Kopf == null) return null;
   else return Kopf.object;

   if (Kopf.next == null) { Kopf=null; Ende=null; }
   else ...
}
```


Ein paar Gedanken zu Algorithmen

Wie würden Sie einen Algorithmus entwerfen, um:

- Spielkarten • Jede Karte hat ihre definierte Position
- beliebige Karten • Vermutlich durch sukzessives Einfügen
- $n!$ • Am besten als Tabelle für die ersten $n < N$
- Atombombe simulieren • Klar, aber keine Ahnung von Atomphysik
- Wettervorhersage • Klar definiert, aber zu chaotisch
- Psychotherapeut • Unklar definiert
- 1234567891 prim? • Einfaches Programm: PRINT “ist prim“
- Bibel / Goethe • Natürlich nicht jedes Wort mit jedem vergl.
- Spiegelei • Erst mal in den Keller gehen?
- general problem solver • Völlig unmöglich (nachweisbar)