



Universität Bremen

Algorithmenkonstruktion

Thomas Röfer

Praktische Informatik 2

VAK: 03-700.02

Voraussetzung: 03-700.01 (PI-1)

ECTS: 6



Rückblick PI-1

- ▶ **Gesamt: 284 Studierende**
- ▶ **Übungen erfolgreich absolviert: 222 (78%)**
- ▶ **Fachgespräche bestanden: 197 (69%)**
- ▶ **Durchschnittsnote: 2,0**

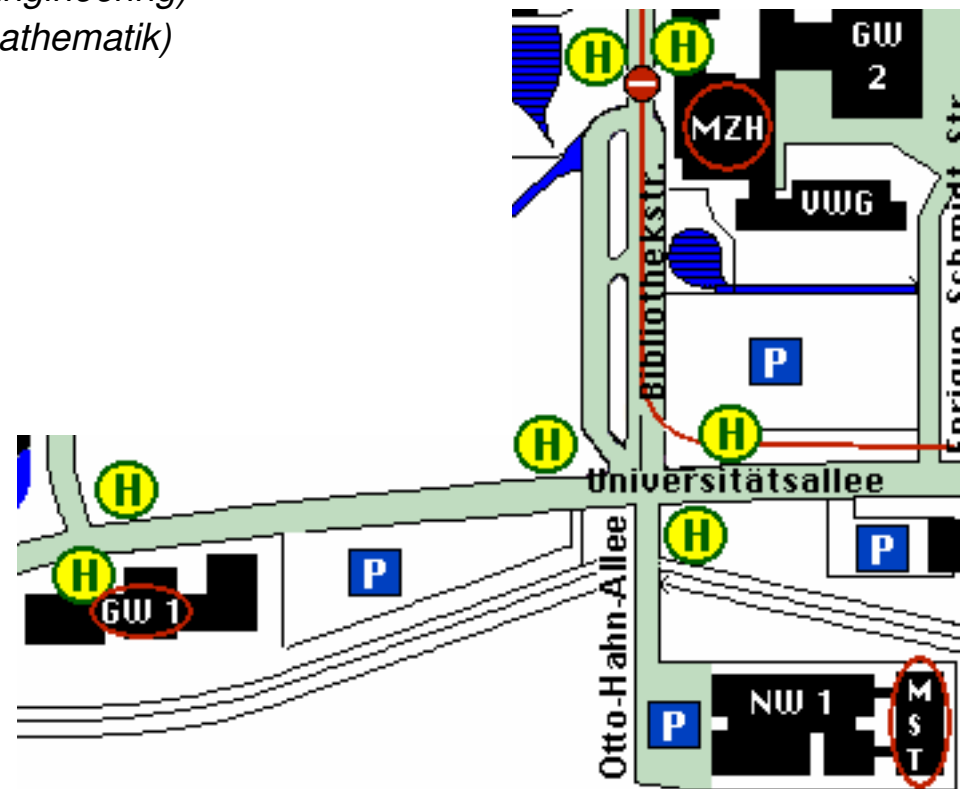
Organisatorisches

▶ Tutorien

- ▶ Mo, 17:00 – 19:00
 - ▶ MZH 7220, Kai Florian Richter (Systems Engineering)
 - ▶ MZH 4194, Jan Oliver Wallgrün (Technomathematik)
 - ▶ MZH 5210, Elena Svetogorova
- ▶ Di, 08:00 – 10:00
 - ▶ GW1 B2130, Thomas Wagner
 - ▶ MZH 4194, Achim Mahnke
- ▶ Di, 13:00 – 15:00
 - ▶ GW1 A0150, Zicheng Wang
- ▶ Mi, 10:00 – 12:00
 - ▶ MZH 7220, Arne Schuldt (Digitale Medien)
- ▶ Do, 10:00 – 12:00
 - ▶ MZH 7220, Elena Svetogorova
 - ▶ MZH 7250, Achim Mahnke

▶ Zuordnung

- ▶ Dreiergruppen





Literatur

▶ Die Bücher zur Vorlesung

- ▶ Wolfgang Küchlin, Andreas Weber: [Einführung in die Informatik - objektorientiert mit Java](#). 3. Auflage, Springer-Verlag 2005, ISBN 3-540-20958-1.
- ▶ Thomas Ottmann, Peter Widmayer: [Algorithmen und Datenstrukturen](#). 4. Aufl., Spektrum Akademischer Verlag, ISBN 3-8274-1029-0.

▶ Weitere Literatur

- ▶ Robert Sedgewick. [Algorithmen](#). 2. Auflage. Pearson Studium, 3-8273-7032-9.
- ▶ Robert Sedgewick. [Algorithmen in Java](#). Teil 1-4, 3., überarbeitete Auflage. Pearson Studium, 3-8273-7072-8.



Informationsquellen zu PI-2

▶ Website

- ▶ <http://www.informatik.uni-bremen.de/pi2> oder
- ▶ <http://www.tzi.de/pi2>
- ▶ Vorlesungsfolien
- ▶ Übungszettel, Musterlösungen
- ▶ Software, Dokumentenvorlagen

▶ Mail

- ▶ An mich: <mailto:Thomas.Roefer@dfki.de>
- ▶ An Tutoren: im Tutorium erfragen
- ▶ Quelltexte an Tutoren: BlueJ kann diese direkt verschicken

▶ Newsgroup

- ▶ <news://news.informatik.uni-bremen.de/fb3.lv.pi2>

Übungsblätter

▶ **Bearbeitung**

- ▶ In Gruppen aus jeweils drei Studierenden (im selben Tutorium!)

▶ **Ablauf**

- ▶ Übungsblätter stehen montags zur Vorlesung im Netz.
- ▶ Sie werden in den Tutorien besprochen.
- ▶ Sie werden am folgenden Montag in der Vorlesung abgegeben. Dazu werden Umschläge mit den Namen der Tutoren ausliegen.
- ▶ Quelltexte von Programmen werden per E-Mail an die Tutoren geschickt.
- ▶ Die korrigierten und bewerteten Übungsblätter werden im Tutorium eine Woche später zurückgegeben.
- ▶ Eine Musterlösung erscheint im Netz.



Erwerb eines SBLN

▶ Bedingungen für den Erhalt eines SBLN

- ▶ Ein Übungsblatt gilt als erfolgreich bearbeitet, wenn mindestens 40% der Punkte erreicht wurden
- ▶ $n-1$ Übungsblätter müssen erfolgreich bearbeitet werden ($n = 11$)
- ▶ Note ergibt sich aus den $n-1$ besten Übungsblättern

▶ Benotung

- | | | |
|--|-----------------------------|-----------------------------|
| ▶ | $\geq 95\% \rightarrow 1,0$ | $\geq 90\% \rightarrow 1,3$ |
| ▶ $\geq 85\% \rightarrow 1,7$ | $\geq 80\% \rightarrow 2,0$ | $\geq 75\% \rightarrow 2,3$ |
| ▶ $\geq 70\% \rightarrow 2,7$ | $\geq 65\% \rightarrow 3,0$ | $\geq 60\% \rightarrow 3,3$ |
| ▶ $\geq 55\% \rightarrow 3,7$ | $\geq 50\% \rightarrow 4,0$ | |
| ▶ $< 50\% \rightarrow$ nicht bestanden | | |

▶ Fachgespräch

- ▶ In den ersten beiden Woche der vorlesungsfreien Zeit
- ▶ Individualitätsprüfung
- ▶ Gruppenweise, 5-10 Min. pro Gruppenmitglied
- ▶ Veranstalter prüft, Tutoren sind Beisitzer
- ▶ Wiederholung gegen Ende der vorlesungsfreien Zeit



Erwerb eines SBLN

▶ **Täuschungsversuch**

- ▶ Abgegebener Lösungsvorschlag wurde zumindest teilweise ohne Quellenangabe abgeschrieben
 - ▶ *von Kommilitonen*
 - ▶ *aus dem Internet*
 - ▶ *aus einem Buch*
- ▶ Abgegebener Lösungsvorschlag wurde gemeinsam mit einer anderen Gruppe erstellt
- ▶ Ein Täuschungsversuch führt
 - ▶ *zum sofortigen Verlust des SBLN*
 - ▶ *zur Meldung beim Prüfungsamt*

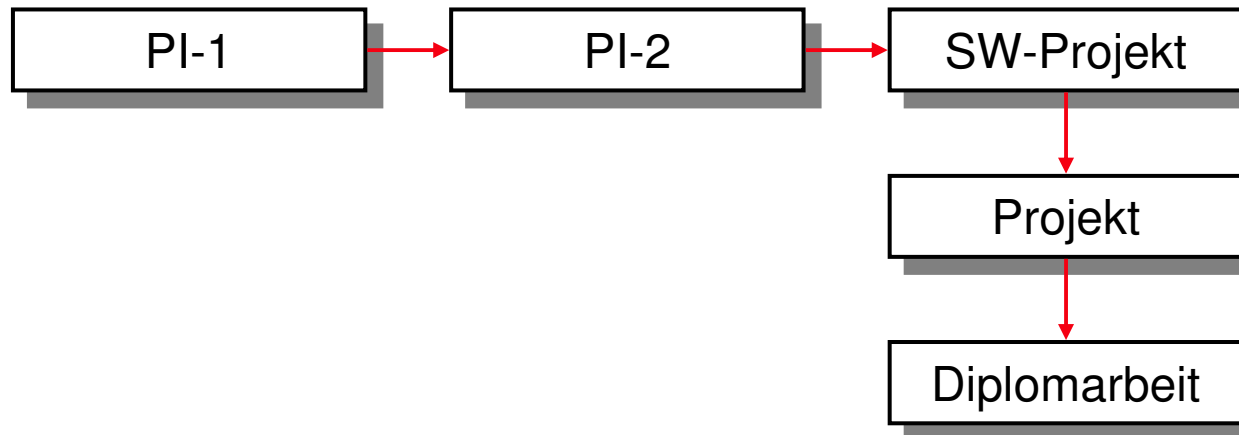
▶ **Ärztliches Attest wegen Krankheit**

- ▶ Bei verspäteter Abgabe von Übungsblättern
 - ▶ *Möglichst irgendwie ankündigen (wg. Musterlösung im Web)*
- ▶ Bei Nichterscheinen zu Fachgesprächen



Bedeutung von PI-2 für das Studium

- ▶ **Bestehen von PI-2 ist Voraussetzung für Teilnahme am Software-Projekt**



- ▶ **Durchfallen in PI-2 verlängert Studium um mindestens 1 Jahr!**
 - ▶ Zumindest für Informatik-Bachelor/Diplom-Studierende



Algorithmus – Definition

▶ Spezifikation

- ▶ *Eingabespezifikation*: Eingabegrößen, Anforderungen an diese
- ▶ *Ausgabespezifikation*: Ausgabegrößen, Eigenschaften dieser

▶ Durchführbarkeit

- ▶ *Endliche Beschreibung*: Verfahren muss in endlichem Text vollständig beschrieben sein
- ▶ *Effektivität*: jeder Schritt muss effektiv ausführbar sein
- ▶ *Determiniertheit*: Verfahrensablauf ist zu jedem Zeitpunkt fest vorgeschrieben

▶ Korrektheit

- ▶ *partielle Korrektheit*: jedes berechnete Ergebnis genügt der Ausgabespezifikation, sofern die Eingaben der Spezifikation genügt haben
- ▶ *Terminierung*: Halt nach endlich vielen Schritten mit einem Ergebnis, sofern die Eingaben der Spezifikation genügt haben



Grundschemata des Algorithmenaufbaus

- ▶ Name des Algorithmus und Liste der Parameter
- ▶ Spezifikationen des Ein-/Ausgabeverhaltens
- ▶ **Schritt 1: Vorbereitung**
 - ▶ Einführung von Hilfsgrößen etc.
- ▶ **Schritt 2: Trivialfall?**
 - ▶ Prüfe, ob ein einfacher Fall vorliegt. Falls ja, Beendigung mit Ergebnis.
- ▶ **Schritt 3: Arbeit (Problemreduktion, Ergebnisaufbau)**
 - ▶ Reduziere die Problemstellung X auf eine einfachere Form X' , mit $X > X'$ bezüglich einer wohlfundierten Ordnung $>$. Baue entsprechend der Reduktion einen Teil des Ergebnisses auf.
- ▶ **Schritt 4: Rekursion bzw. Iteration**
 - ▶ Rufe zur Weiterverarbeitung den Algorithmus mit dem reduzierten X' erneut auf (Rekursion), bzw. fahre mit X' bei Schritt 2 fort (Iteration).



Beweis rekursiver Algorithmen

▶ Beweisverfahren

- ▶ Rekursive Algorithmen können durch vollständige Induktion bewiesen werden

▶ Beweis

- ▶ Behauptung: Die Funktion

$$factorial(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot factorial(n-1) & \text{sonst} \end{cases}$$

berechnet die Fakultät von n , d.h. $factorial(n) = n!$

▶ Induktionsanfang

- ▶ $0! = 1 = factorial(0)$

▶ Induktionsschritt

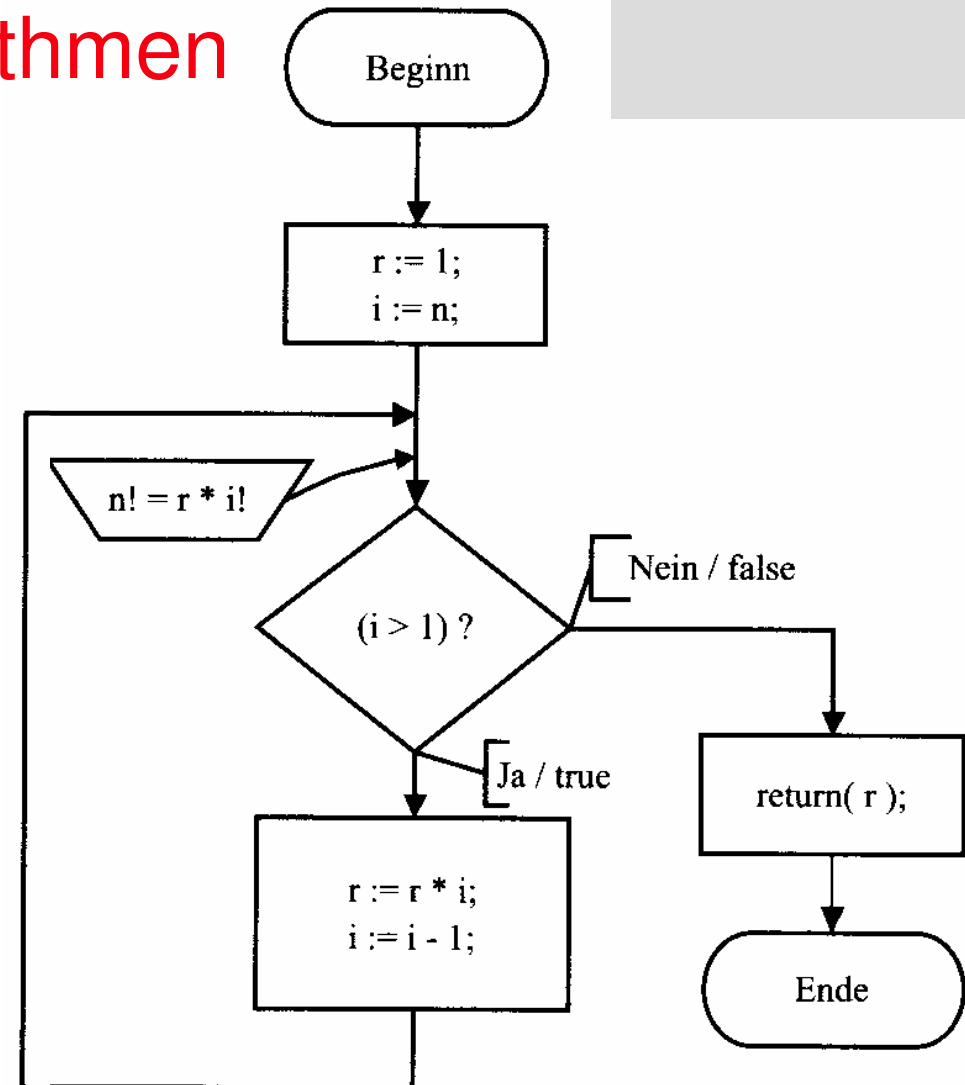
- ▶ $(n+1)! = factorial(n+1)$ || Def. von $factorial()$ einsetzen
- ▶ $= (n+1) \cdot factorial(n+1-1)$
- ▶ $= (n+1) \cdot factorial(n)$ || Induktionsannahme einsetzen
- ▶ $= (n+1) \cdot n!$
- ▶ $= (n+1)!$

q.e.d.

Beweis iterativer Algorithmen

► Verifikation nach Floyd

- Finde eine geeignete Formel $F(V)$ und zeige, dass sie eine Schleifeninvariante am Anfang der Schleife ist; bezeichne $F(V)$ mit $INV(V)$
- Zeige, dass aus der Eingabespezifikation folgt, dass $INV(V)$ vor dem ersten Schleifendurchgang gültig ist
- Zeige, dass nach dem letzten Schleifendurchgang aus $INV(V)$ und der unerfüllten Fortsetzungsbedingung der Schleife die Ausgabespezifikation folgt



Beweis iterativer Algorithmen

- ▶ **Schleifeninvariante**

- ▶ $n! = r \cdot i!$

- ▶ **Vor der Schleife**

- ▶ $n! = r \cdot i!$

- ▶ $= 1 \cdot n!$

- ▶ $= n!$

- ▶ **Invarianz in der Schleife**

Zu Beginn: $n! = r \cdot i!$

- ▶ $r' = r \cdot i, i' = i - 1$

Nach einem Durchlauf:

- ▶ $n! = r' \cdot i'!$

- ▶ $= (r \cdot i) \cdot (i - 1)!$

- ▶ $= r \cdot i \cdot (i - 1)!$

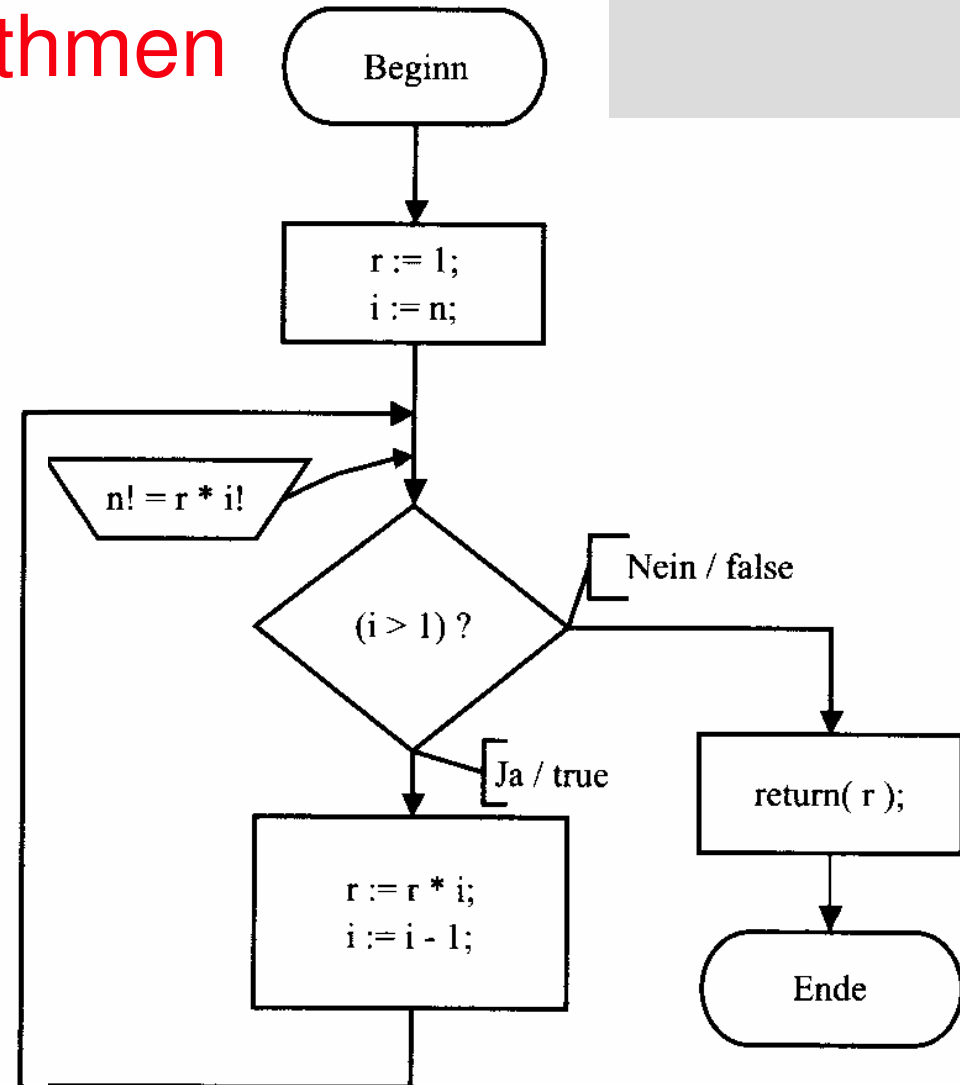
- ▶ $= r \cdot i!$

- ▶ **Nach der Schleife**

- ▶ $n! = r \cdot i!$

- ▶ $= r \cdot 1$

- ▶ $= r$





Algorithmenentwurf – Vorgehensweise

▶ **Bottom-up**

- ▶ Konstruktion von Objekten und Methoden immer höherer Abstraktionsstufe
- ▶ Fertig, wenn Methode gefunden, die das Gesamtproblem löst

▶ **Top-down**

- ▶ Zerlegung des Gesamtproblems in immer kleinere Teile immer geringerer Abstraktionsstufe
- ▶ Fertig, wenn sich alle Teile ohne weitere Zerlegung direkt programmieren lassen
- ▶ Definition von Schnittstellen beim Top-down-Entwurf ermöglicht Zusammenarbeit mehrerer Entwickler

▶ **In der Praxis**

- ▶ Grober Entwurf Top-down, Programmierung Bottom-up
- ▶ Je größer eine Problemstellung ist, desto wichtiger wird Top-down



Algorithmenentwurf – Ansätze

- ▶ **Greedy (gierig)**
 - ▶ Handle einfache und triviale Fälle
 - ▶ Reduziere das Problem in *einer* Richtung
 - ▶ Rekursiver Aufruf
- ▶ **Divide and conquer (Teile und herrsche)**
 - ▶ Handle einfache und triviale Fälle
 - ▶ Teile: Reduziere das Problem in *zwei* oder *mehrere* Teilprobleme
 - ▶ Herrsche: löse die Teilprobleme (typischerweise rekursiv)
 - ▶ Kombiniere: setze die Teillösungen zur Gesamtlösung zusammen
- ▶ **Dynamische Programmierung**
 - ▶ Werden Teillösungen mehrfach benötigt, rechne sie vorab aus und schreibe die Ergebnisse in eine Tabelle
 - ▶ Wird eine der Teillösungen benötigt, lies sie aus der Tabelle aus



Algorithmenentwurf – Beispiele

```
int greedySum(int[] a)
{
    return greedySum2(a, 0);
}

int greedySum2(int[] a, int i)
{
    if(i == a.length)
        return 0;
    else
        return a[i] + greedySum2(a, i + 1);
}
```

a

	0	1	2	3	4
	15	7	9	8	3

↓ ↓ ↓ ↓ ↓ ↓

```
int dacSum(int[] a)
{
    if(a.length == 0)
        return 0;
    else
        return dacSum2(a, 0, a.length);
}

int dacSum2(int[] a, int bottom, int top)
{
    if(bottom + 1 == top)
        return a[bottom];
    else
    {
        int middle = (bottom + top) / 2;
        return dacSum2(a, bottom, middle) +
            dacSum2(a, middle, top);
    }
}
```

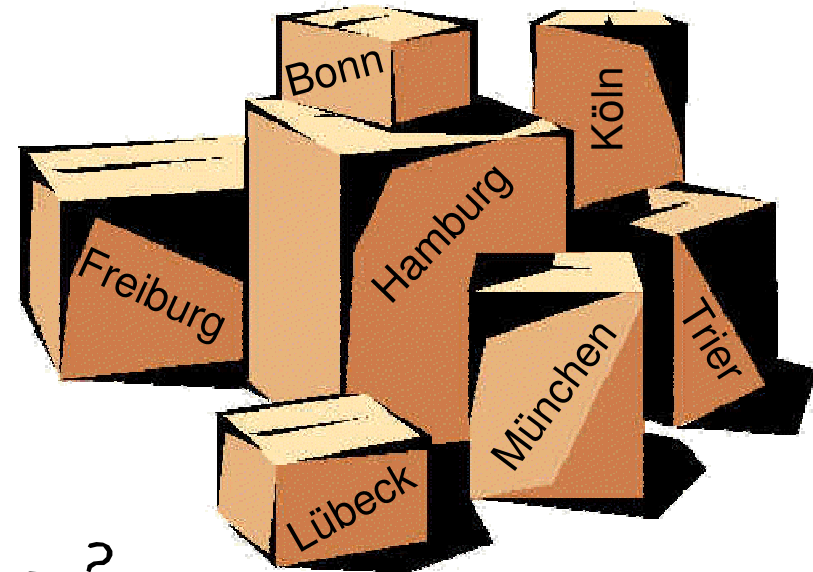
Beispiel: Rundreise-Problem

▶ Problemstellung

- ▶ Finde den kürzesten/schnellsten Weg, um ausgehend von einem Anfangsort alle gegebenen Orte genau einmal zu besuchen und zum Anfangsort zurückzukehren
- ▶ Auch bekannt als *Handlungsreisendenproblem* oder *Traveling Salesman Problem*

▶ Varianten

- ▶ Müllabfuhrproblem
- ▶ n -Rundreiseproblem (z.B. für Fahrzeugflotte)



Problemanalyse

▶ Rundreise

- ▶ Lässt sich durch eine Folge von Orten beschreiben
 - ▶ *Bremen, Hamburg, Lübeck, ...*
- ▶ Jeder Ort kommt nur einmal vor
- ▶ Vom letzten Ort in der Folge wird implizit eine Rückkehr zum ersten Ort angenommen
- ▶ Durch unterschiedliche Anordnung der Orte entstehen unterschiedliche Rundreisen

▶ Bewertung

- ▶ Eine Rundreise hat eine Gesamtlänge/-dauer
- ▶ Allgemein spricht man hierbei von *Kosten*
- ▶ Die Gesamtkosten ergeben sich aus der Summe der Einzelkosten, die durch das Fahren von einem zum jeweils nächsten Ort in der Rundreise entstehen
- ▶ Unterschiedliche Besuchsreihenfolgen der Orte erzeugen unterschiedliche Kosten



Problemzerlegung

▶ Grundidee

- ▶ Komplexe Probleme lassen sich einfacher lösen, wenn man sie in Teilprobleme zerlegt
- ▶ Die Teilprobleme können möglicherweise wieder zerlegt werden
- ▶ Im Idealfall sollte die Zerlegung soweit gehen, dass alle Teilprobleme in das anfangs beschriebene Schema zum Algorithmenaufbau passen

▶ Zerlegung für das Rundreiseproblem

- ▶ Erzeuge alle möglichen Rundreisen
 - ▶ *Erzeuge eine Rundreise*
- ▶ Bestimme für jede einzelne Rundreise die Kosten
 - ▶ *Bestimme die Kosten für die Fahrt zwischen zwei Orten*
- ▶ Liefere die Rundreise mit den geringsten Kosten als Ergebnis



Formalisierung

▶ Orte

- ▶ $O = \{Bremen, Hamburg, München, \dots\}$

▶ Rundreise

- ▶ $R = [r_1, r_2, r_3, \dots], r_i \in O, \#R = \#O, \forall_{o_i \in O} o_i \in R$

▶ Beispiel

- ▶ $[Bremen, Hamburg, München] \in R$
- ▶ $[Bremen, München, Hamburg] \in R$

▶ Kosten

- ▶ $k: O \times O \rightarrow \mathbb{N}$

▶ Gesamtkosten

- ▶ $gk: R \rightarrow \mathbb{N}$

- ▶ $gk(\dot{R}) = k(r_{\#R}, r_0) + \sum_{i=1}^{\#\dot{R}-1} k(r_i, r_{i+1}) \quad , r_i \in \dot{R}$



Kostenfunktion

	Aachen	Berlin	Dortmund	Dresden	Düsseldorf	Frankfurt / M	Hamburg	Hannover	Karlsruhe	Köln	Leipzig	Magdeburg	München	Nürnberg	Rostock	Saarbrücken	Stuttgart	Würzburg
Aachen		638	154	651	80	256	482	354	346	73	569	494	631	475	663	263	518	370
Berlin	638		492	193	556	545	286	285	673	575	184	153	585	438	223	723	632	495
Dortmund	154	492		507	68	224	349	210	358	95	428	350	617	428	520	321	420	338
Dresden	651	193	507		581	492	495	382	581	591	140	225	491	325	444	671	525	382
Düsseldorf	80	556	68	581		220	392	278	341	42	500	417	611	438	562	277	401	338
Frankfurt / M	256	545	224	492	220		512	361	132	191	405	444	412	228	680	190	201	128
Hamburg	482	286	349	495	392	512		152	631	370	391	270	781	612	133	688	658	507
Hannover	354	285	210	382	278	361	152		489	294	247	136	661	488	320	551	534	377
Karlsruhe	346	673	358	581	341	132	631	489		303	521	558	271	261	809	188	80	199
Köln	73	575	95	591	42	191	370	294	303		481	422	577	422	567	282	373	289
Leipzig	569	184	428	140	500	405	391	247	521	481		88	418	260	371	588	466	408
Magdeburg	494	153	350	225	417	444	270	136	558	422	88		511	349	321	606	559	449
München	631	585	617	491	611	412	781	661	271	577	418	511		159	781	421	212	291
Nürnberg	475	438	428	325	438	228	612	488	261	422	260	349	159		601	362	218	109
Rostock	663	223	520	444	562	680	133	320	809	567	371	321	781	601		851	812	694
Saarbrücken	263	723	321	671	277	190	688	551	188	282	588	606	421	362	851		213	314
Stuttgart	518	632	420	525	401	201	658	534	80	373	466	559	212	218	812	213		149
Würzburg	370	495	338	382	338	128	507	377	199	289	408	449	291	109	694	314	149	



Bildung aller möglichen Rundreisen

▶ **Kürzeste Rundreise**

- ▶ $R_{\min} = \dot{R} \in R, gk(\dot{R}) = \min_{\ddot{R} \in R} gk(\ddot{R})$

▶ **Bildung von Rundreisen**

- ▶ Bildung aller möglichen Reihenfolgen von Orten \rightarrow *Permutation*
- ▶ Erster Ort bleibt immer fest, nur die anderen müssen permutiert werden

▶ **Algorithmus**

- ▶ Vertausche den ersten Ort der Reihe nach mit jedem Ort in dem Rundweg (auch mit sich selbst)
- ▶ Für jede Vertauschung: Führe den Algorithmus ab dem nächsten Ort wieder aus
- ▶ Wenn nur noch ein Ort übrig ist, bewerte den Rundweg



Permutationen – Beispiel

