Kapitel 5
Algorithmenmuster

- Grundmuster von Algorithmen (**Pattern**, **Paradigmen**)
- Vorteile:
  - Verständnis von Algorithmen
  - Orientierung bei der Entwicklung von Lösungsalgorithmen
  - gemeinsame Komplexitätsanalyse für Algorithmenmuster

Algorithmenmuster
bekannte Muster

- Verkleinerungsprinzip
- schrittweise Annäherung
- Divide and conquer (teile und herrsche)
- Greedy-Methode (greedy=gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

Algorithmenmuster
bekannte Muster

- Verkleinerungsprinzip
- schrittweise Annäherung

- Divide and conquer (teile und herrsche)
- Greedy-Methode (greedy=gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

Algorithmenmuster
bekannte Muster

- Verkleinerungsprinzip
- schrittweise Annäherung
- Divide and conquer (teile und herrsche)
- Greedy-Methode (greedy=gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

Algorithmenmuster
bekannte Muster

- Verkleinerungsprinzip
- schrittweise Annäherung
- Divide and conquer (teile u
- Greedy-Methode (greedy=gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

Algorithmenmuster
bekannte Muster

- Verkleinerungsprinzip
- schrittweise Annäherung
- Divide and conquer (teile und herrsche)
- Greedy-Methode (greedy=gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

Algorithmenmuster
bekannte Muster

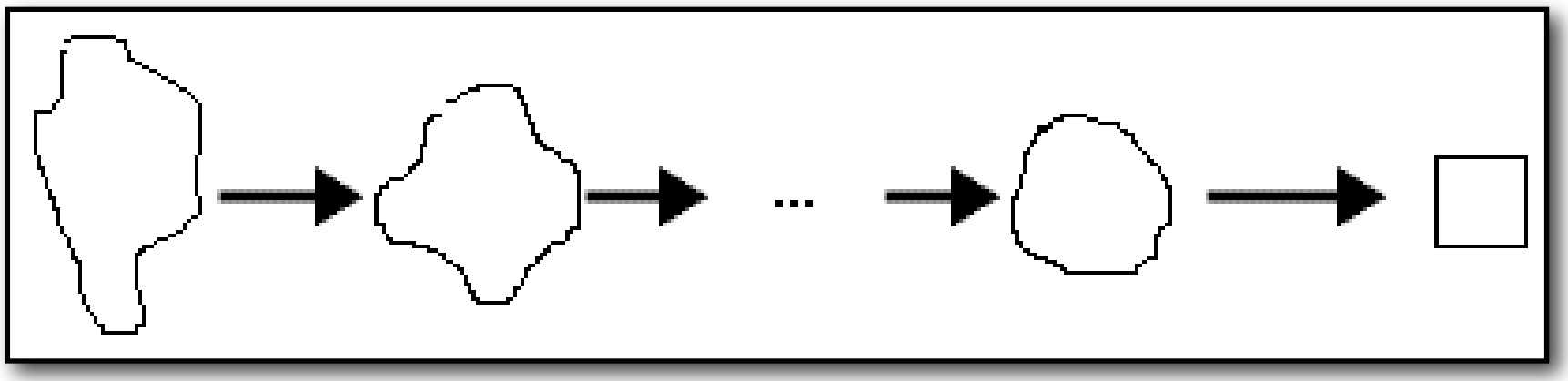
- Verkleinerungsprinzip
- schrittweise Annäherung
- Divide and conquer (teile und herrsche)
- Greedy-Methode (g gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

Algorithmenmuster
bekannte Muster

- Verkleinerungsprinzip
- schrittweise Annäherung
- Divide and conquer (teile und herrsche)
- Greedy-Methode (greedy=gierig)
- Backtracking
- dynamisches Programmieren
- Branch and bound

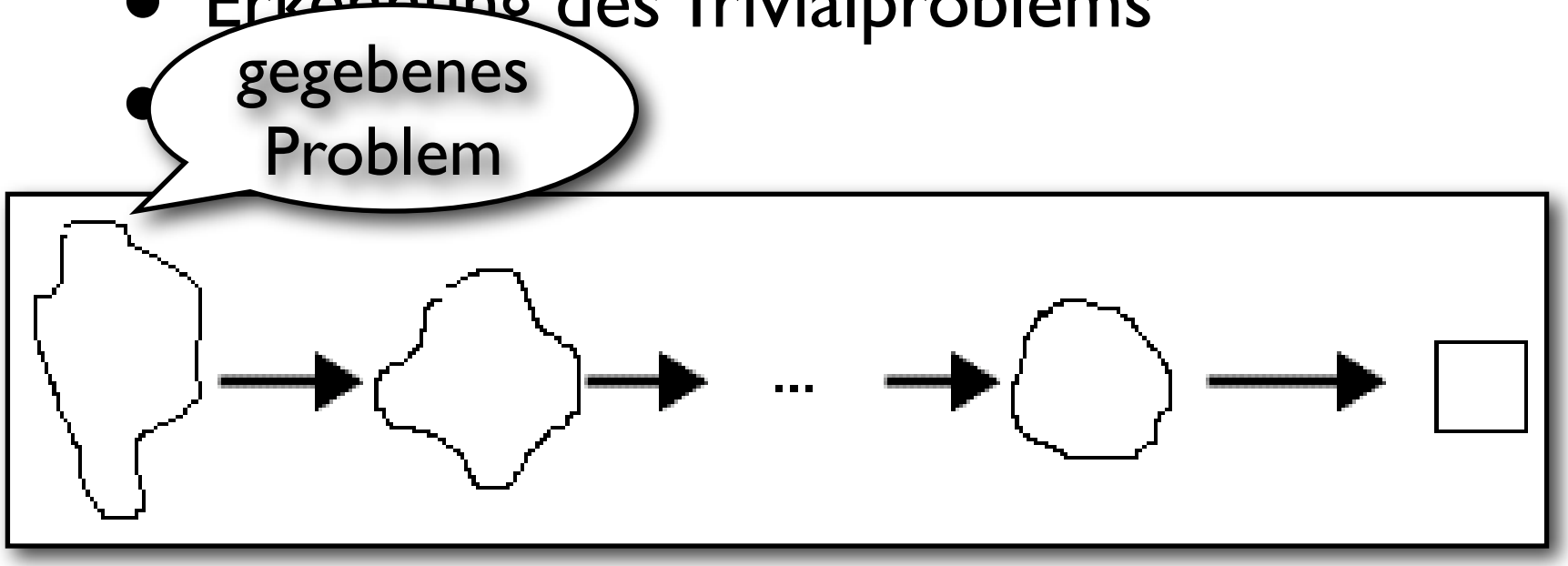
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt “verkleinert”, bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



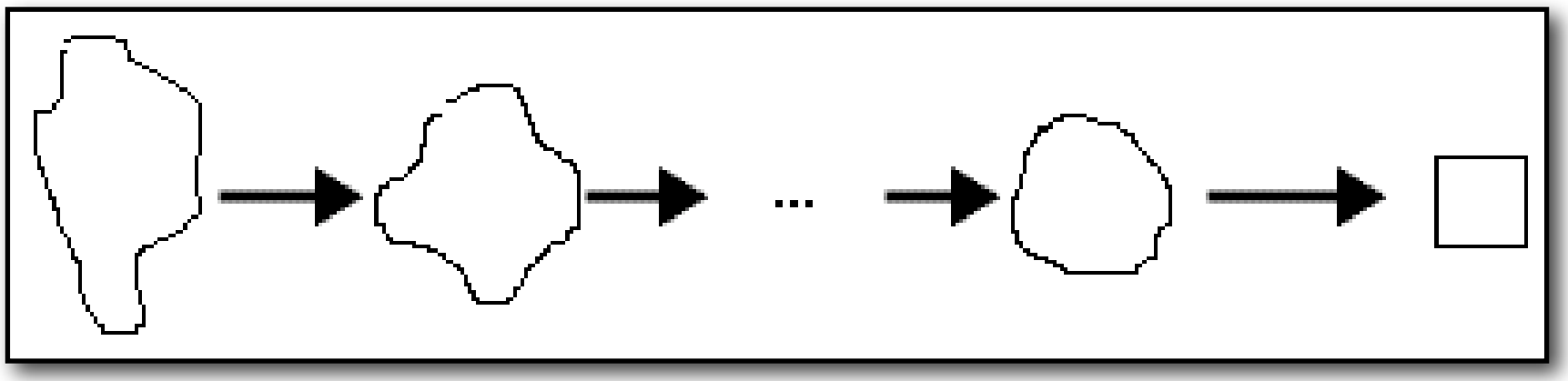
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt "verkleinert", bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  -



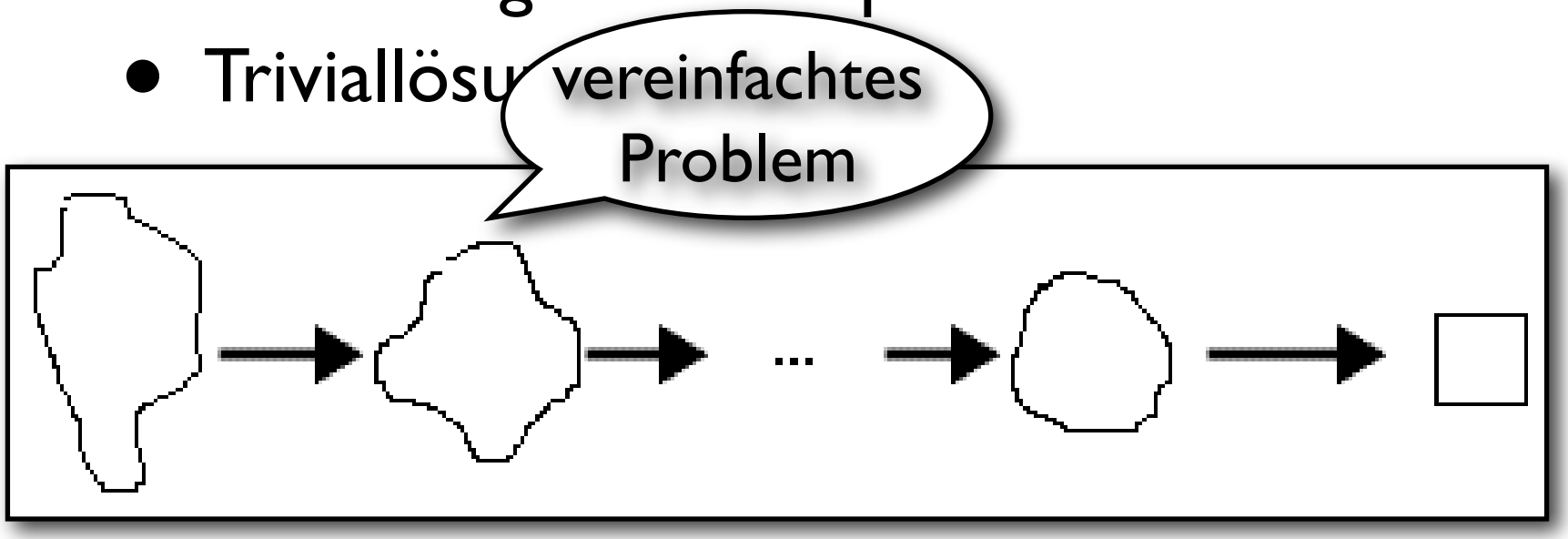
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt “verkleinert”, bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



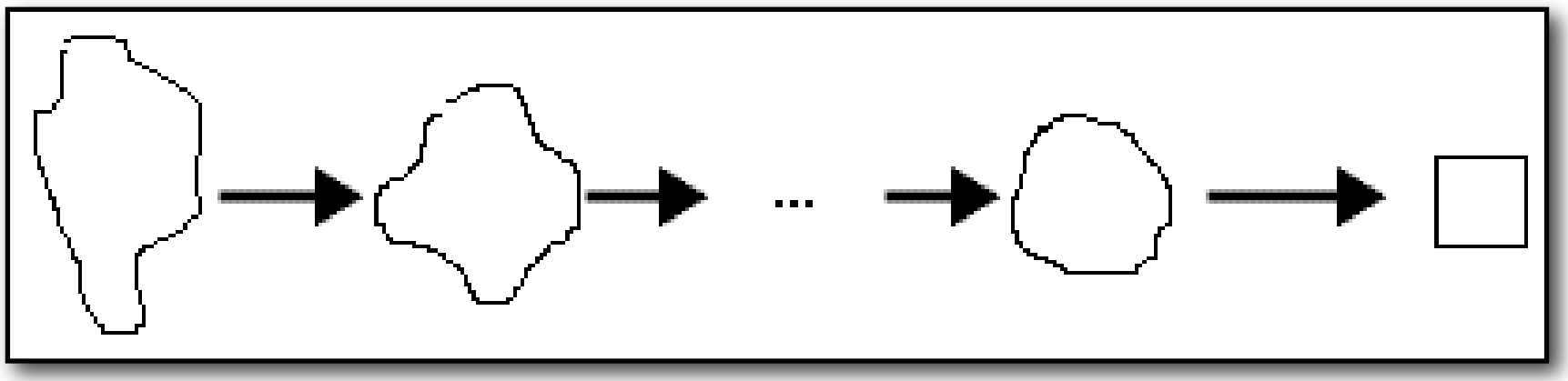
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt "verkleinert", bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösu



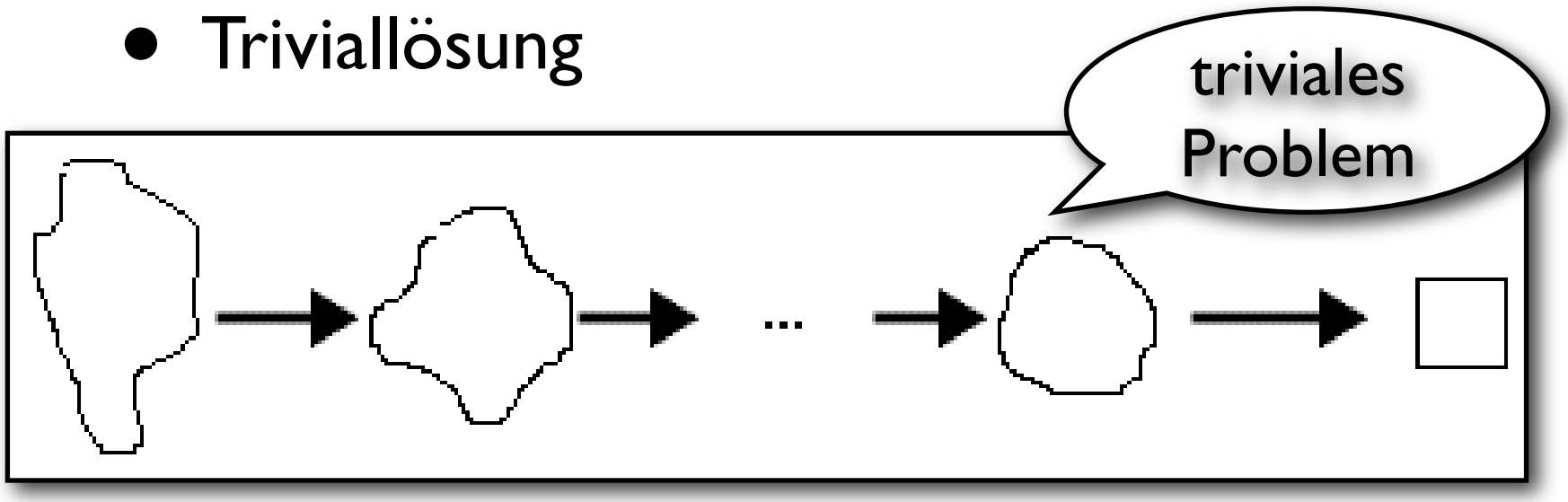
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt “verkleinert”, bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



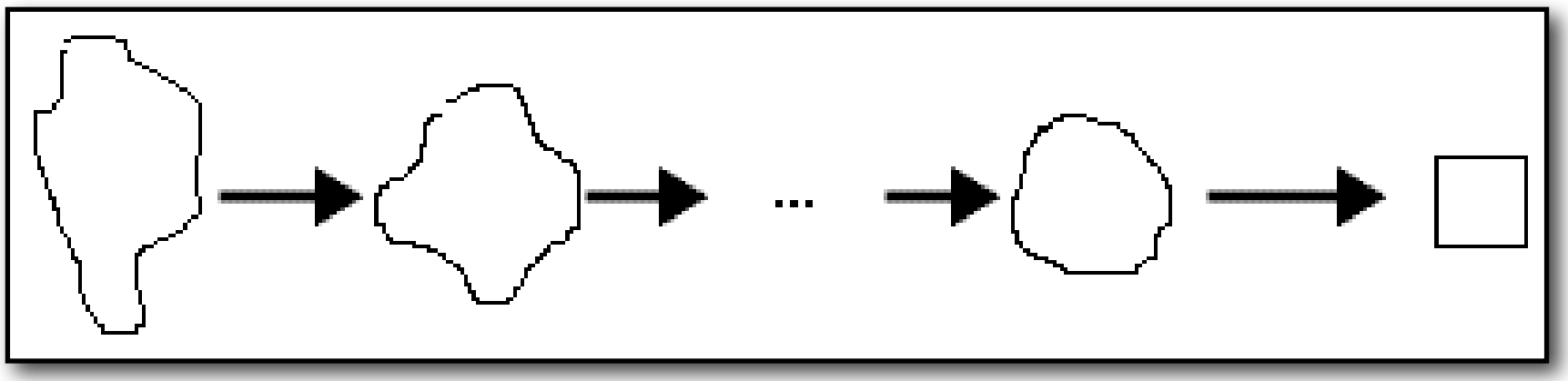
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt "verkleinert", bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



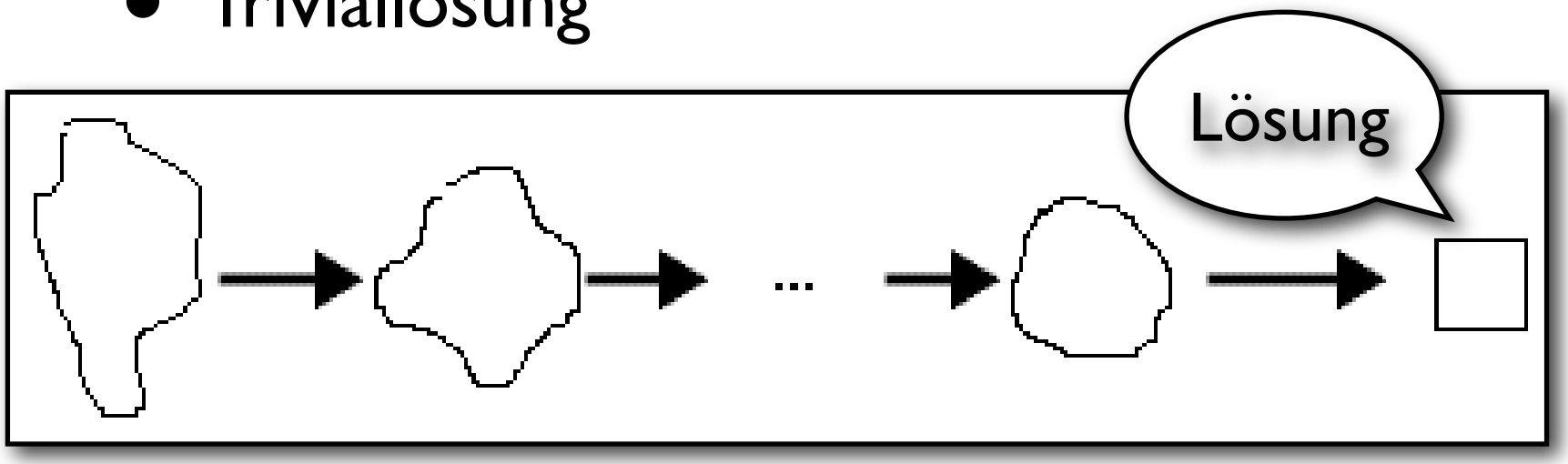
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt “verkleinert”, bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



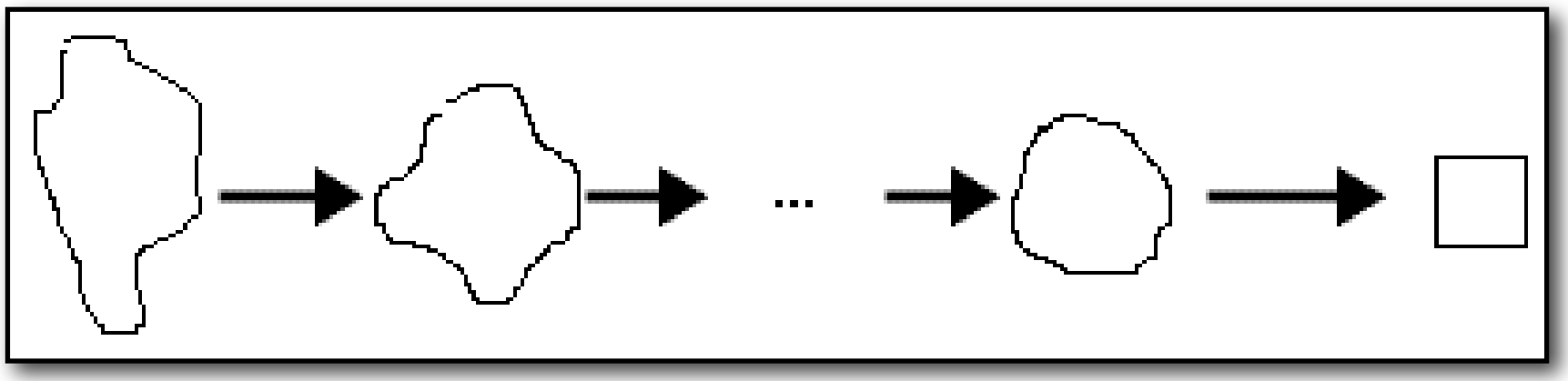
Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt "verkleinert", bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



Algorithmenmuster
Verkleinerungsprinzip

- Problem wird in jedem Schritt “verkleinert”, bis ein triviales Problem vorliegt.
- Algorithmenelemente:
  - Verkleinerungsschritt
  - Erkennung des Trivialproblems
  - Triviallösung



Algorithmenmuster
Verkleinerungsprinzip - Schema

- iteratives Muster:

  while “Problem nicht trivial” do

  “verkleinere Problem”;

  “löse triviales Problem”

- rekursives Muster:

  function verkleinern (P: Problem): Lösung;

  if “P trivial” then “Lösung für P” else

  verkleinern(“verkleinere P”)

Algorithmenmuster
Verkleinerungsprinzip - Komplexität

- Typische Varianten:
  - je Verkleinerungsschritt reduziert sich Problem der Größe n um festen Anteil a:

    Laufzeit: $O(T(n)+T(n-a)+T(n-2a)+...)=O(n \cdot T(n))$
  - je Verkleinerungsschritt reduziert sich Problem der Größe n proportional um Faktor a ($0<a<1$):

    Laufzeit: $O(T(n)+T(a \cdot n)+T(a^2 \cdot n)+...)=O(T(n) \cdot \log n)$

Algorithmenmuster

Verkleinerungsprinzip - Beispiel ggT

Eingabe: natürliche Zahlen a, b

Ausgabe: ggT (a, b)

Methode:

```
while b≠0 do
begin
    r := a mod b;
    a := b; b := r
end;
write (a)
```

Algorithmenmuster

Verkleinerungsprinzip - Beispiel ggT

Eingabe: natürliche Zahlen a, b

Ausgabe: ggT (a, b)

Methode:

```
while b≠0 do
begin
    r := a mod b;
    a := b; b := r
end;
write (a)
```

hier: proportionaler Verkleinerungsschritt.
Daher: O(log a) Schleifendurchläufe

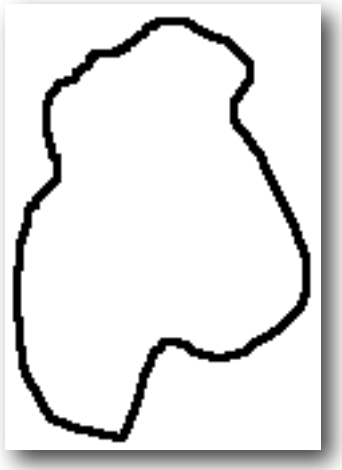
Algorithmenmuster
Divide and Conquer

- Zerlegung eines Problems in zwei (oder mehr) Teilprobleme gleicher Beschaffenheit (**divide**)
- Lösung der Teilprobleme
- Zusammensetzen der Teillösungen zur Gesamtlösung (**conquer**)

Algorithmenmuster

Divide and Conquer - Schema

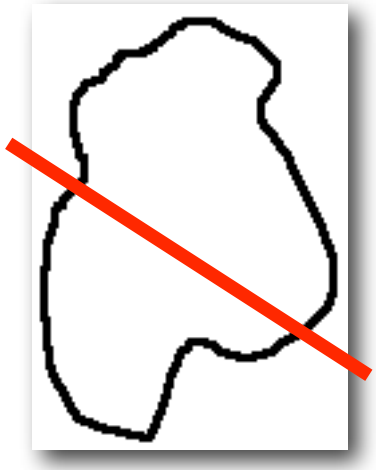
Algorithmenmuster
Divide and Conquer - Schema



Problem

Algorithmenmuster

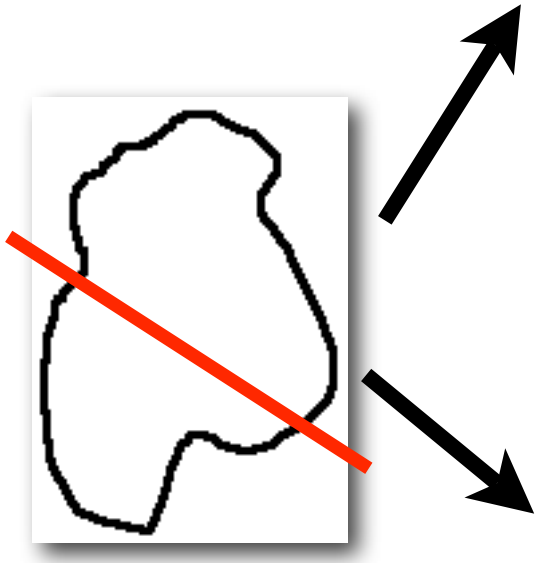
Divide and Conquer - Schema



Problem

Algorithmenmuster

Divide and Conquer - Schema

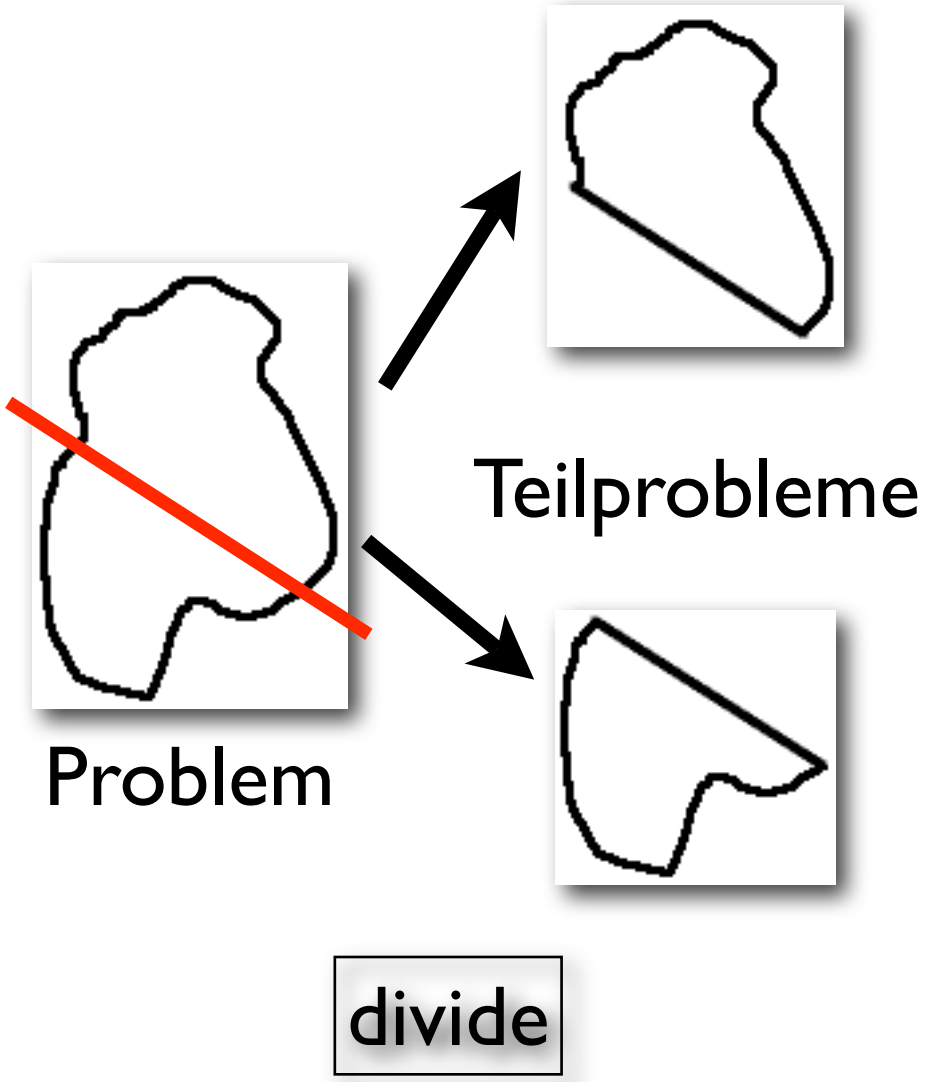


Problem

divide

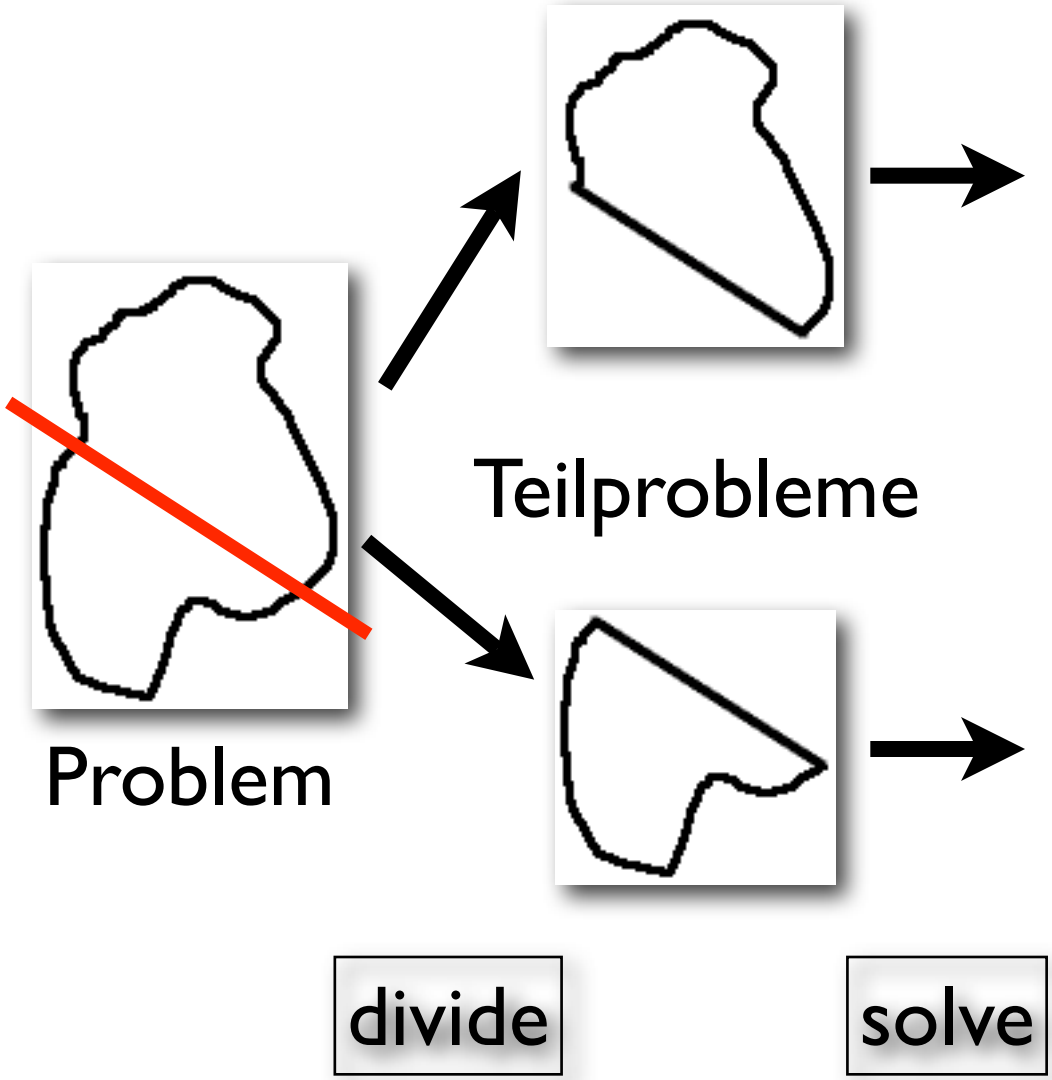
Algorithmenmuster

Divide and Conquer - Schema



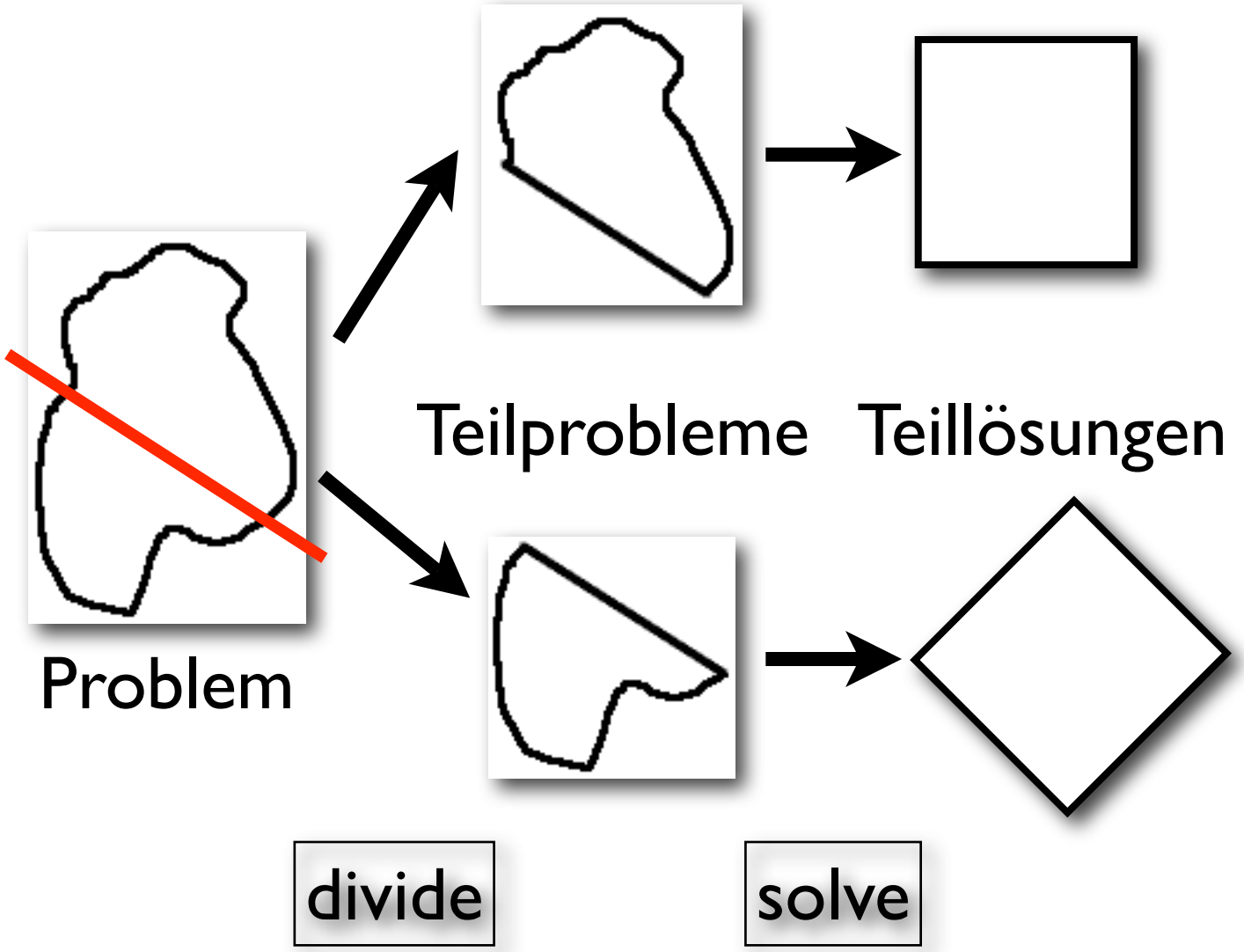
Algorithmenmuster

Divide and Conquer - Schema



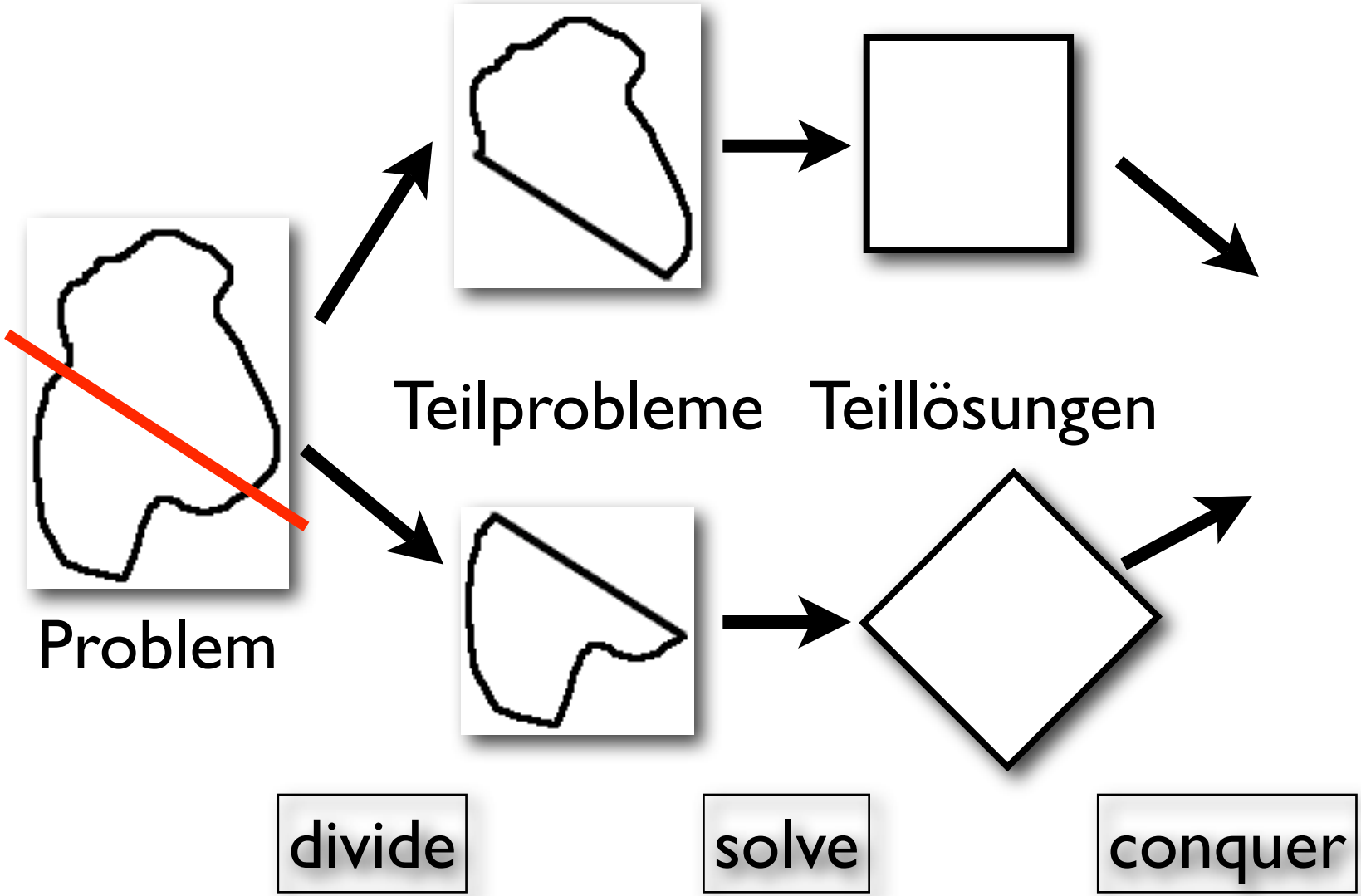
Algorithmenmuster

Divide and Conquer - Schema



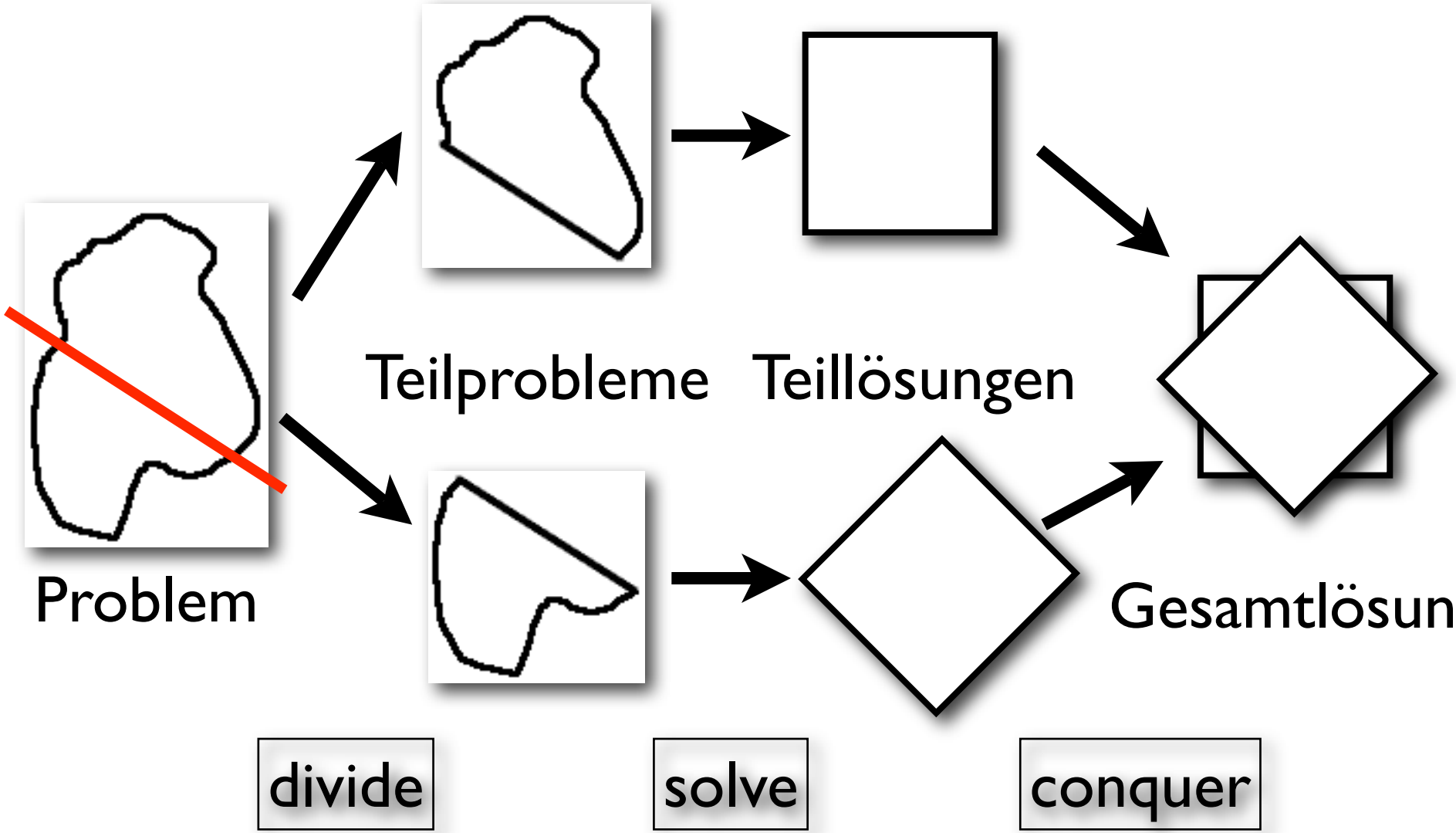
Algorithmenmuster

Divide and Conquer - Schema



Algorithmenmuster

Divide and Conquer - Schema



Algorithmenmuster
Divide and Conquer - Schema

- rekursives Muster:

  function D-C (P: Problem): Lösung;

  if "P trivial" then "Lösung für P" else

  Conquer(D-C(π_{12}(Divide(P)),

  D-C(π_{22}(Divide(P))).

Algorithmenmuster
Divide and Conquer - Komplexität

- Zwei Dimensionen relevant:
  - Komplexität für Divide und Conquer
  - Komplexität der Anzahl der D&C-Schritte

Algorithmenmuster
Divide and Conquer - Komplexität

- Komplexität für Divide und Conquer
  - konstante Laufzeit: O(1)
  - lineare Laufzeit: O(n)
  - höhere Laufzeiten selten
- Komplexität der Anzahl der D&C-Schritte
  - Abspalten eines konstanten Teilproblems: O(n)
  - Abspalten eines proportionalen Teilproblems: O(log n)

Algorithmenmuster
Divide and Conquer - Komplexität

	Schrittkomplexität O(1)	Schrittkomplexität O(n)
Zahl der Schritte O(log n)	O(log n)	O(n log n)
Zahl der Schritte O(n)	O(n)	$O(n^2)$

Algorithmenmuster
Rekursionsgleichungen

- Analyse der Komplexität von rekursiven Algorithmen führt auf Rekursionsgleichungen für Laufzeit und Speicher:
  - binäres Suchen: $T(n)=C+T(n/2)$
  - Multiplikation: $T(n)=4T(n/2)+O(n)$
- Wie löst man solche Gleichungen?

Algorithmenmuster
Rekursionsgleichungen - Lösungsschema

- Genauer:
  - g gegeben
  - f(n)=f(g(m)), m<n
  - Anfangswert, z.B.: f(1)=a
- Gesucht: geschlossene Darstellung für f
- Methoden:
  - Substitutionsmethode
  - Iterationsmethode
  - Mastermethode

Algorithmenmuster
Rekursionsgleichungen - Substitution

- Bilde eine Vermutung für die Lösung
- Setze Lösung in Gleichung ein und verifiziere

- Beispiel: $T(n)=2\,T(\lfloor n/2 \rfloor)+n, T(1)=1$
- Vermute: $T(n)=O(n \log n)$

Algorithmenmuster
Substitution - Beispiel

Vermute: $T(n)=O(n \log n)$

Zeige per Induktion: $T(n) \leq cn\log_2 n$ für ein $c>0$.

Induktionsanfang: $T(2)=4 \leq 2c\log_2 2$ für $c \geq 2$

Induktionsvor.: Sei Beh. korrekt für $n-1 \geq 1$.

Induktionsbeh.: Dann ist die Beh. auch korrekt für n.

Algorithmenmuster
Substitution - Beispiel

Induktionsbeh.: Dann ist die Beh. auch korrekt für n.

$T(n)=2T(\lfloor n/2 \rfloor)+n$

$\leq 2(c \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor)+n$

$\leq cn \log_2 \lfloor n/2 \rfloor +n$

$=cn (\log_2 n-\log_2 2)+n$

$=cn \log_2 n-cn+n$

$\leq cn \log_2 n$ für $c \geq 1$.

Algorithmenmuster
Rekursionsgleichungen - Iteration

- Wickle die Rekursion durch fortlaufende Einsetzung ab
- Schätze die entstehende Reihe durch einen geschlossenen Ausdruck ab

- Beispiele:
  - binäres Suchen
  - Multiplikation
  - Matrixmultiplikation

Algorithmen auf Zahlen
Matrixmultiplikation - Iteration

$T(n)=7T(n/2)+9n^2/2$

$=7\cdot(7T(n/4)+(9/2)(n/2)^2)+9n^2/2$

$=7^2\,T(n/4)+7\cdot(9/2)(n/2)^2)+9n^2/2$

Nach k Iterationen erhält man:

$=7^k\,T(n/2^k)+\Sigma_{i=0}^{k-1}\ 7^i\cdot(9/2)(n/2^i)^2$

mit k=log n erhält man

$=7^{\log n}\,T(1)+\Sigma_{i=0}^{\log n-1}\ \ (7/4)^i$

$=2^{\log 7 \log n}+(9/2)n^2\ \Sigma_{i=0}^{\log n-1}\ \ (7/4)^i$

$=n^{\log 7}+(9/2)n^2\ (((7/4)^{\log n}-1)/(7/4-1)$

$\leq n^{\log 7}+(9/2)n^2\ (4/3)\ n^{\log 7-\log 4}$

$\leq\ n^{\log 7}+6\ n^{\log 7}$

$=7\ n^{\log 7}\ \approx\ \ 7n^{2.81}$

Algorithmenmuster
Rekursionsgleichung - Mastermethode

- Löse Klasse von Rekursionsgleichungen in grundsätzlicher Weise:

$$T(n)=a\cdot T(n/b)+f(n),\ a\geq 1,\ b>1$$

- hier nur Ergebnis; Beweis im Buch von Mehlhorn

Algorithmenmuster
Mastermethode - Fall 1

Satz (Master-Theorem-1)

Seien $a \geq 1$, $b > 1$ Konstanten, $f: \mathbb{N} \to \mathbb{N}$ sei eine Funktion, $T: \mathbb{N} \to \mathbb{N}$ sei definiert durch:

$$T(n) = a \cdot T(n/b) + f(n)$$

Dann kann $T(n)$ folgendermaßen asymptotisch begrenzt werden:

Fall 1: Falls $f(n) = O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$, dann gilt:

$$T(n) = O(n^{\log_b a})$$

Algorithmenmuster
Mastermethode - Fall 2

Satz (Master-Theorem-2)

Seien $a \geq 1$, $b > 1$ Konstanten, $f: \mathbb{N} \to \mathbb{N}$ sei eine Funktion, $T: \mathbb{N} \to \mathbb{N}$ sei definiert durch:

$$T(n) = a \cdot T(n/b) + f(n)$$

Dann kann $T(n)$ folgendermaßen asymptotisch begrenzt werden:

Fall 2: Falls $f(n) = O(n^{\log_b a})$, dann gilt:

$$T(n) = O(n^{\log_b a} \log_2 n)$$

Algorithmenmuster
Mastermethode - Fall 3

Satz (Master-Theorem-3)

Seien $a \geq 1$, $b > 1$ Konstanten, $f: \mathbb{N} \to \mathbb{N}$ sei eine Funktion, $T: \mathbb{N} \to \mathbb{N}$ sei definiert durch:

$$T(n) = a \cdot T(n/b) + f(n)$$

Dann kann $T(n)$ folgendermaßen asymptotisch begrenzt werden:

Fall 3: Falls $f(n) \geq O(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$ und falls $a \cdot f(n/b) \leq c \cdot f(n)$ für ein $c > 1$ und alle $n \geq n_0$, dann gilt:

$$T(n) = O(f(n))$$

Algorithmenmuster
Mastermethode - Beispiel 1

- $T(n)=9T(n/3)+n$
- $a=9, b=3$
- $f(n)=n=O(n^{\log_b a-\varepsilon})=O(n^{\log_3 9-\varepsilon})=O(n^{2-\varepsilon})$, wobei $\varepsilon=1$
- Fall 1: $T(n)=O(n^2)$

Algorithmenmuster
Mastermethode - Beispiel 2

- $T(n)=T(2n/3)+1$
- $a=1, b=3/2$
- $f(n)=1=O(n^{\log_b a})=O(n^{\log_{3/2} 1})=O(n^0)=O(1)$
- Fall 2: $T(n)=O(\log_2 n)$

Algorithmenmuster
Mastermethode - Beispiel 3

- $T(n)=2T(n/2)+n \log_2 n$
- $a=2, b=2$
- $f(n)=n \log_2 n=O(n^{\log_b a+\varepsilon})=O(n^{\log_2 2+\varepsilon}) \overset{!}{\geq} O(n^{1+\varepsilon})$ mit $\varepsilon>0$
- Es gibt kein $\varepsilon>0$, so daß $f(n) \geq O(n^{1+\varepsilon})$, da das Polynom schneller wächst
- Fall 3 ist nicht anwendbar.
- Die anderen Fälle auch nicht.
- Master-Theorem hilft nicht.