5 Entwurfsmethoden für Algorithmen

1. Informatik: Eine Übersicht
2. Objektorientierte Modellierung
3. Logik (Einführung / Grundlagen)
4. Algorithmen
5. Entwurfsmethoden von Algorithmen
6. Sortieralgorithmen
7. Dynamische Datenstrukturen

©AIFB

5 Entwurfsmethoden für Algorithmen

5.1 Entwurfsmethoden für Algorithmen
- 5.1.1 Vorbemerkung

5.2 Entwurfsprinzipien
- 5.2.1 Schrittweise Verfeinerung
- 5.2.2 Modularisierung

5.3 Entwurfstechniken
- 5.3.1 Systematisches Probieren und Backtracking
- 5.3.2 Divide and Conquer
- 5.3.3 Problemtransformation

AIFB

©AIFB

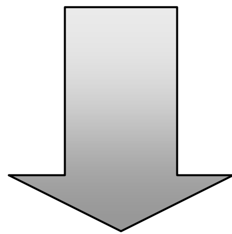
5.1.1 Vorbemerkungen

Bisherige Betrachtungen: kurze und einfach strukturierte Algorithmen

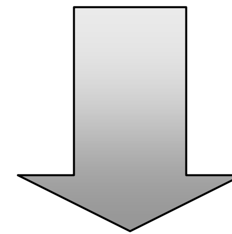
➡ *Entwurf solcher Algorithmen ist relativ problemlos!*

In der **Praxis:**

- oft sehr umfangreiche und wenig überschaubare Algorithmen
- komplizierter Entwurf



Systematisierung des Entwurfs



Angabe von Regeln und Techniken als Entwurfshilfen

AIFB

©AIFB

5.1.1 Vorbemerkungen

Algorithmenentwurf (allgemeiner: Softwareentwurf):

- Teilgebiet des Software-Engineering. Entwurfs- und Implementierungsphase. Vgl. Kapitel 3.
- Ziel: Systematisierung / Vereinheitlichung des Entwurfs.
- Heute teilweise schon computergestützt (Softwareentwicklungstools, Softwareentwicklungssysteme).

Forderungen:

Systematischer und nachvollziehbarer Entwurf
• ermöglicht systematisches Testen / Verifikation • Erleichterung der Pflege/Wartung/Erweiterung des Programms

arbeitsteiliger Entwurf (bei großen Problemstellungen)
• frühzeitige Strukturierung • Zerlegung in Teilprobleme

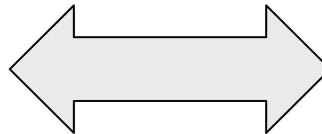
Effizienz des entworfenen Algorithmus
• Zeitkomplexität • Speicherkomplexität

AIFB

©AIFB

5.1.1 Vorbemerkungen

Wir unterscheiden:

Entwurfsprinzipien:		**Entwurfstechniken:**
allgemeingültige und anerkannte Konzepte und Richtlinien des Entwurfs, universell anwendbar.		im Speziellen anwendbare Verfahren und Vorgehensweisen für den Entwurf.

AIFB

©AIFB

5 Entwurfsmethoden für Algorithmen

5.1 Entwurfsmethoden für Algorithmen
- 5.1.1 Vorbemerkung

5.2 Entwurfsprinzipien
- 5.2.1 Schrittweise Verfeinerung
- 5.2.2 Modularisierung

5.3 Entwurfstechniken
- 5.3.1 Systematisches Probieren und Backtracking
- 5.3.2 Divide and Conquer
- 5.3.3 Problemtransformation

AIFB

©AIFB

5.2 Entwurfsprinzipien

- Allgemeingültige Prinzipien und Konzepte für Algorithmenentwurf
- Unterstützen ingenieurmäßige Erstellung von "guter" Software

Die wichtigsten Konzepte sind:

  - **schrittweise Verfeinerung**
  - **Modularisierung**

AIFB

©AIFB

5.2.1 Schrittweise Verfeinerung

Die elementaren Operationen, die zur Beschreibung von Algorithmen zur Verfügung stehen, sind i.a. sehr einfach.

Wenn man einen ***Algorithmus als eine komplexe Operation*** auffasst, die sich aus vielen elementaren Operationen und Ablaufstrukturen zusammensetzt, so ergibt sich daraus folgende Frage:

> *Wie gestaltet man den Schritt von der komplexen Operation zu den elementaren Operationen* ***möglichst überschaubar und einfach*** *?*

©AIFB

5.2.1 Schrittweise Verfeinerung

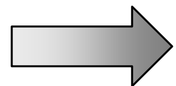
©AIFB

Prinzip der schrittweisen Verfeinerung:

1. Entwurf eines groben Algorithmus mit abstrakten Operationen und Datentypen.

2. Verfeinerung der im ersten Schritt benutzten Operationen, d.h. "Implementation" mit weniger abstrakten Operationen und Datentypen.

3. Wiederholung des Verfeinerungsschrittes, bis man einen Algorithmus erhält, der nur zur Verfügung stehende Ablaufstrukturen, Elementaroperationen und Datentypen enthält.



Top-Down-Entwurf

5.2.1 Schrittweise Verfeinerung

■ Beispiel **Sortieren durch direktes Einfügen**

Problem:

Eine vorgegebene Folge $F = (k_1,k_2,...,k_n)$ von Schlüsselwerten ist entsprechend einer Ordnung < anzuordnen.

Vereinbarung:

a[*i*] ::= Folgenelement an Position *i*
(nicht die Nummerierung des Schlüsselwerts).

Version 1:

```
Sortiere (k1,...,kn);
```

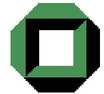
Version 2:

```
FOR i := 2 TO n DO
    "füge i-tes Element a[i]
    in (a[1],...,a[i-1]) an der
    richtigen Stelle ein"
END (* FOR *);
```

AIFB

©AIFB

5.2.1 Schrittweise Verfeinerung



AIFB

Version 3:

```
FOR i := 2 TO n DO
    j := i - 1;
    merke den Wert von a[i]
    WHILE (j ≠ 0) AND (a[i] < a[j]) DO
        verschiebe a[j] eine Stelle nach rechts
        j := j - 1
    END (* WHILE *);
    "füge den gemerkten Wert an (j +1)-ter Stelle ein"
END (* FOR *);
```

Version 4:

```
FOR i := 2 TO n DO
    j := i - 1;
    k := a[i];
    WHILE (j ≠ 0) AND (k < a[j]) DO
        a[j +1] := a[j];
        j := j - 1
    END (* WHILE *);
    a[j +1] := k;
END (* FOR *);
```

©AIFB

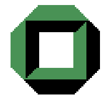
5.2.1 Schrittweise Verfeinerung

Bemerkung:

Top-Down-Entwurf erfordert, dass simultane Verfeinerung / Konkretisierung von

- Operationen,
- Kontrollflüssen (Ablaufstrukturen)
- Datenstrukturen

aufeinander abgestimmt werden, d.h. die einzelnen Bestandteile von Algorithmen sollten auf gleichem Niveau verfeinert werden.



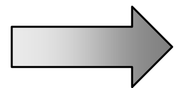
AIFB

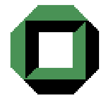
©AIFB

5.2.2 Modularisierung

Modularisierung: Zerlegung von Problemen in Teilprobleme, die

- klar voneinander abgegrenzt sind,
- getrennt bearbeitet werden können,
- weitgehend unabhängig voneinander sind,
- deren Lösungen nebenwirkungsfrei gegen Alternativlösungen austauschbar sind (keine "Seiteneffekte").

 Einzelne Lösungsbausteine/Teillösungen: **Module**



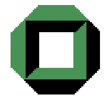
AIFB

©AIFB

5.2.2 Modularisierung

Charakteristika:

- Keine Berücksichtigung von Reihenfolge-Festlegungen (im Unterschied zur schrittweisen Verfeinerung)
- Spezifikation weitgehend unabhängiger Teilaufgaben mit klar definierten Schnittstellen
- Späteres Zusammenfügen der einzelnen Teillösungen über die vereinbarten Schnittstellen



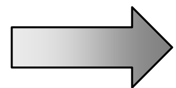
AIFB

©AIFB

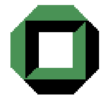
5.2.2 Modularisierung

Vorteile der Modularisierung:

- Reduzierung der Komplexität eines Problems durch Zerlegung in überschaubare Einzelprobleme
- Einzelne Module können unabhängig voneinander getestet oder verifiziert werden
- Einzelne Module sind austauschbar und erweiterbar
- Erstellung von Algorithmen und Software erfolgt nach dem "Baukastenprinzip"



Modularisierung erlaubt einen unproblematischen und zügigen Übergang zu Nachfolgeversionen eines Softwareprodukts (Austausch einzelner Module).



AIFB

©AIFB

5.2.2 Modularisierung

Möglichkeiten der **Moduleinteilung**:

- **problemorientierte Modularisierung**:
  Einteilung in geschlossene Verarbeitungseinheiten,
  (z.B. zeitliche Abfolge der Bearbeitung eines Problems)
- **datenorientierte Modularisierung**
  Einteilung, die sich an der Bearbeitung bestimmter Daten orientiert.
- **funktionsorientierte Modularisierung**:
  Einteilung, die Algorithmen mit ähnlicher Funktionalität zusammenfasst.

AIFB

©AIFB

5 Entwurfsmethoden für Algorithmen

5.1 Entwurfsmethoden für Algorithmen
    5.1.1 Vorbemerkung

5.2 Entwurfsprinzipien
    5.2.1 Schrittweise Verfeinerung
    5.2.2 Modularisierung

5.3 Entwurfstechniken
    5.3.1 Systematisches Probieren und Backtracking
    5.3.2 Divide and Conquer
    5.3.3 Problemtransformation

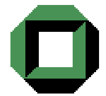
©AIFB

5.3 Entwurfstechniken

Entwurfstechniken:
umfassen Lösungswege und -konzepte, die für bestimmte Problemstellungen geeignet sind.

Solche Techniken sind:

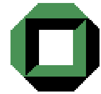
- **systematisches Probieren** und **Backtracking**
- **Divide and Conquer**
- **Transformation** von Problemen



AIFB

©AIFB

5.3.1 Systematisches Probieren und Backtracking



©AIFB

Systematisches Probieren:

Annahme:
• Kenntnis / Erzeugbarkeit der **möglichen Ausgabewerte** (Lösungsraum eventuell abhängig von konkreter Eingabe), welche die **gültigen Ausgabewerte** enthalten.

Vorgehensweise:
• Systematisch alle möglichen Ausgabewerte aufzählen; bei jedem Wert prüfen, ob er ein gültiger Ausgabewert ist.

z. B. Suchen in einer Folge
Lösungsraum: alle Folgenelemente

5.3.1 Systematisches Probieren und Backtracking

©AIFB

Rückverfolgung (backtracking)

Vorgehen:

- Backtracking (Trial-and-Error-Verfahren) ist eine **systematische Suche** in einem vorgegebenen Lösungsraum.
- Führt eine Teillösung in eine **Sackgasse**, dann wird der jeweils letzte Schritt rückgängig gemacht.
- Die dann erhaltene reduzierte Teillösung versucht man auf einem anderen Weg wieder zu einer **Gesamtlösung** auszubauen.

5.3.1 Systematisches Probieren und Backtracking

Backtracking: Realisierung

Vorstellung:
• gesamter Lösungsraum als Baum angeordnet

Vorgehensweise:

- Backtracking-Algorithmus durchläuft diesen Baum in bestimmter Reihenfolge;
  grob skizziert:
  - Ausgehend von Knoten des Baumes, der aktuelle Teillösung repräsentiert:
    prüfe, ob Nachfolgeknoten zu Erfolg führt.
  - falls nicht: prüfe anderen Nachfolgeknoten.
  - falls kein Nachfolgeknoten zu Erfolg führt, gehe zurück zum Vorgänger des aktuellen Knotens
  - Prüfung erfolgt rekursiv

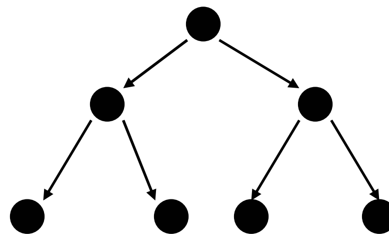
AIFB

©AIFB

5.3.1 Systematisches Probieren und Backtracking

Informelle Definition von einem Baum

- Ein Baum besteht aus einer Menge von Knoten und einer Menge von gerichteten Kanten. Knoten werden durch Punkte od. Kreise und Kanten durch Pfeile zwischen den Knoten dargestellt.
- Knoten **a** heißt Vaterknoten von Knoten **b**, wenn es eine Kante von **a** nach **b** gibt.
- Es gibt genau einen Knoten, der keinen Vatenknoten hat. Dieser Knoten wird auch Wurzel genannt. Alle anderen Knoten haben genau einen Vaterknoten.



AIFB ©AIFB

5.3.1 Systematisches Probieren und Backtracking

- Beispiel **Rucksack- Problem**

Gegeben: *n* Gegenstände mit Gewicht $x_1, ..., x_n$

Gesucht: Teilmenge (wird in den Rucksack gepackt), so dass das Gewicht aller enthaltenen Elemente maximal ist und kleiner gleich einer oberen Schranke *y*.

Spezifikation:

Eingabe: $x_1, ..., x_n$, mit $x_i > 0$ für jedes $i \in \{1, ..., n\}$;
y, mit $y > 0$

Ausgabe: Menge $M \subseteq \{x_1, ..., x_n\}$, für die gilt:

$$\sum_{x \in M} x \leq y \quad \text{und} \quad \sum_{x \in M} x \quad \text{ist maximal.}$$

AIFB

©AIFB

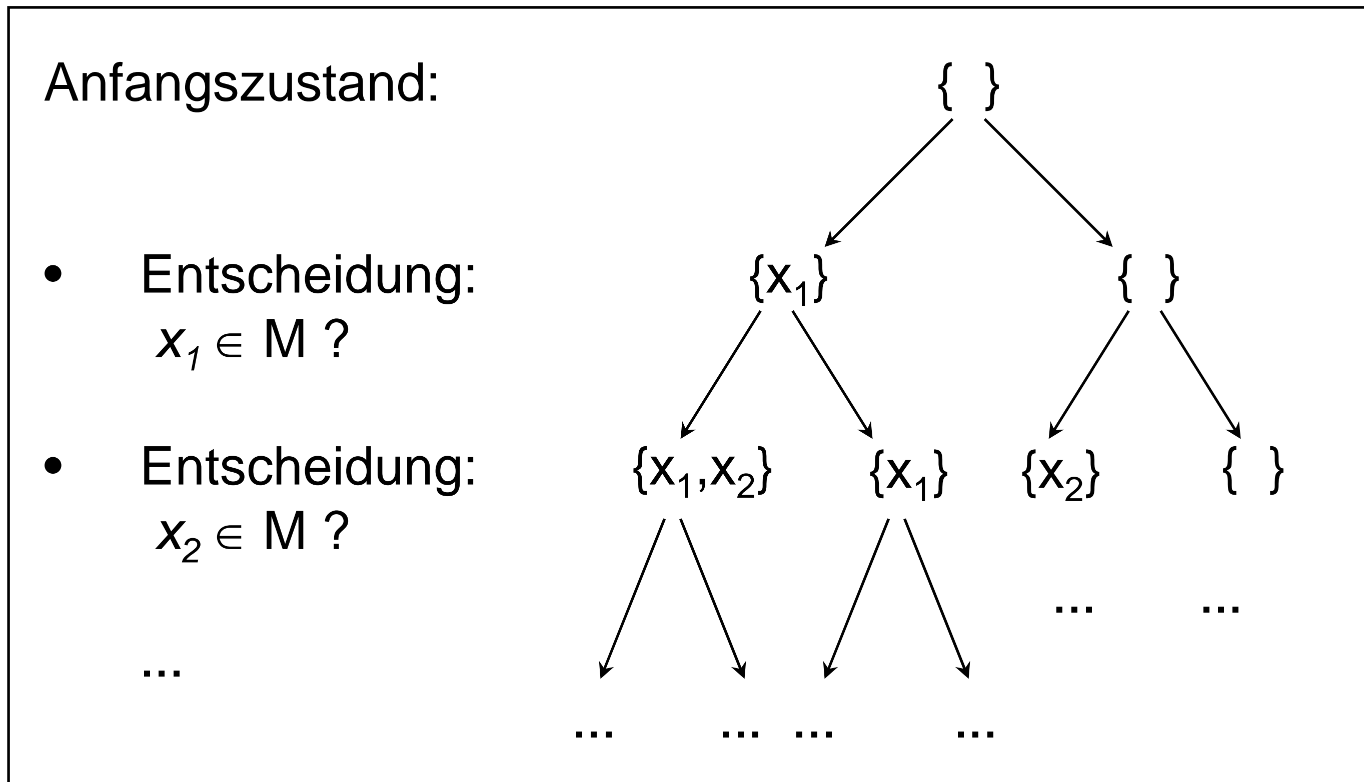
5.3.1 Systematisches Probieren und Backtracking

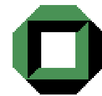
Lösung durch **Backtracking:**

Entscheidungsbaum:

(veranschaulicht sämtliche Teilmengen von $\{x_1, ..., x_n\}$.

$\Rightarrow$ 2^n Möglichkeiten (= IP $(\{x_1, ..., x_n\})$)





AIFB

©AIFB

5.3.1 Systematisches Probieren und Backtracking

Vereinbarungen:

- Als Variablen werden benutzt:

M:	aktuelle Menge ausgewählter Zahlen aus $\{x_1, ..., x_n\}$,
M_{opt}:	enthält zu jedem Zeitpunkt die bisher optimale Menge.

- Gewicht(M):
  Der Ausdruck "*prüfe Optimalität*" im Algorithmus bedeutet:

IF ($Gewicht(M) \leq y$ und $Gewicht(M) > Gewicht(M_{opt})$)

THEN $M_{opt} := M$;

AIFB

©AIFB

5.3.1 Systematisches Probieren und Backtracking

```
ALGORITHMUS Rucksack1(i);
BEGIN
    (* x_i ∉ M *)
    IF i < n THEN
            Rucksack1(i + 1)
    ELSE "prüfe Optimalität"
    END (* IF *);
    M := M + {x_i};
    (* x_i ∈ M *)
    IF i < n THEN
            Rucksack1(i + 1)
    ELSE "prüfe Optimalität"
    END (* IF *);
    M := M - {x_i};
END.
```

Vor dem ersten Aufruf des Algorithmus:
Einlesen von $x_1, ..., x_n$ und y und Initialisieren von M und M_{opt} mit der leeren Menge.

Erster Aufruf mit Rucksack1(1).

AIFB
©AIFB

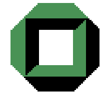
5.3.1 Systematisches Probieren und Backtracking

©AIFB

Beobachtungen:

- Der Algorithmus prüft **alle** 2^n Pfade des Entscheidungsbaumes
- Wenn er an einem Blatt angekommen ist, wird geprüft, ob die dort stehende Menge "besser" ist als das aktuell angenommene Optimum M_{opt}.

5.3.1 Systematisches Probieren und Backtracking



AIFB

©AIFB

Bemerkungen:

- Backtracking-Algorithmen sind im allgemeinen von exponentieller Zeitkomplexität.
- Denkbar sind auch Entscheidungsbäume mit mehr als 2 Nach-folgern in den inneren Knoten.
- Andere Problemstellungen, die man mit Backtracking bearbeiten kann, sind etwa:

→ **Schachprobleme:**

8-Damen-Problem:

Es sind 8 Damen auf einem Schachbrett so zu plazieren, dass sich je 2 nicht wechselseitig bedrohen.

→ **Graphenprobleme:**

Hamiltonscher Zyklus (Rundreiseproblem)

5.3.2 Divide and Conquer

Grundidee:
• Zerlegung eines Problems in kleinere Teilprobleme

Vorgehensweise:
• Zerlege P in mehrere kleinere Teilprobleme $P_1, ..., P_k$ derselben Art.
• Löse alle Teilprobleme P_i, $1 \leq i \leq k$.
• Setze die k Teillösungen von $P_1, ..., P_k$ zu einer Gesamtlösung zusammen.

Im allg.:
Rekursive Anwendung des Verfahrens, bis hinreichend kleine Problemgröße erreicht ist.

Dann:
Abbruch der Rekursion und „direkte“ Lösung der einzelnen Teilprobleme.

AIFB
©AIFB

5.3.2 Divide and Conquer

- Beispiel **Bestimmung des Maximums**

Spezifikation:

Eingabe: $M \subseteq$ IN, mit $|M| = n \in$ IN

Ausgabe: $m \in M$, mit $m >= x$ für alle $x \in M$

```
ALGORITHMUS Maximum (M);
BEGIN (* Zerlegung: *)
    IF |M| = 1 THEN
                    RETURN m ∈ M (*Direkte Lösung*)
    ELSE "Zerlege M in gleich große (± 1 Element),
            disjunkte Mengen M1, M2";
            m1 := Maximum(M1);
            m2 := Maximum(M2);
            (* Zusammensetzen: *)
        IF m1 > m2 THEN
                    RETURN m1
        ELSE RETURN m2
        END; (* IF *)
    END; (* IF *)
END.
```

©AIFB

5.3.2 Divide and Conquer

Komplexität:

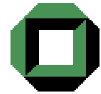
T(Maximum; n) = 2 * T(Maximum; n/2) + c

c : konstanter Aufwand (unabhängig von n) für

- die Zerlegung in Teilprobleme
- das Zusammenfassen der Teillösungen.

Auflösung der Gleichung: T(Maximum; n) $\in$ O(n)

*Divide-and-Conquer-Technik bewirkt eine **Effizienzverbesserung**, wenn der Zeitaufwand für das Lösen der Einzelprobleme und für das Zusammensetzen der Teillösungen kleiner ist als für die Lösung des Gesamtproblems!*



AIFB

©AIFB

5.3.2 Divide and Conquer

Komplexitätsbetrachtungen (Divide and Conquer):

- Gegeben: Problem P der Größe $n \in$ IN
- A sei ein rekursiver Algorithmus, der P im folgenden Sinne löst:

n = 1:

- Direkte Lösung mit Algorithmus B. Dabei gelte: $T(B; 1) = a > 0$

n > 1:

- Zerlegung von P in $k > 1$Teilprobleme P_i (derselben Art)
  Größe der Teilprobleme jeweils: $n_i = n/c$
- Lösung aller P_i mit Algorithmus A
- Zusammensetzen der Teillösungen zu einer Gesamtlösung
- d(n): Aufwand für die Zerlegung des Problems und das
  Zusammensetzen der Teillösungen

AIFB

©AIFB

5.3.2 Divide and Conquer

Komplexitätsbetrachtungen (Divide and Conquer):

$$T(A; n) = \begin{cases} a, & \text{falls } n = 1 \\ k * T(A; n/c) + d(n), & \text{falls } n > 1 \end{cases}$$

Für $n = c^m$ folgt durch vollständige Induktion über m:

$$T(A; n) = T(A; c^m) = a * k^m + \sum_{j=0}^{m-1} \left(k^j * d\left(c^{m-j}\right)\right)$$

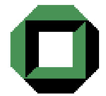
©AIFB

5.3.2 Divide and Conquer

Zusammenfassung:

Die Zeitkomplexität von Divide-and-Conquer-Algorithmen ist abhängig von:

- Anzahl *k* der Teilprobleme bei der Zerlegung,
- Größe *n/c* der einzelnen Teilprobleme,
- Komplexität *d(n)* für die Zerlegung in Teilprobleme und das Zusammensetzen der Lösung,
- Komplexität *a* für die direkte Lösung bei Abbruch der Rekursion.



AIFB

©AIFB

5.3.3 Problemtransformation

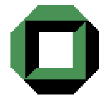
„Transformation eines Problems auf ein anderes, bereits gelöstes Problem ergibt oft einfache Lösung“.

Situation:

Gegeben: Problem P.
Gesucht: Algorithmus A, der P löst.
Bekannt: Algorithmus B, der ein zu P ähnliches / verwandtes Problem Q löst.

Vorgehensweise:

- Durchführung einer Transformation $P \rightarrow Q$ (Umformulierung).
- Entwurf eines Algorithmus TPQ zur Transformation der Eingabedaten: $E(P) \rightarrow E(Q)$.
- Entwurf eines Algorithmus TQP zur Transformation der Ausgabedaten: $A(Q) \rightarrow A(P)$.

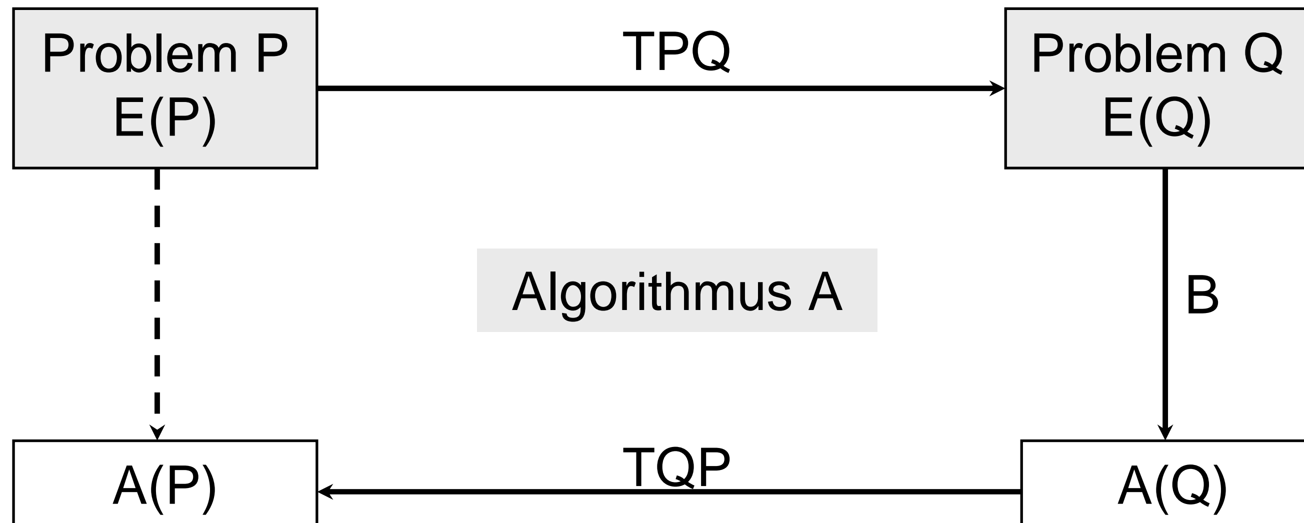


AIFB

©AIFB

5.3.3 Problemtransformation

Gesamtalgorithmus A arbeitet dann nach folgendem Schema:



- Anwendung von TPQ, d.h. Abbildung der Eingabedaten E(P) → E(Q);
- Anwendung von Algorithmus B
- Anwendung von TQP, d.h. Abbildung der Ausgabedaten A(Q) → A(P).

Komplexität: T(A; n) = T(TPQ; n) + T(B; n) + T(TQP, n)

AIFB

©AIFB