Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Verhaltenswissenschaft
Departement Informatik

# "Greedy" und "Teile und Herrsche"

## Ein Leitprogramm in Informatik

Zielgruppe: Gymnasium, FH
Bearbeitungsdauer: ca. 8 Stunden

Autorin: Judith Gull
Betreuer: Prof. J. Hromkovič

Fassung vom: 30. September 2005

**Dieses Leitprogramm wurde noch nie erprobt.**

# Einführung

In der Informatik können oft verschiedene Lösungsansätze auf einen gemeinsamen Nenner gebracht werden. In diesem Leitprogramm lernst du zwei Methoden kennen, mit denen man Algorithmen für unterschiedliche Probleme entwerfen kann.

Das Leitprogramm besteht aus zwei Teilen. Die ersten beiden Kapitel behandeln die sogenannte "Greedy-Methode". In den Kapiteln 3 und 4 wird die "Teile-und-Herrsche" Methode behandelt.

## Was kannst du nach diesem Leitprogramm?

Du kannst die obengenannten Methoden anwenden, um selbst Algorithmen zu entwerfen. Du kennst auch einige Beispiele für Probleme, die man damit gut lösen kann.

## Vorkenntnisse

### Programmieren

Um dieses Leitprogramm erfolgreich bearbeiten zu können brauchst du grundlegende Programmierkenntnisse.

Die Lösungen für die Programmieraufgaben sind in Java geschrieben. Bei einigen Aufgaben sollst du auch etwas in bestehendem Java Code ergänzen. Du sollst deshalb Variablen, Schleifen, Kontrollstrukturen und Arrays kennen und wissen wie man sie in Java benützt.

### Mathematik

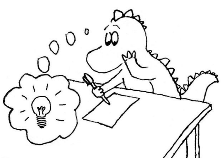
Du brauchst keine grossen Vorkenntnisse in der Mathematik. Du sollst aber etwa wissen, was eine Menge oder was eine Funktion ist.

# Arbeitsanleitung:

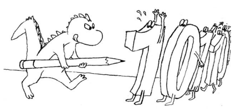

**Übersicht:** Am Anfang jedes Kapitels wird kurz beschrieben, um was es geht. Hier erfährst du auch, was du alles tun sollst im jeweiligen Kapitel.

**Lernziele:** Anschliessend werden die Lernziele bekannt gegeben. Hier steht beschrieben, was du in diesem Kapitel lernen sollst und was nachher im Test geprüft wird.

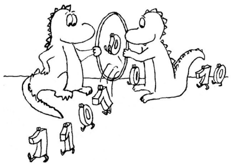
In jedem Kapitel wirst du zwischendurch gebeten, Aufgaben zu lösen:



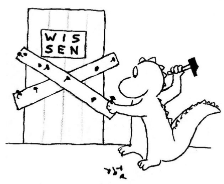
Aufgabe



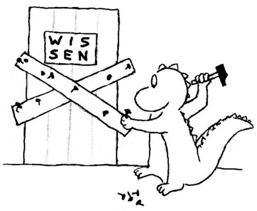
Programmieraufgabe



Partneraufgabe: Zu zweit zu lösen!




Wissenssicherung: Hast du wirklich alles verstanden?




**Lernkontrolle:** Am Schluss jedes Kapitels kannst du selbst prüfen, ob du alles verstanden hast.


**Test:** Wenn du am Schluss der zwei Kapitel über Greedy Algorithmen angelangt bist, kannst du dich bei deinem Lehrer zum Test melden. Dasselbe gilt für die zwei Kapitel über Teile-und-Herrsche.


**Lösungen:** Am Ende des Leitprogramms sind die Lösungen aufgeführt.

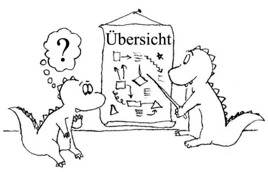
# Inhaltsverzeichnis

# Teil I

# Greedy-Algorithmen

# Kapitel 1

# Getränkeautomaten und Rucksäcke




## Übersicht

### Worum geht es?

In diesem Kapitel wird eine Methode vorgestellt, mit der Algorithmen entworfen werden können. Sie heisst **Greedy-Methode**, auf deutsch gierige Methode. Ein Algorithmus, der nach dieser Methode vorgeht, heisst entsprechend **Greedy-Algorithmus**. Du wirst nun das Prinzip der Greedy-Methode kennen lernen und es an mehreren Beispielen anwenden.
Mit der Greedy-Methode kannst du Algorithmen für viele praxisrelevante Probleme entwerfen. Greedy-Algorithmen sind meistens relativ einfach zu verstehen und zu implementieren. Du lernst also hier eine wichtige Entwurfsmethode, die du wahrscheinlich immer wieder antriffst, wenn du programmieren wirst.

### Was tust du hier?

Lies das Kapitel durch und versuche die Aufgaben zu lösen. Es ist wichtig, dass du nicht weiter liesst, bevor du die Aufgaben gelöst und verstanden hast. Es gibt auch eine Programmieraufgabe, die du am Computer durchführen sollst.

# Lernziele

Am Schluss dieses Kapitels sollst du:

- Das Prinzip der Greedy-Methode verstehen.
- Die Greedy-Algorithmen für das Rückgeldproblem und das Rucksackproblem kennen.
- Für ein gegebenes Problem selbst einen Greedy-Algorithmus entwerfen und implementieren können.

## 1.1 Ein Getränkeautomat

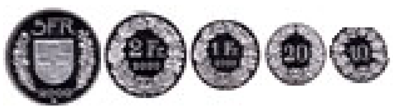
Greedy Algorithmen braucht man häufig, um ein Maximum oder ein Minimum zu suchen. Vielleicht hast du schon das Minimum und Maximum einer Funktion berechnet, indem du die Funktion abgeleitet hast. Darum geht es aber hier nicht. Die Greedy-Methode benützt eine andere Strategie, mit welcher z.B. auch das Maximum und Minimum von Funktionen, die man nicht ableiten kann, berechnet werden kann. Was bedeutet das aber genau?

### Ein Beispiel:

An einem Getränkeautomat kann man verschiedene Getränke kaufen:

| | Getränk | Preis (CHF) |
|---|---|---|
| 1 | Kaffee | 2.30 |
| 2 | Tee | 2.20 |
| 3 | Mineralwasser | 1.10 |

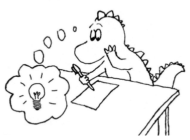
Der Automat funktioniert so:
Wenn man die Nummer eines Getränkes drückt, erscheint der Preis des Getränkes. Nun kann man den Betrag einwerfen. Danach wird das Getränk und das Rückgeld ausgegeben.

Folgende Münzen können eingeworfen werden: 

Das Rückgeld besteht auch aus diesen Münzen. Zur Vereinfachung nehmen an, dass immer genügend Münzen im Automat vorhanden sind.

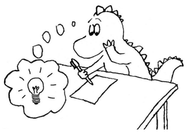
⇒ Der Automat soll jetzt so programmiert werden, dass er für jeden Betrag möglichst wenige Münzen zurück gibt!

Bei diesem Beispiel geht es darum, ein Minimum zu suchen: Die **Anzahl Münzen**, die der Automat zurück gibt soll minimiert werden.



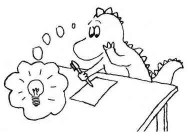
### Aufgabe 1

Kann der Automat auf jeden Betrag Rückgeld herausgeben? Begründe!



## Aufgabe 2

Ein Kunde wählt Kaffee und wirft 3.- ein. Gib alle möglichen Münzkombinationen an, mit denen das korrekte Rückgeld bezahlt werden kann! Welche dieser Kombinationen hat die kleinste Anzahl Münzen und ist somit optimal?



## Aufgabe 3

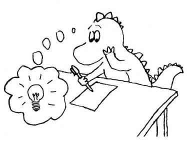
Was ist die minimale Anzahl Münzen, die man zurückbekommt, wenn man für ein Mineralwasser 2.- einwirft?

**Das Rückgeldproblem kann man auch anders formulieren:**

1. Von jeder Münzenart nehmen wir eine bestimmte Anzahl. Die Summe aller dieser Münzen muss gleich dem Rückgeldbetrag $R$ sein, d.h.:

$$a \cdot 5 + b \cdot 2 + c \cdot 1 + d \cdot 0.5 + e \cdot 0.2 + f \cdot 0.1 = R$$

2. $a$, $b$, $c$, $d$, $e$, $f$ müssen ganze Zahlen sein.
3. Die Anzahl der Münzen soll minimal sein: *minimiere* $a + b + c + d + e + f$



## Aufgabe 4

Schau nochmals die Aufgabe 2 an! Welche Werte haben dort $a, b, c, d, e, f$?

## 1.2 Die gierige Vorgehensweise

Das Prinzip der Greedy Methode ist recht einfach. Man geht im Wesentlichen so vor:
Die Lösung bestimmt man in mehreren Schritten. In jedem Schritt versucht man, die meistversprechendste der Möglichkeiten auszuwählen. Davon kommt auch der Begriff gierig ("greedy").

**Ein Beispiel:**
Erinnern wir uns wieder an das Beispiel im letzten Abschnitt. Der Getränkeautomat soll für den Rückgeldbetrag $B_0$ möglichst wenige Münzen herausgeben. Wir wenden nun die Greedy-Methode darauf an.

$B$ bezeichnet immer den Betrag, den der Automat noch herausgeben muss.

- $B =$ Rückgeldbetrag $B_0$, (noch keine Münzen herausgegeben)
- wiederhole:
  - Falls $B = 0$, dann sind wir fertig.
  - Sonst gib die grösstmögliche Münze M aus, die nicht grösser ist als $B$ und setze $B = B - M$

Nehmen wir an, der Automat müsste einen Betrag von 7 Franken herausgeben.

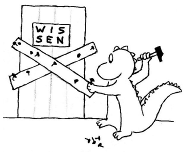
- Wir setzen $B = 7$.
- Die grösste Münze, die höchstens 7 Franken ist, ist der Fünfliber ($M = 5$). Wir geben ihn aus und setzen $B = 7 - 5 = 2$.
- Die nächste Münze, die wir ausgeben ist ein Zweifränkler ($M = 2$). $B$ setzen wir neu auf $B = 2 - 2 = 0$.
- Nun sind wir fertig, da $B = 0$ ist. Wir haben insgesamt zwei Münzen herausgegeben.

Greedy-Methode

Wir starten mit Nichts: Mit der leeren Teillösung.

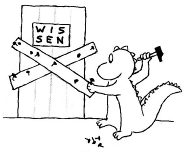
Dann nehmen wir im nächsten Schritt gierig immer die Teillösung hinzu, die am besten scheint.

Wir fahren so fort, bis wir die ganze (optimale) Lösung haben.



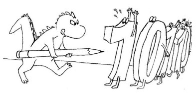

## Wissenssicherung 1

Der Automat soll 1.80.- Rückgeld geben. Wende den Greedy-Algorithmus an diesem Beispiel an!




## Wissenssicherung 2

Erkläre nochmals deiner Banknachbarin mit eigenen Worten, wie das Prinzip eines Greedy-Algorithmus geht.



## Programmieraufgabe 5

Implementiere den Greedy-Algorithmus für das Rückgeldproblem! Benütze dazu die Datei Automat.java und ergänze die Methode rueckgeld! Zur Vereinfachung kannst du das Geld immer in Rappen speichern.
Die möglichen Münzen und die Preise der Getränke sind in zwei Arrays gespeichert:

```java
private static int [] Muenzen;
private static int [] Getraenke;
```

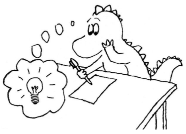
Die Rückgeldmünzen kannst du auf der Konsole ausgeben:

```java
System.out.println(...);
```

Die Methode rueckgeld hat als Argument den Betrag, den der Kunde zahlt und die Nummer des Getränks.

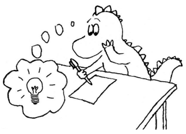
```
public static void rueckgeld(int betrag, int nummer){
   ...
}
```

Kompiliere Automat.java und teste, ob alles richtig läuft!



## Aufgabe 6

Nimm an, in einem Land gibt es nur die Münzen 41, 20 und 1. Ein Automat, der den Greedy-Algorithmus von vorher verwendet, soll 60 zurückzahlen. Welche Münzen gibt er aus? Ist diese Lösung optimal?



## Aufgabe 7

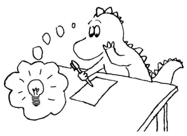
- Finde einen anderen Betrag, für den der Greedy-Algorithmus nicht die optimale Lösung liefert für die Münzen 41, 20 und 1.
- Finde andere 3 Münzen, für die der Greedy-Algorithmus nicht für alle Beträge die optimale Lösung liefert. Gib auch den Betrag an, bei dem Greedy nicht geht.

Leider funktioniert die Greedy-Methode nicht für alle Probleme. Manchmal findet ein Greedy-Algorithmus nicht die gewünschte Lösung, also nicht das Minimum oder Maximum. Das hast du in den vorherigen Aufgaben gesehen.

## Rucksack Problem

Ein Dieb hat einen Rucksack, mit dem man höchstens 50 kg tragen kann. Er möchte seinen Rucksack so füllen, dass sein Profit möglichst hoch ist. Es stehen die folgenden Gegenstände zur Verfügung:

| Gegenstand | Wert | Gewicht |
|---|---|---|
| 1 | 200 | 40 |
| 2 | 120 | 20 |
| 3 | 120 | 30 |

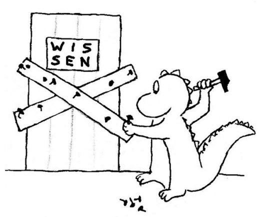


### Aufgabe 8

Entwickle einen Greedy-Algorithmus für dieses Problem! Was ist die optimale Lösung?

| Gegenstand | Wert | Gewicht |
|---|---|---|
| 1 | 60 | 10 |
| 2 | 100 | 20 |
| 3 | 120 | 30 |

Was ist die optimale Lösung und die Lösung deines Greedy-Algorithmus für diese Gegenstände?




# Lernkontrolle

Ein Wanderer will seinen Rucksack mit Gegenständen füllen.

Für ihn hat jeder Gegenstand einen bestimmten Wert pro Volumen. Schokolade ist ihm z.B. wichtiger als Brot aber weniger wichtig als Wasser.

Die Gegenstände kann man alle beliebig teilen. Er kann z.B. nur $0.5l$ Orangensaft einpacken oder eine halbe Tafel Schokolade, die etwa ein Volumen von $0.1l$ hat.

Natürlich ist nicht von jedem Gegenstand beliebig viel vorhanden. Der Wanderer hat z.B. nur eine Tafel Schokolade zu Hause und in seine Trinkflasche kann er höchstens $1.5l$ Wasser einfüllen.

Das Volumen des Rucksacks beträgt $5l$.

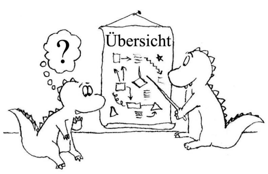
| Gegenstand | Wert (pro Liter) | Vorhandenes Volumen (in Liter) |
|---|---|---|
| Wasser | 5 | 1.5 |
| Schokolade | 4 | 0.2 |
| Orangensaft | 3 | 2 |
| Brot | 2 | 2 |
| Aufschnitt | 1 | 0.1 |
| Gurke | 1 | 0.5 |

1. Wie muss der Wanderer seinen Rucksack füllen, damit er einen möglichst grossen Wert erreichen kann? Wie gross ist dieser Wert?

2. Entwickle einen Greedy-Algorithmus für dieses Problem! Gibt dein Algorithmus immer die optimale Lösung?

# Kapitel 2

# Strassen planen und durch Städte reisen




## Übersicht

### Worum geht es?

Im letzten Kapitel hast du schon gesehen, dass Greedy-Algorithmen nicht immer funktionieren. Für manche Probleme finden sie nicht das Minimum oder das Maximum.

Oft ist aber dann die Lösung, die der Greedy-Algorithmus findet, gar nicht so schlecht. Bei Problemen, die sehr schwierig zu lösen sind, können wir oft mit der Lösung zufrieden sein, die der Greedy-Algorithmus liefert. In diesem Kapitel wirst du ein solches Beispiel kennenlernen.

Zuerst lernst du aber nochmals ein Beispiel kennen, bei dem der Greedy-Algorithmus gut funktioniert.

### Was tust du hier?

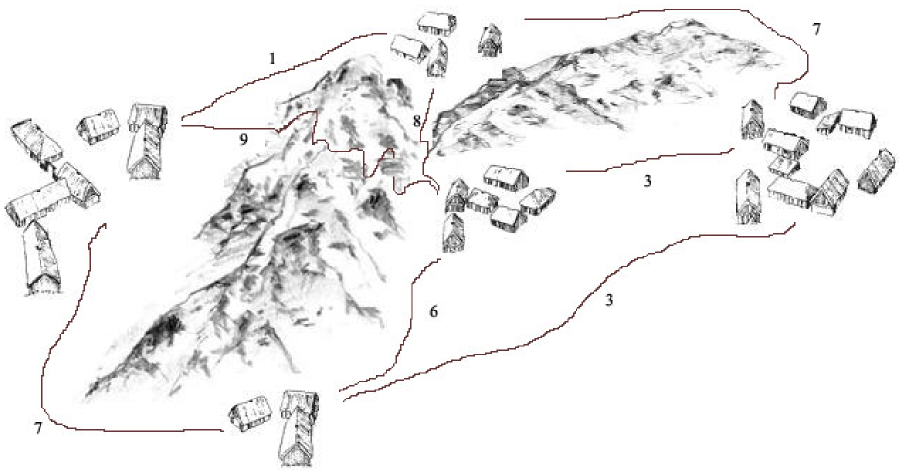
Für dieses Kapitel brauchst du Zugriff aufs Internet. Du wirst Applets benutzen und zusammen mit einem Kollegen/Kollegin ein Java Programm benutzen. Die meisten Aufgaben kannst du aber auf Papier lösen. Du kannst einen Taschenrechner benutzen.

# Lernziele

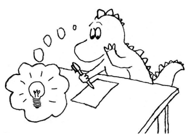
1. Wissen, was ein Minimaler Spannbaum ist und wie man ihn berechnet.
2. Das Travelling Salesman Problem kennen und wissen, wie man mit Hilfe eines Minimalen Spannbaums eine Approximation dafür findet.

## 2.1 Strassen planen

Stell dir vor, du musst bei der Planung eines neuen Strassennetzes mithelfen. Dörfer, zu denen heute noch keine befahrbare Strasse führt, sollen durch neue Strassen verbunden werden. Der Strassenbau kann teilweise sehr teuer werden. Je nachdem müssen dazu Brücken oder Tunnels gebaut werden.
Deine Aufgabe ist nun eine möglichst billige Lösung zu finden, so dass jedes Dorf von jedem anderen aus erreichbar ist.



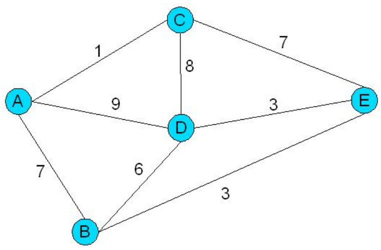

In diesem Plan sind die Dörfer eingezeichnet und die möglichen Strassen. Neben jeder Strasse steht der geschätzte Preis (in 100000 CHF).



### Aufgabe 1

Markiere im obigen Plan die Strassen, die du bauen würdest! Was sind die totalen Kosten für deine Lösung?

## Strassennetz anders dargestellt




Wir können dieses Strassennetz auch als einen Graphen darstellen. Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten, die die Knoten verbinden.
Wenn wir jeder Kante noch einen Wert zuordnen, nennen wir diesen Graphen gewichtet.

Die Knoten entsprechen den Dörfern und die Kanten den möglichen Strassen. Die Werte, die wir den Kanten zugeordnet haben, entsprechen den geschätzten Preisen für den Bau der jeweiligen Strasse.

Die Knotenmenge ist also hier:

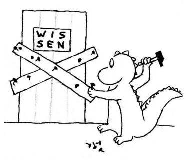
$$V = \{A, B, C, D, E\}$$

Eine Kante kann durch ihre Eckpunkte beschrieben werden. Die Kantenmenge ist hier gleich:

$$E = \{(A, B), (A, C), (A, C), (B, D), (B, E), (C, D), (C, E), (D, E)\}$$

Ein gewichteter Graph hat also

- eine Knotenmenge V
- eine Kantenmenge E. E ist eine Menge von Knotenpaaren aus V
- eine Funktion f, die jeder Kante einen Wert zuordnet




## Wissenssicherung 1

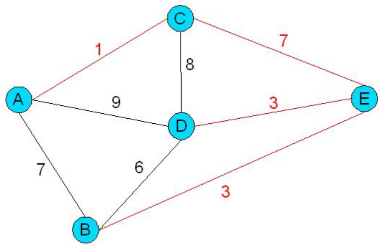
Zeichne einen gewichteten Graphen mit der Knotenmenge V, der Kantenmenge E und der Gewichtsfunktion f!

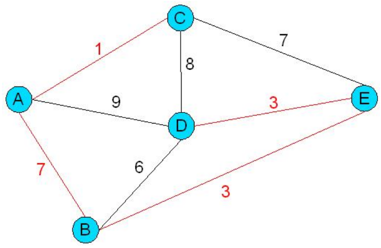
$$V = \{a, b, c, d\} \quad E = \{e_1, e_2, e_3, e_4\}$$

$e_1 = (a, b), \quad e_2 = (a, d), \quad e_3 = (c, d) \quad e_4 = (b, c),$

$f(e_1) = 7, \quad f(e_2) = 6, \quad f(e_3) = 5 \quad f(e_4) = 4$

## Die beste Lösung:

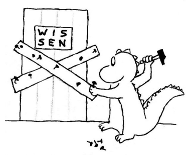



Um das Strassenplanungsproblem zu lösen, muss man eine **Teilmenge der Kanten** auswählen.

Eine Bedingung ist, dass jeder Knoten in mindestens einer Kante enthalten ist. Sonst wären nicht alle Dörfer ans Strassennetz angeschlossen.

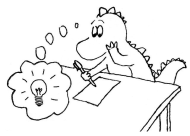
Wir wollen diejenige Teilmenge der Kanten (Strassen), die die kleinsten totalen Kosten hat. D.h. die Summe der Werte der Verbindungslinien soll minimal sein.




## Wissenssicherung 2

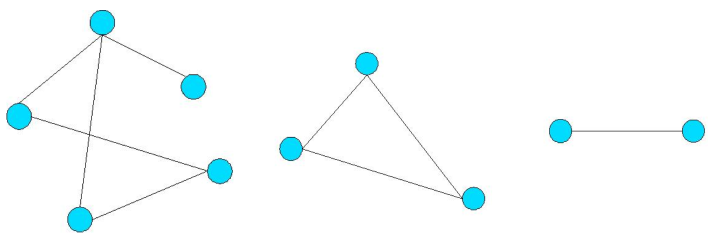
Was ist bei diesen zwei Lösungen die Teilmenge der ausgewählten Kanten? Teste, ob die oben genannte Bedingung zutrifft!

## Ein Baum?



## Aufgabe 2

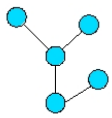
Ein Strassenplaner hat einen Vorschlag eingereicht. Sein Vorschlag enthält einen Zyklus. Das heisst man kann von einem Punkt A über andere Punkte wieder zu A zurückfahren.



Zyklus Zyklus kein Zyklus

Was kannst du tun, um eine solche Lösung zu verbessern?

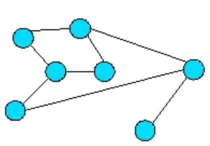
## Spannbaum



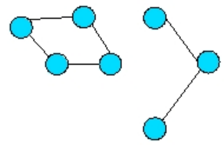
Baum

Wenn ein Graph keinen Zyklus enthält und zusammenhängend ist, dann nennt man ihn auch einen **Baum**.

Ein Graph ist zusammenhängend, wenn jeder Knoten von jedem anderen Knoten erreichbar ist.



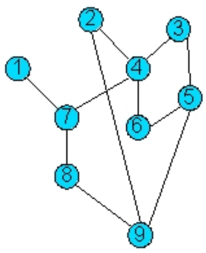
zusammenhängend



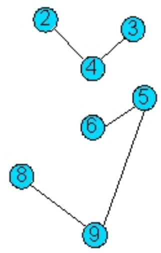
nicht zusammenhängend

Ein Graph T ist ein **Teilgraph** eines Graphen G, falls

- die Knotenmenge von T eine Teilmenge der Knotenmenge von G ist und falls
- die Kantenmenge von T eine Teilmenge der Kantenmenge von G ist.



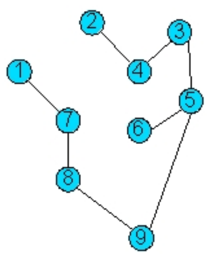

(a) Graph G




(b) Teilgraph T von G

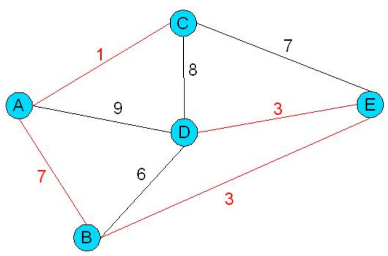
Ein **Spannbaum** ist ein besonderer Teilgraph: Ein Graph heisst Spannbaum von G, wenn er

- ein Teilgraph von G ist
- alle Knoten von G enthält
- ein Baum ist

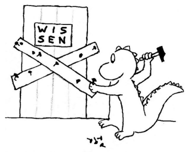



(c) Spannbaum von G

Um das Strassenplanungsproblem zu lösen, müssen wir also einen Spannbaum zu dem Graphen, der das Strassennetz darstellt, finden. Ausserdem wollen wir, dass die Kosten minimal sind. D.h. die Summe der Gewichte des Spannbaums soll minimal sein. Ist dies der Fall, dann nennt man den Spannbaum auch **Minimalen Spannbaum** oder kurz **MST** (Minimum Spanning Tree).

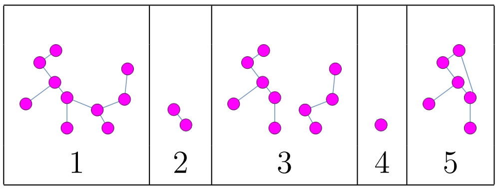


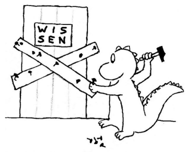

(d) Minimaler Spannbaum




## Wissenssicherung 3

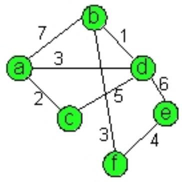
Welche dieser Beispiele sind Bäume?

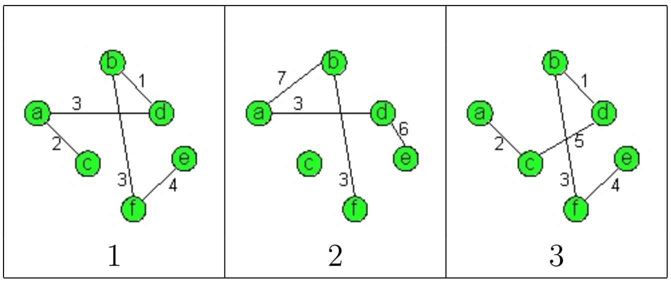



## Wissenssicherung 4

Welche der folgenden Beispiele sind Spannbäume dieses Graphen? Welches sind minimale Spannbäume?




## Anzahl Spannbäume

Die Anzahl der Spannbäume wird schnell ziemlich gross für viele Kanten. Ein Graph mit n Knoten kann $n^{n-2}$ Spannbäume haben. Es macht deshalb keinen Sinn alle Spannbäume durchzugehen und dann den kleinsten zu berechnen. Für grosse Graphen braucht das zu viel Rechenzeit. Viel besser ist es hier einen Greedy-Algorithmus zu benutzen.

| $\boldsymbol{n}$ | 5 | 6 | 10 | 50 |
|---|---|---|---|---|
| $\boldsymbol{n^{n-2}}$ | 125 | 1296 | $10^8$ | $3.6 * 10^{81}$ |

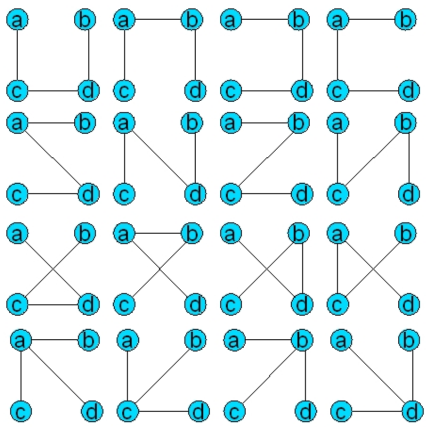
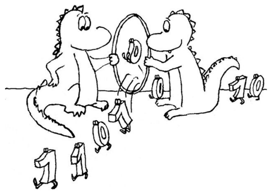


Abbildung 2.1: Alle möglichen Spannbäume für einen Graphen mit 4 Knoten

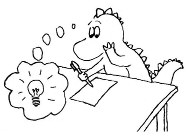


## 2.2 Partnerarbeit: Greedy-Algorithmen zur Berechnung des minimalen Spannbaums

### Algorithmus von Kruskal

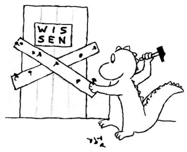
Ihr sollt nun selbst herausfinden, wie der Greedy-Algorithmus von Kruskal funktioniert. Benützt dazu das Applet auf

http://gilco.inpg.fr/ rapine/Graphe/Arbre/kruskalApplet.html[5]



### Aufgabe 3

Lasst das Applet laufen und versucht herauszufinden, wie der Algorithmus von Kruskal geht! Beschreibt den Algorithmus als Text oder in Pseudocode!




## Wissenssicherung 5

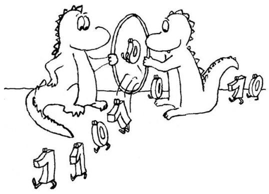
Um euch den Algorithmus von Kruskal nochmals einzuprägen sollt ihr noch ein anderes Applet verwenden. Hier könnt ihr selber Graphen zeichnen.
Ein Mitglied der Gruppe zeichnet einen Graph, die anderen sollen sagen, wie an diesem Beispiel der Algorithmus funktioniert.

Ihr könnt schrittweise testen, ob ihr richtig liegt.

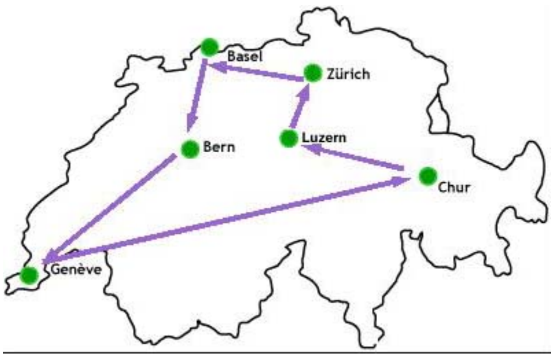
http://links.math.rpi.edu/applets/appindex/graphtheory.html[2]!

**Hinweise zur Bedienung:**

- Setzt den Algorithmus auf "Kruskal's Algorithm".
- Knoten hinzufügen: "add Vertex" einstellen und klicken.
- Kanten hinzufügen: "add Undirected Edge" einstellen und auf den Anfangs- und Endknoten klicken.
- Gewichte hinzufügen: "Modify Edge or Vertex" einstellen und eine Kante doppelklicken. Das Feld "Weight" ändern. Das Feld "Capacity" müsst ihr nicht ändern



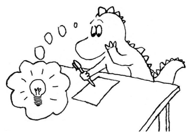
## 2.3 Partnerarbeit: Der Handelsreisende




Ein Handelsreisender plant eine Rundreise durch einige Städte, um seine Waren zu verkaufen.

Jede Stadt will er genau einmal besuchen und am Schluss möchte er wieder in seine Heimatstadt zurückkehren.

Soweit scheint sein Vorhaben ganz einfach. Doch der Handelsreisende will eine möglichst kurze Strecke zurücklegen, um Zeit und Reisekosten zu sparen. Er stellt sich nun die Frage, in welcher Reihenfolge er die Städte besuchen soll.
Dieses Problem heisst auf englisch *travelling salesman problem* oder kurz ***TSP***.



### Aufgabe 4

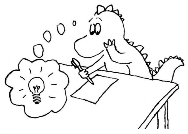
Auf http://www.ibl-unihh.de/tsp/applet.html werden euch drei TSP Probleme gestellt mit Städten in Europa. Befolgt die Anweisungen, die dort stehen. Anschliessend werden noch drei andere Probleme gestellt. Die müsst ihr nicht lösen!

### GraphBench Software [3]

Um das Problem des Handelsreisenden genauer kennenzulernen, sollt ihr die Software GraphBench benutzen. Ihr könnt GraphBench unter

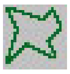
http://www.inf.ethz.ch/personal/braendle/graphbench-de/download.html

herunterladen. Doppelklickt das File graphbench.jar, um Graphbench zu starten und klickt auf "Travelling-Salesman Problem"!



## Aufgabe 5

Mit GraphBench könnt ihr selber ein TSP Problem erzeugen. Ihr könnt mit einem Doppelklick Punkte in die Ebene setzen.

Die Lösung wird berechnet, wenn ihr auf das Icon  klickt.

Überlegt zusammen einen Greedy-Algorithmus, der das TSP Problem möglichst gut löst! Versucht nachher auch ein Beispiel zu finden, für welches euer Algorithmus nicht funktioniert.

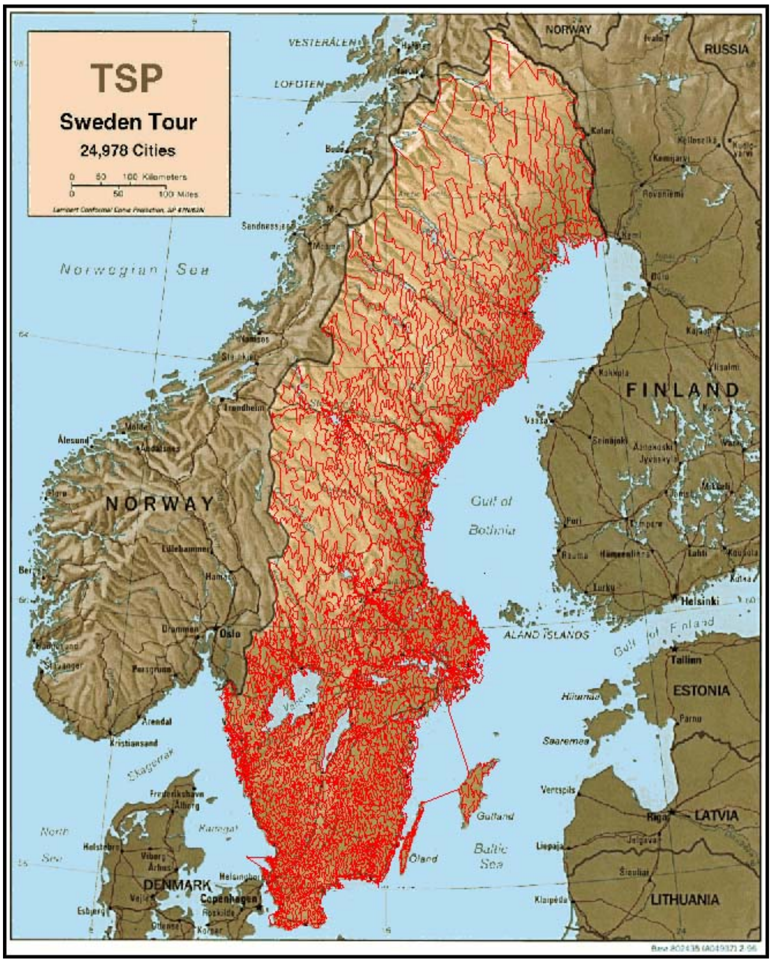
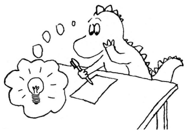



Abbildung 2.2: TSP für 24978 Städte in Schweden [1]

## 2.4 TSP - Ein Schwieriges Problem

Vielleicht habt ihr schon bemerkt, dass GraphBench lange braucht, um die Lösung zu finden,wenn viele Knoten gegeben sind. Das liegt daran, dass bei diesem Algorithmus die Länge für alle möglichen Rundreisen berechnet wird. Dann ist das Resultat die Rundreise mit der kürzesten Länge.

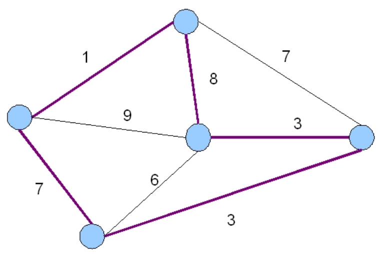


### Aufgabe 6

Zeichne alle möglichen verschiedene Rundreisen mit 4 Knoten! (Die Richtung, in der man herum geht, und der Startknoten, spielen keine Rolle.) Wie viele wären es für einen Graphen mit n Knoten?

### Varianten des TSP

Bisher haben wir die Länge einer Kante immer gleich dem Abstand der zwei Endknoten gesetzt. Wir werden auch weiterhin nur diesen Fall betrachten. Man könnte aber auch die kürzeste Rundreise für einen Graphen mit anderen Längen (Gewichten) berechnen. Ähnlich wie beim minimalen Spannbaum. Diese und andere Varianten wirst du vielleicht antreffen, wenn du andere Literatur zu TSP liest.




### Mehr Information über TSP

Die Forschung über TSP hat etwa in den 30er Jahren begonnen. Seither wurde und wird immer noch sehr viel Forschungsaufwand betrieben, um TSP zu lösen. Es wurden auch sehr grosse Fortschritte gemacht. 1954 ist TSP erstmals für 49 Städte der USA gelöst worden. Damals eine gute Leistung. 50 Jahre später für 24978 schwedische Städte.

| Year | Research Team | Size of Instance | Name |
|---|---|---|---|
| 1954 | G. Dantzig, R. Fulkerson, and S. Johnson | 49 cities | dantzig42 |
| 1971 | M. Held and R.M. Karp | 64 cities | 64 random points |
| 1975 | P.M. Camerini, L. Fratta, and F. Maffioli | 67 cities | 67 random points |
| 1977 | M. Grötschel | 120 cities | gr120 |
| 1980 | H. Crowder and M.W. Padberg | 318 cities | lin318 |
| 1987 | M. Padberg and G. Rinaldi | 532 cities | att532 |
| 1987 | M. Grötschel and O. Holland | 666 cities | gr666 |
| 1987 | M. Padberg and G. Rinaldi | 2,392 cities | pr2392 |
| 1994 | D. Applegate, R. Bixby, V. Chvátal, and W. Cook | 7,397 cities | pla7397 |
| 1998 | D. Applegate, R. Bixby, V. Chvátal, and W. Cook | 13,509 cities | usa13509 |
| 2001 | D. Applegate, R. Bixby, V. Chvátal, and W. Cook | 15,112 cities | d15112 |
| 2004 | D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun | 24,978 cities | sw24798 |

Rekorde für TSP [1]

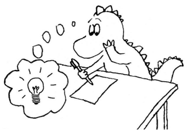
Dieser Fortschritt ist nicht nur auf schnellere Prozessoren zurückzuführen, sondern vor allem auch auf bessere Lösungsmethoden. Es gibt sehr viele verschiedene davon.

Falls du Interesse hast, mehr über TSP zu erfahren, dann schau doch im Internet nach! Eine gute Seite ist z.B.: http://www.tsp.gatech.edu/ (auf Englisch)

## 2.5 MST als Approximation für TSP

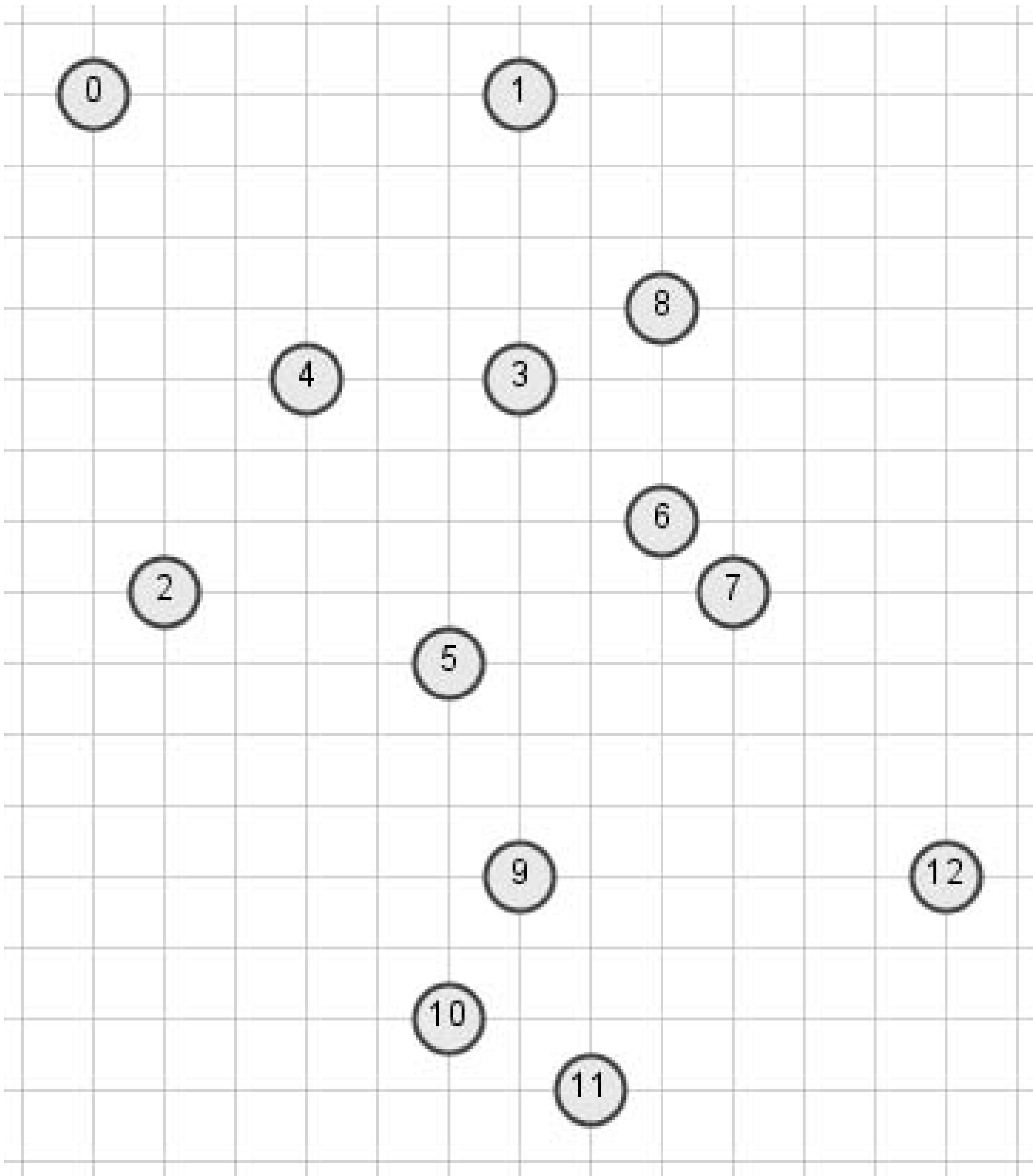
Mit Hilfe eines minimalen Spannbaums kann man für einen Graphen eine (ziemlich kleine) Rundreise berechnen. Nennen wir sie MST-Tour. Die MST-Tour ist nicht unbedingt die kleinste. Sie ist also nicht die optimale Lösung des Travelling Salesman Problems.
Wenn wir eine MST-Tour eines Graphen kennen, dann wissen wir, dass die optimale Lösung kleiner sein muss als die Lösung der MST-Tour.
Die MST-Tour ist also nur eine Näherung der optimalen Lösung. Man nennt eine solche Näherung auch **Approximation**.
Im letzten, freiwilligen Abschnitt dieses Kapitels wird gezeigt, dass die MST-Tour

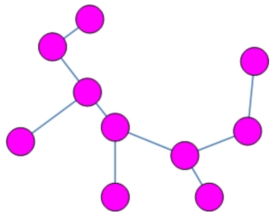
höchstens doppelt so gross sein kann, wie die optimale Tour.



## Aufgabe 7

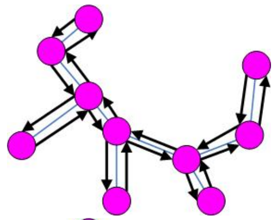
Befolge die Anleitung auf der nächsten Seite und führe sie am folgenden Beispiel durch. Du kannst einfach ins Bild zeichnen.



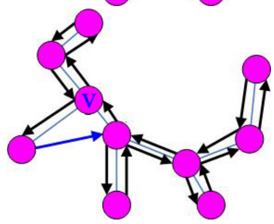


1. Berechne einen minimalen Spannbaum für den Graphen, bei dem alle Städte (Knoten) miteinander verbunden sind. Das Gewicht einer Kante ist jeweils gleich der Länge dieser Kante.

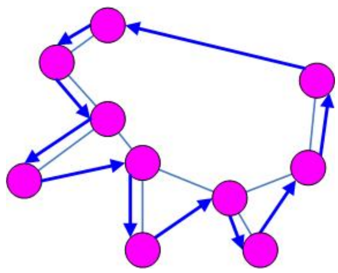
2. Aus dem minimalen Spannbaum bekommst du eine Rundreise, indem du aussen um den Baum herum gehst.
Hier gibt es Knoten, die mehrmals besucht werden. Das kannst du nun Schrittweise ändern:

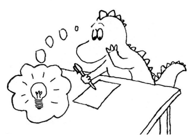


3. Wähle einen Knoten v aus, zu dem mehr als eine Kante führt. Wähle auch eine Kante aus, die zum Knoten hinführt e = (u, v) und eine, die vom Knoten wegführt f = (v, w).
Ersetze diese zwei Kanten durch eine kürzere: g = (u,w).




4. Wiederhole Schritt 3, bis kein Knoten mehr als eine eingehende Kante hat.





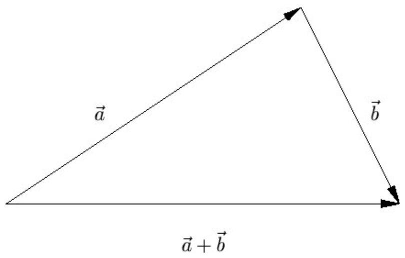
## Aufgabe 8

Zu Schritt 2: Du hast den minimalen Spannbaum gegeben als ein Array von Knoten: und ein Array von Kanten. Wie berechnest du jetzt diese Rundreise aussen herum? Schreibe deine Lösung in Pseudocode oder in Java auf!

## Wie gut ist die Approximation? (freiwillig)

Nennen wir die Länge des minimalen Spannbaums $MST$ und die Länge der Rundreise, die man mit der obigen Anleitung bekommt $MSTTour$. Die Länge der kürzesten Rundreise nennen wir $OPTTour$

In Schritt 2 bekommen wir eine Tour, die genau 2 mal so lang (teuer) ist wie $MST$. Wenn wir Schritt 3 anwenden wird die Tour nie länger. Warum das so ist, zeigt die Dreiecksungleichung.



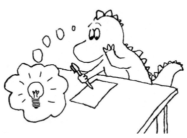

Dreiecksungleichung: $|a+b| \leq |a| + |b|$

**Beobachtung 1:**
Wir können daraus schliessen, dass

$$MSTTour \leq 2MST$$

**Beobachtung 2:**
Wenn wir von der optimalen Tour irgendeine Kante wegnehmen, dann ist das Resultat nicht grösser als der minimale Spannbaum.

$$OPTTour - Länge_{irgendeineKante} \geq MST$$



### Aufgabe 9

Überlege, warum die Beobachtung 2 stimmen muss! Überlege dir den Fall $OPTTour - irgendeineKante < MST$!

Hinweis: $OPTTour - Länge_{irgendeineKante}$ ist ein Baum. Denn wenn man aus einer Rundreise eine Kante wegnimmt, dann gibt es keinen Zyklus mehr. $OPTTour - Länge_{irgendeineKante}$ ist sogar ein Spannbaum, denn es werden alle Knoten besucht.

Aus der Beobachtung 2 können wir schliessen, dass

$$OPTTour \geq MST + Länge_{kleinsteKante}$$

Aus der Beobachtung 1 können wir schliessen, dass

$$MST \geq \frac{1}{2} MSTTour$$

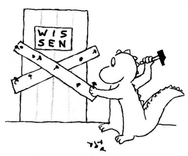
Also ist

$$OPTTour \geq \frac{1}{2} MSTTour + Länge_{kleinsteKante} \geq \frac{1}{2} MSTTour$$

Schlussendlich bekommen wir daraus die Ungleichung

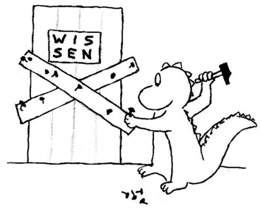
$$\boxed{MSTTour \leq 2OPTTour}$$

**⇒ Die MSTTour ist also höchstens doppelt so lange wie die optimale Tour!**



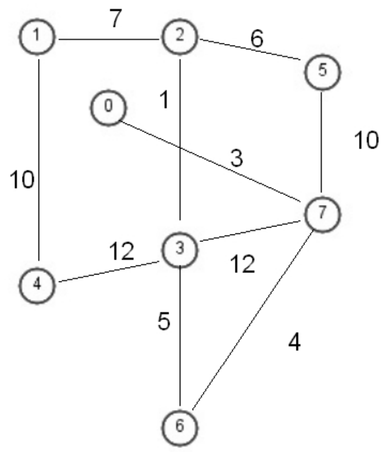

## Wissenssicherung 6

Suche dir einen Kollegen oder eine Kollegin und erkläre ihm/ihr in wenigen Sätzen, warum die Approximation mit dem minimalen Spannbaum höchstens doppelt so lange wie die optimale Tour sein kann!

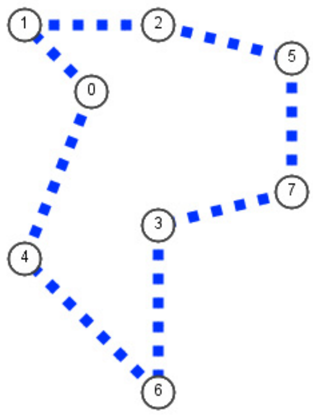



# Lernkontrolle

Zeichne den minimalen Spannbaum ein für den folgenden Graphen! Kannst du damit eine gute Näherung für das Problem des Handelsreisenden finden?




Hier ist die optimale Lösung für das Problem des Handelsreisenden eingezeichnet. Kannst du mit dem minimalen Spannbaum von vorher diese Lösung approximieren?

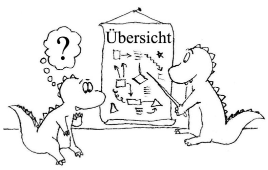


# Teil II

# Teile und Herrsche

# Kapitel 3

# Suchen und anderes




## Übersicht

### Worum geht es?

Du lernst hier eine zweite Entwurfsmethode für Algorithmen kennen. Sie heisst "Teile und Herrsche" (auf Englisch: divide and conquer) Die Idee ist hier die folgende: Man versucht ein Problem aufzuteilen in kleinere Teilprobleme, bis diese so klein sind, dass sie einfach lösbar sind. Aus den Lösungen der Teilprobleme berechnet man dann die Lösung des ursprünglichen Problems.

### Was tust du hier?

Arbeite wie gewohnt das Kapitel durch. Es gibt darin eine Partnerarbeit und mehrere Programmieraufgaben. Für eine Aufgabe benötigst du Jasskarten.

## Lernziele

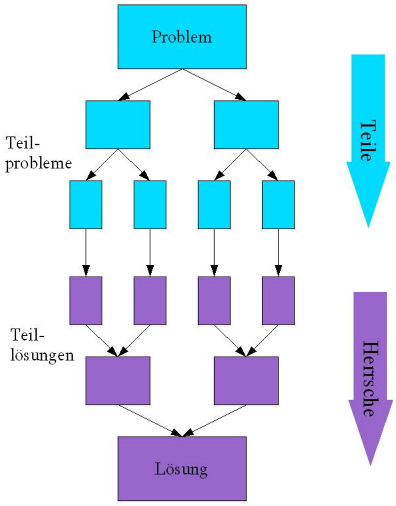
Am Ende des Kapitels sollst du die Teile-und-Herrsche Methode verstehen. Du sollst auch mindestens zwei Beispiele dafür kennen.

## 3.1 Minimum und Maximum gleichzeitig berechnen

Bei der Teile-und-Herrsche Methode nützt man aus, dass man kleinere Probleme einfacher lösen kann als grosse.

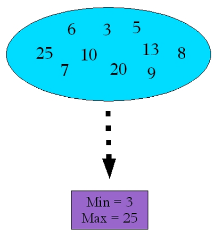
Das Grundprinzip der Entwurfsmethode Teile-und-Herrsche geht so:

1. Teile: Zerlege das Problem rekursiv in Teilprobleme, bis die Teilprobleme so klein sind, dass sie einfach zu lösen sind.
2. Herrsche: Löse die Teilprobleme
3. Herrsche: Setze die Teillösungen wieder zu einer Gesamtlösung zusammen



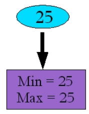

### Beispiel: Kleinstes und grösstes Element einer Menge

Für eine Menge von Zahlen wollen wir gleichzeitig das kleinste Element $Min$ und grösste Element $Max$ suchen. Für eine leere Menge gibt es kein kleinstes und kein grösstes Element. Wir betrachten deshalb diesen Fall nicht. Wie man dieses Problem mit der Teile-und-Herrsche Methode löst, siehst du in der folgenden Anleitung:

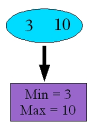



### Anleitung:

Falls die Menge nur ein Element e hat, dann ist dieses gleichzeitig das Minimum und das Maximum. Setze $Min = e$ und $Max = e$.

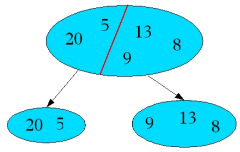



Für eine Menge mit zwei Elementen ($e_1$ und $e_2$) kann man mit einem Vergleich herausfinde, welches das Minimum und welches das Maximum ist.




```
if(e1 < e2){Min = e1; Max = e2; }
else {Min = e2; Max = e1;}
```

Falls die Menge mehr als zwei Elemente hat, dann teile sie auf in zwei etwa gleich grosse Teilmengen.




Für diese beiden Teilmengen müssen wir nun dasselbe Problem lösen wie für unsere Anfangsmenge. Wir wollen nämlich je das kleinste und das grösste Element für beide Teilmengen wissen. Befolge also diese Anleitung für jede Teilmenge. Als Resultat bekommst du je ein kleinstes und ein grösstes Element ($Min1, Max1, Min2, Max2$). Daraus kannst du einfach $Min$ und $Max$ der Anfangsmenge herausfinden.

```
if(Min1 < Min2){Min = Min1}
else{Min = Min2}

if(Max1 > Max2){Max = Max1}
else{Max = Max2}
```

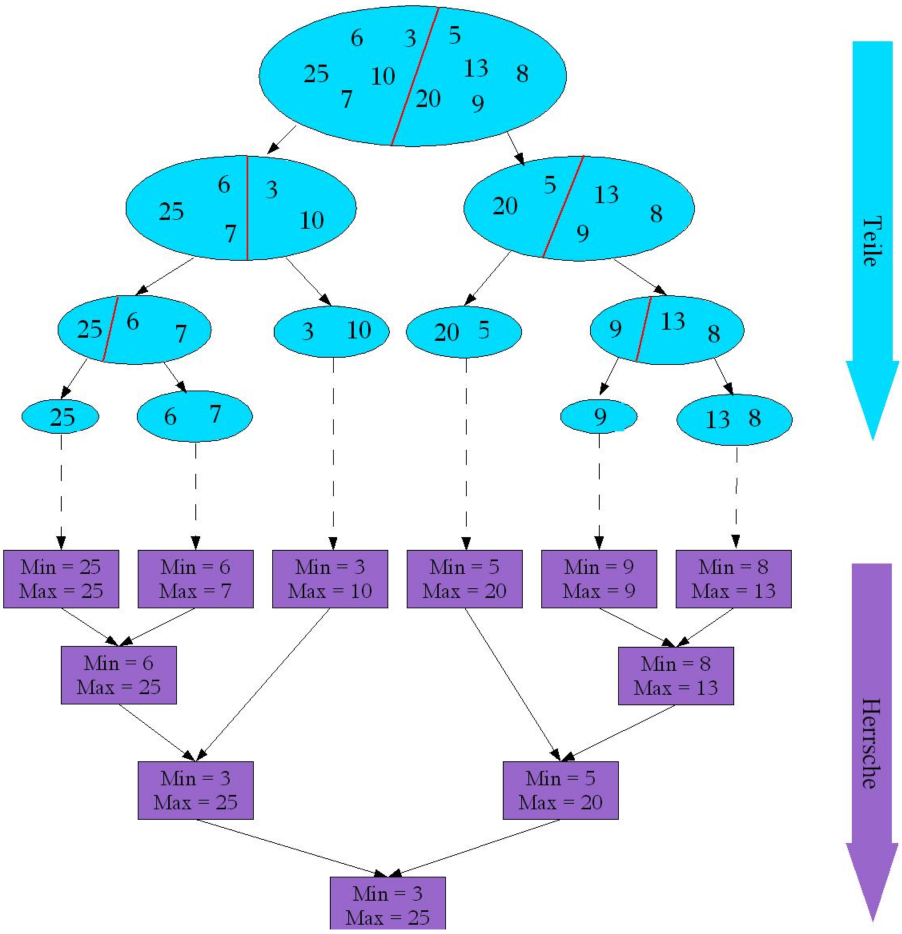


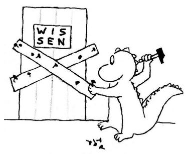

Abbildung 3.1: Minimales und Maximales Element für 10 Zahlen

In Pseudocode haben Teile-und-Herrsche Algorithmen die folgende Form:

Lösung *teile_und_herrsche*(Problem){

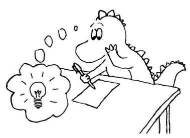
- falls das Problem genügend klein ist, löse es und gib die Lösung zurück
- teile das Problem in Teilprobleme auf
- berechne für jedes Teilproblem die entsprechende Teillösung (Teillösung = *teile_und_herrsche*(Teilproblem))
- setze die Teillösungen zu einer Gesamtlösung zusammen
- gib die Gesamtlösung zurück

}




## Wissenssicherung 1

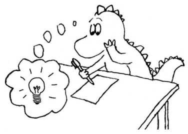
Für diese Aufgabe benötigst du einige Jasskarten dergleichen Farbe: Mische die Karten und führe dann den oben beschriebenen Algorithmus aus! Vergewissere dich, dass du auch wirklich die kleinste und grösste Karte gefunden hast!



## Aufgabe 1

Anstatt die Menge immer in 2 Teilmengen zu teilen, sollst du sie jeweils in 4 Teilmengen teilen. Führe den veränderten Algorithmus nochmals mit Jasskarten durch! Zeichne dann auf, wie der Algorithmus für 10 Zahlen aussehen könnte.

## Vergleiche zählen

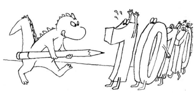


### Aufgabe 2

1. Führe den Algorithmus nochmals mit 8 Karten aus und zähle dabei die Vergleiche, die du dazu benötigst!

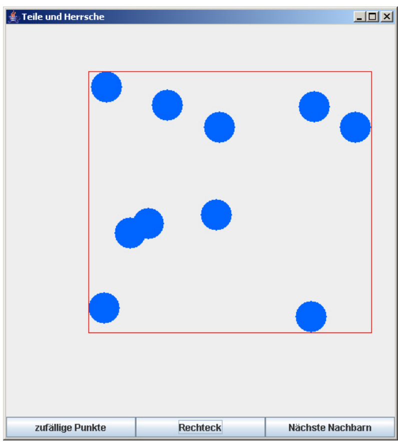
2. Zähle die Anzahl Vergleiche für 8 Karten, wenn du den obigen, veränderten Teile-und-Herrsche Algorithmus benützt, bei dem du eine Menge immer in 4 Teilmengen aufteilst!

3. Zähle die Anzahl Vergleiche für den folgenden Algorithmus: Setze zuerst min und max auf die erste Karte. Nimm die nächste Karte: Falls sie grösser als max oder kleiner als min ist, ändere min oder max entsprechend auf diese Karte. Wiederhole diesen Schritt, bis alle Karten bearbeitet sind. **Hinweis:** Man braucht 2 Vergleiche, um herauszufinden, ob eine Karte kleiner als min oder grösser als max ist.

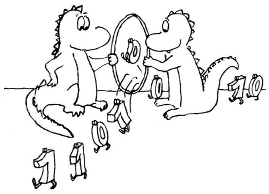


### Programmieraufgabe 3

Für diese Aufgabe benötigst du die Datei *TeileundHerrsche.jar*. Wenn du das Programm startest siehst du ein Fenster mit 3 Knöpfen. Wenn du auf "zufällige Punkte" klickst, erscheinen 10 Punkte mit zufälligen Koordinaten. Der Knopf "Rechteck" soll ein Rechteck zeichnen, das alle Punkte enthält. Das ist jedoch noch nicht vollständig implementiert.




Dazu müssen die kleinste sowie die grösste x- und y-Koordinate berechnet werden. Implementiere den Teile-und-Herrsche Algorithmus, den du eben gelernt hast.
Ergänze dazu die Methode *minmax* in *MinMax.java*. Sie soll jeweils das Minimum und das Maximum von zwei Zahlen berechnen.



## 3.2 Partnerarbeit: Ratespiel

Spielt zwei Runden dieses Spiels. In einer Runde spielt jeder von euch einmal Partner A und einmal Partner B.

**Spielregeln:**

Partner A merkt sich eine Zahl zwischen 1 und 100. Partner B versucht diese Zahl mit möglichst wenigen Fragen zu erraten. Er darf aber nur Fragen stellen, die die folgende Form haben:

Ist die Zahl grösser als ... ?

Zählt die Anzahl Fragen und tragt sie in die Tabelle ein!

**Achtung:** Partner B darf keine falschen Zahlen raten, sonst ist das Spiel ungültig! B darf deshalb die Zahl erst sagen, wenn er ganz sicher sein kann, dass es die richtige ist.

| | Name 1: | | Name 2: | |
|---|---|---|---|---|
| Runde | Gesuchte Zahl | Anzahl Fragen | Gesuchte Zahl | Anzahl Fragen |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| | Total: | | Total: | |

Für die dritte Runde sollt ihr euch zuerst eine Strategie überlegen. Schreibt eure Strategie auf, so dass euer Partner sie nicht sieht. Natürlich muss die Strategie für alle möglichen Zahlen, die der Partner wählt, funktionieren.

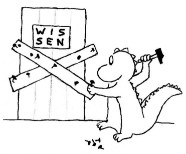
Zählt die Punkte zusammen! Der Verlierer gratuliert dem Gewinner. Keine Angst, es gibt eine Revanche.

## Strategie "Rückwärts"

Liest den Text durch und beantwortet zusammen die Fragen!

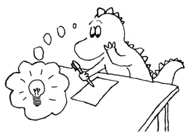
Ein Spieler hat die folgende Strategie:

| | |
|---|---|
| Seine erste Frage lautet: | Ist die Zahl grösser als 99 ? |
| Falls der Gegenspieler mit ja antwortet, weiss er, dass dieser sich 100 gemerkt hat und ist fertig. Sonst erniedrigt er um 1: | Ist die Zahl grösser als 98 ? |
| Falls der Gegenspieler mit ja antwortet, ist er fertig. Sonst erniedrigt er um 1: | Ist die Zahl grösser als 97 ? |
| Er fährt so weiter, bis entweder die Antwort ja ist oder er bei 1 angelangt ist. | ⋮ |
| | Ist die Zahl grösser als 1 ? |




## Wissenssicherung 2

Ist diese Strategie eine Teile-und-Herrsche Strategie? Falls ihr euch nicht sicher seid, schaut euch nochmals die Theorie im Abschnitt vorher an! Was ist hier das Problem, was sind die Teilprobleme und Teillösungen?

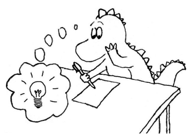


## Aufgabe 4

Es geht wieder darum, eine Zahl zwischen 1 und 100 zu erraten. Welche Zahl muss der Gegenspieler wählen, damit mit der Rückwärtsstrategie möglichst wenige Fragen gestellt werden müssen und wieviele Fragen sind es? Bei welcher Zahl werden die meisten Fragen gestellt?

## Strategie Mitte

Spielt nochmals das Spiel mit der folgenden Strategie: Wähle m immer in der Mitte aller noch in Frage kommenden Zahlen. Frage dann: Ist die Zahl grösser als $m$?



### Aufgabe 5

Wie viele Fragen werden jetzt gestellt? Bei welcher Zahl werden die meisten, bei welcher die wenigsten Fragen gestellt?

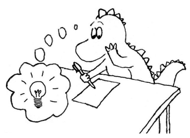
### Programmieraufgabe 6

Schreibe diese Strategie als Pseudocode auf!

## Ratespiel Intervall

Spielt nochmals das Ratespiel mit leicht geänderten Spielregeln: Eine Zahl gilt schon als richtig, wenn sie höchstens um 2 abweicht von der Zahl, die sich der Partner gemerkt hat.

| Name 1: | | Name 2: | |
|---|---|---|---|
| Gesuchte Zahl | Anzahl Fragen | Gesuchte Zahl | Anzahl Fragen |
| | | | |
| | | | |
| Total: | | Total: | |

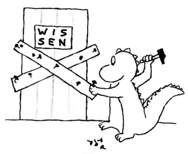


### Aufgabe 7

Wie kann man die Strategie Mitte ändern, dass sie gut fürs Ratespiel Intervall funktioniert? Schreibe deine Lösung in Pseudocode oder in Java!

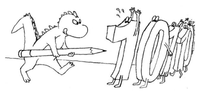
## 3.3 Binäre Suche

Ein ähnliches Problem wie das Ratespiel ist es, ein Objekt in einem sortierten Array zu suchen. Ein Beispiel dazu ist Name und Telefonnummer in einem Telefonbuch zu suchen.



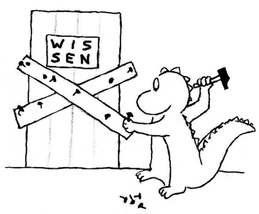

### Wissenssicherung 3

Beschreibe mit eigenen Worten, wie du dein Name und deine Telefonnummer im Telefonbuch suchst. Natürlich sollst du Teile-und-Herrsche verwenden.



### Programmieraufgabe 8

Schreibe einen Teile-und-Herrsche Algorithmus, die dieses Problem löst. Benütze dazu *BinaereSuche.java* und ergänze die Methode *suche*. Teste die Methode mit einem geordneten Array! Was geschieht, wenn ein Wort nicht vorhanden ist?

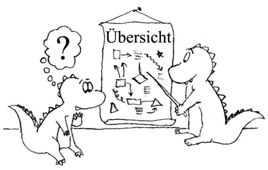



# Lernkontrolle

1. Begründe, warum die binäre Suche nicht funktioniert, wenn die Objekte nicht geordnet sind!
2. Schreibe die einzelnen Schritte des MinMax Algorithmus auf für diese Zahlen:
   7 6 2 9

# Kapitel 4

# Sortieren und nächste Nachbarn finden




## Übersicht

### Worum geht es?

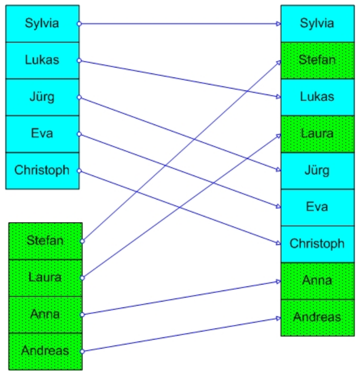
Du wirst nun noch mehr Teile-und-Herrsche Beispiele kennenlernen. Hast du dich z.B. schon mal gefragt, wie man ein Adressbuch sortiert? Hier erfährst du wie es geht.

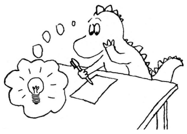
## Lernziele

Das Ziel ist, dass du lernst, die Teile-und-Herrsche Methode anzuwenden, um selbst Algorithmen zu erfinden. Du sollst auch die etwas schwierigeren Teile-und-Herrsche Beispiele in diesem Kapitel verstehen.

## 4.1 MergeSort

Die lernst hier eine Teile-und-Herrsche Methode kennen, um einen ungeordneten Array zu sortieren.
Die Methode beruht darauf, dass man zwei sortierte Arrays einfach und schnell zu einem einzigen, sortierten Array zusammenfügen kann:

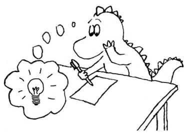


### Aufgabe 1

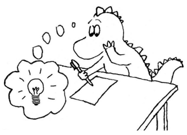
Du brauchst wiedermal Jasskarten. Mische sie und lege dann zwei Reihen von je etwa 5 Jasskarten auf den Tisch. Sortiere jede Reihe einzeln. Jetzt sollst du diese Karten einzeln in eine dritte Reihe verschieben, so, dass diese sortiert ist.

1. Welche Karte kommt als erste in die dritte Reihe?
2. Welche Karten musst du jeweils vergleichen, bevor du eine Karte in die dritte Reihe verschieben kannst?
3. Wie gehst du vor, wenn die erste oder die zweite Reihe leer ist?



### Aufgabe 2

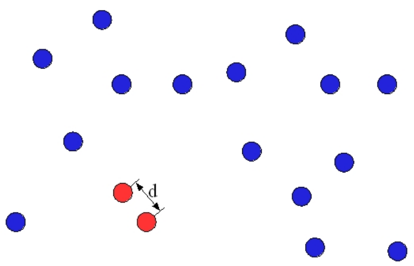
Überlege dir, wie du allgemeit zwei sortierte Arrays zu einem sortierten Array zusammenfügen kannst. Vervollständige dann die Methode *merge* in *MergeSort.java* und teste sie (z.B. mit *merge(teilsortiert,0,4,8);*)



### Aufgabe 3

Wie kann man sortieren mit der Teile-und-Herrsche Methode? Benütze die Methode *merge* der letzten Aufgabe. Überlege dir eine Strategie und vervollständige die Methode *mergeSort*. (teste z.B. mit *sort(unsortiert,0,unsortiert.length -1);*)

## 4.2 Nächste Nachbarn

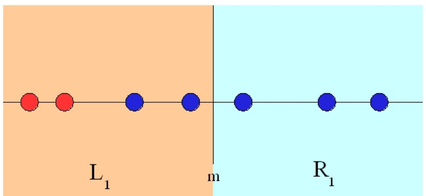



Welche beiden Punkte liegen am nächsten beieinander? Den Abstand für alle möglichen Punktepaare zu vergleichen würde für viele Punkte lange dauern! Zum Glück gibt es eine schnellere Lösung.

### In einer Dimension

Dasselbe in nur einer Dimension Problem ist ein wenig einfacher zu lösen: Gesucht sind die zwei Punkte mit kleinstem Abstand auf einer Gerade.

Wir Teilen die Punkte auf in zwei Teilmengen $L_1$ und $R_1$. Die Aufteilung machen wir an einem Punkt m. Idealerweise liegt m so, dass auf beiden Seiten möglichst gleich viele Punkte sind.

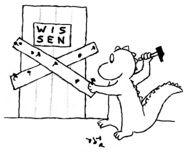



Nehmen wir an, wir hätten das Nächste Nachbarn Problem für $L_1$ und $R_1$ schon gelöst. Wir haben also für $L_1$ die zwei Punkte gefunden mit dem kleinsten Abstand. Dasselbe für $R_1$. Doch wie können wir daraus das Problem für die ganze Punktemenge lösen? Da gibt es noch ein kleines Problem: Das gesuchte Punktepaar könnte auch so liegen, dass ein Punkt in $L_1$ ist und der andere in $R_1$. Es gibt also 3 Möglichkeiten für das Punktepaar mit kleinstem Abstand:

1. Es liegt ganz in $L_1$.

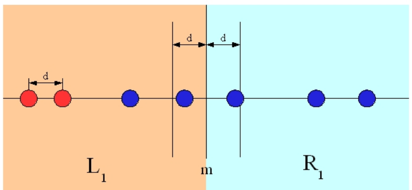
2. Es liegt ganz in $R_1$.

3. Ein Punkt liegt in $L_1$ und der andere in $R_1$



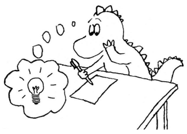

## Wissenssicherung 1

Zeichne ein Beispiel zu jedem dieser drei Fälle und vergleiche es mit einem Mitschüler!




Sagen wir, $d_1$ sei der kleinste Abstand zweier Punkte in $L_1$ und $d_2$ der kleinste Abstand zweier Punkte in $R_1$. Wir nennen $d$ das Minimum von $d_1$ und $d_2$.

Wenn Fall 3 eintrifft, dann wissen wir, das die gesuchten Punkte höchstens einen Abstand d zu m haben können.

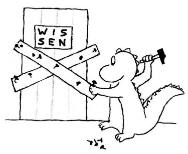


## Aufgabe 4

Warum kann der gesuchte Punkt nicht weiter als d von m entfernt liegen? Kann es sein, dass zwischen m und m+d mehr als ein Punkt liegt?

Der Algorithmus sieht also etwa so aus:

- Falls die Menge nur wenige Elemente hat, dann berechne das Punktepaar mit dem kleinsten Abstand direkt.
- Sonst:
  - Teile:
    - Wähle ein m möglichst in der Mitte
    - Teile die Menge auf in 2 Teilmengen L und R auf, so dass in L alle Elemente kleiner m sind und in R alle grösser oder gleich m.
  - Suche rechts:
    - Suche das Punktepaar mit kleinstem Abstand (dr) in R.
  - Suche links:
    - Suche das Punktepaar mit kleinstem Abstand (dl) in L.
  - Suche in der Mitte:
    - Berechne das Minimum d von dl und dr.
    - Suche alle Punkte, die höchstens Abstand d von m haben. Das sind höchstens 2 Punkte. Berechne ihren Abstand dm.
  - Setze die Lösung zusammen:
    - Berechne das Minimum d von dl, dr und dm und gib das entsprechende Punktepaar zurück.

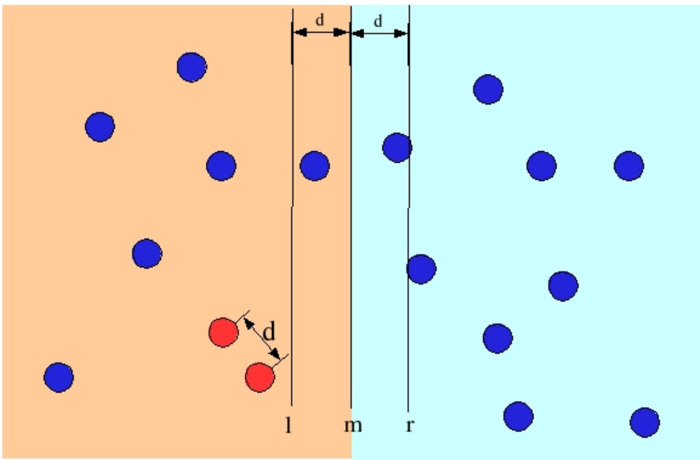



## Wissenssicherung 2

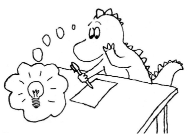
Erkläre kurz deinem Nachbarn in eigenen Worten, wie der obige Algorithmus geht.

## In zwei Dimensionen

In 2D wird die ganze Sache ein wenig schwieriger, aber das Prinzip ist dasselbe. Wir teilen wieder die Punkte in zwei möglichst gleichgrosse Mengen auf. Sei d wieder das Minimum des kleinsten Abstandes in beiden Teilmengen.

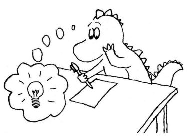



Ein Punkt links von l oder rechts von r kann nicht zum Punktepaar mit dem kleinsten Abstand gehören, weil der Abstand dann sicher grösser als 2d ist.



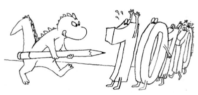
### Aufgabe 5

Wieviele Punkte können jetzt zwischen den Gerade m und Gerade l liegen?



### Aufgabe 6

Ändere den Algorithmus für eine Dimension so, dass er für zwei Dimensionen funktioniert. Schreibe deine Lösung als Text oder Pseudocode auf!

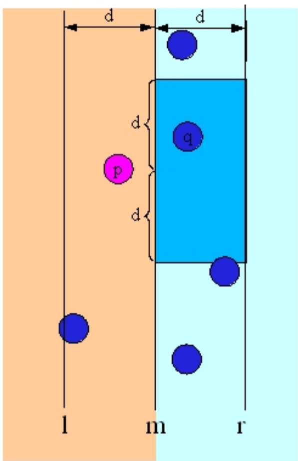


### Programmieraufgabe 7

Nun sollst du deinen Algorithmus implementieren. Benütze wieder *TeileundHerrsche.jar*. Ergänze dazu die Methode *naechsteNachbarn* in *Kreise.java*

## Verbesserung (freiwillig)

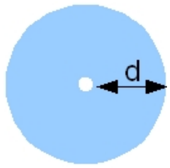
Alle Punkte zwischen l und r zu testen kann ziemlich aufwändig werden. Wir können aber noch eine Verbesserung vornehmen.




Wir wollen wissen, ob es ein Paar nächster Nachbarn gibt, bei dem der eine Punkt p in L1 liegt und der andere Punkt q in R1. Wir wissen dann folgendes über q:

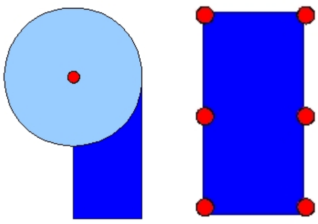
1. q liegt zwischen m und r.
2. Der vertikale Abstand zwischen p und q kann sicher nicht grösser als d sein.

Das heisst für den Punkt p müssen wir nur für alle Punkte im blauen Rechteck den Abstand testen.




In einem Rechteck mit Seitenlängen d und 2d kann es höchstens 6 Punkte geben, die mindestens einen Abstand d voneinander haben:

Im Abstand d von einem Punkt darf kein anderer Punkt sein



Versuche so möglichst viele Punkte zu platzieren.

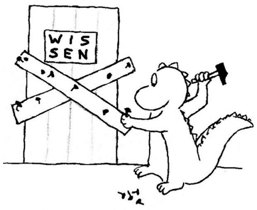
Wir müssen also nur den Abstand zwischen p und seinen 6 Nächsten Nachbarn (zwischen l und m) berechnen.

Der verbesserte Algorithmus sieht also etwa so aus:

- Falls die Menge nur wenige Elemente hat, dann berechne das Punktepaar mit dem kleinsten Abstand direkt.
- Sonst:
  - Teile:
    1. Wähle eine Gerade m, die durch keinen Punkt geht.
    2. Teile die Menge auf in 2 Teilmengen L und R auf, so dass in L alle Elemente links von m sind und in R alle rechts von m.
  - Suche rechts:
    * Suche rekursiv das Punktepaar mit kleinstem Abstand (dl) in L.
  - Suche links:
    * Suche rekursiv das Punktepaar mit kleinstem Abstand (dr) in R.
  - Suche in der Mitte:
    1. Berechne das Minimum d von dl und dr.
    2. Setze l = m-d und r = m + d
    3. Speichere alle Punkte rechts von l und links von m in einem Hilfsarray tmpl.
    4. Speichere alle Punkte rechts von m und links von r in einem Hilfsarray tmpr.
    5. Sortiere tmpl und tmpr bezüglich der y-Koordinate.
    6. Suche die 6 Nachbarn in tmpr für alle Punkte in tmpl und berechne den Abstand. Falls er kleiner ist als der kleinste bisherige, dann merke den Abstand und die Punkte.
    7. Suche die 6 Nachbarn in tmpl für alle Punkte in tmpr und berechne den Abstand. Falls er kleiner ist als der kleinste bisherige, dann merke den Abstand und die Punkte.

## Programmieraufgabe 8

Ändere die Methode *naechsteNachbarn* in *Kreise.java* nochmals, so dass der verbesserte Algorithmus implementiert ist.




# Lernkontrolle

1. Beschreibe, wie du die zwei Zahlen in einem Array findest, die den kleinsten Abstand haben. Du sollst z.B. 2,3 in [2,3,5,10,12,24] finden.
2. Beschreibe, wie die Zahlen 3,7,4,14,7 mit Merge Sort sortiert werden.

# Lösungen

# Lösungen Kapitel 1


## Lösung 1

Ja, der Automat kann auf jeden Betrag Rückgeld herausgeben. Er könnte zum Beispiel den Rückgeldbetrag nur mit Zehnräpplern bezahlen. Der Betrag, den der Kunde bezahlt ist sicher durch 10 Rappen teilbar, da alle möglichen Münzen die er einwerfen kann durch 10 Rappen teilbar sind. Auch alle Preise sind durch 10 Rappen teilbar. Dann muss auch der Rückgeldbetrag (Betrag-Preis) durch 10 Rappen teilbar sein.


## Lösung 2

Das Rückgeld ist $3.00 - 2.30 = 0.70$. Die möglichen Münzkombinationen sind:

| | 0.50 | 0.20 | 0.10 |
|---|---|---|---|
| 1. | 1 | 1 | 0 |
| 2. | 1 | 0 | 2 |
| 3. | 0 | 3 | 1 |
| 4. | 0 | 2 | 3 |
| 5. | 0 | 1 | 5 |
| 6. | 0 | 0 | 7 |

Die erste Möglichkeit ist optimal: Es werden nur 2 Münzen gebraucht.


## Lösung 3

Das Rückgeld beträgt 0.90. Die minimale Anzahl Münzen ist 3. (0.50+2*0.20)


## Lösung 4

| $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 0 | 3 | 1 |
| 0 | 0 | 0 | 0 | 2 | 3 |
| 0 | 0 | 0 | 0 | 1 | 5 |
| 0 | 0 | 0 | 0 | 0 | 7 |


## Lösung Wissenssicherung 1

Der Automat gibt die Münzen 1.00, 0.50, 0.20 und 0.10 zurück:

$$\begin{array}{lll}
1. & & B = 1.80 \\
2. & \mathrm{M} = 1.00 & B = 0.80 \\
3. & \mathrm{M} = 0.50 & B = 0.30 \\
4. & \mathrm{M} = 0.20 & B = 0.10 \\
5. & \mathrm{M} = 0.10 & B = 0.00
\end{array}$$


## Lösung 5

Eine mögliche Lösung:

```
/*
 * Gibt das Rückgeld auf der  Konsole aus für einen
 * gegebenen Preis und Betrag.
```

```
 */
public static void rueckgeld(int betrag, int nummer){
  if(nummer<Getraenke.length && betrag>Getraenke[nummer]){

    int b = betrag - Getraenke[nummer];
    int m,i;

    System.out.println("Rückgeld: "+ b);
    System.out.print("Münzen: ");

    while(b>0){
      i = 0;
      while(i < Muenzen.length && Muenzen[i]>b){i++;}
      if(i<Muenzen.length){
        m = Muenzen[i];
        b = b-m;
        System.out.print(m + ", ");}
    }
    System.out.println();
  }
}
```


## Lösung 6

Der Greedy-Automat gibt 41 und 19*1 aus (20 Münzen). Die optimale Lösung wäre 3*20 (3 Münzen).


## Lösung 7

- Betrag 80. Optimal: $80 = 4 * 20$ (4 Münzen). Greedy: $80 = 41 + 20 + 19 * 1$
- Münzen 1,3,4: Greedy funktioniert nicht für den Betrag 6. Optimal: $6 = 3{+}3$, Greedy: $6 = 4 + 1 + 1$


## Lösung 8

Die optimale Lösung für ist für beide Beispiele Gegenstand 2 und 3 (Wert 220).

Ein möglicher Greedy-Algorithmus: Nimm immer den Gegenstand mit dem höchsten Wert, den man noch nehmen kann, ohne das Maximalgewicht des Rucksacks zu überschreiten.
Beim 1. Beispiel wird der Gegenstand 1 ausgewählt (Wert 200). Das ist nicht optimal. Beim 2. Beispiel werden die Gegenstände 2 und 3 ausgewählt (optimal).

Eine andere Möglichkeit ist: Nimm immer den Gegenstand, der das grösste Verhältnis Wert/Gewicht hat.
Beim 1. Beispiel werden 2 und 3 ausgewählt (optimal). Beim 2. Beispiel 1 und 2 (nicht optimal).

| Gegenstand | Wert | Gewicht | Wert/Gewicht |
|---|---|---|---|
| 1 | 60 | 10 | 6 |
| 2 | 100 | 20 | 5 |
| 3 | 120 | 30 | 4 |


# Lösung Lernkontrolle

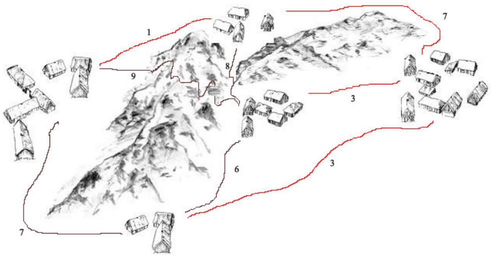
1. Der grösste Wert wird mit 1.5 l Wasser, 0.2 l Schokolade, 2 l Orangensaft und 1.3 l Brot erreicht. Der Wert beträgt $1.5*5+0.2*4+2*3+1.3*2 = 16.9$

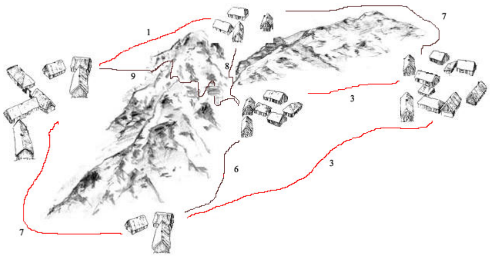
2. Greedy: Nimm immer den (noch vorhandenen) Gegenstand mit dem höchsten Wert pro Volumen und packe so viel davon ein, dass das Volumen des Rucksacks nicht überschritten wird! Der Algorithmus liefert immer den grösstmöglichen Wert, wenn man die Gegenstände beliebig teilen kann.

# Lösungen Kapitel 2


## Lösung 1

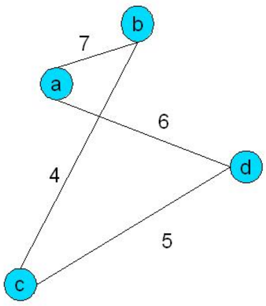
Es gibt 2 Lösungen mit denselben Kosten: (3+3+7+1=14)

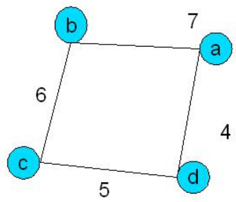





## Lösung Wissenssicherung 1

Es spielt keine Rolle, wo die Knoten gezeichnet sind und ob die Kanten sich schneiden. Zwei mögliche Lösungen sind:



## Lösung Wissenssicherung 2

Die Menge der ausgewählten Kanten ist für die Lösung 1:

$$(A, C), (B, E), (C, E), (D, E)$$

Für die Lösung 2:

$$(A, B), (A, C), (B, E), (D, E)$$

## Lösung 2

Wenn es einen Zyklus gibt, kann man immer eine Kante weglassen. Am besten diejenige mit dem grössten Gewicht. Es werden dann immer noch alle Knoten besucht.

## Lösung Wissenssicherung 3

In den Bildern 1-3 sind Bäume dargestellt.

## Lösung Wissenssicherung 4

1 und 3 sind Spannbäume. 1 ist minimal.

## Lösung 3

Gegeben ist ein gewichteter Graph mit Knotenmenge V, Kantenmenge E und Gewichtsfunktion g: $E \to \mathbb{R}$. Gesucht ist ein minimaler Spannbaum. Die Knoten des minimalen Spannbaums sind genau die Knoten des Graphen V. Die Kantenmenge des Spannbaums müssen wir noch herausfinden. Wir bezeichnen sie mit $E_S$.

> Setze anfangs $E_S = \{\}$.
> Für alle Kanten:
>   Wähle diejenige mit dem kleinsten Gewicht, die keinen Kreis schliesst.

Wie merkt man am besten, ob ein Kreis entsteht, wenn man eine Kante hinzufügt? Eine effiziente Möglichkeit ist die folgende:

> Wir speichern eine Menge von Bäumen B. Anfangs enthält B alle Knoten einzeln. Setze anfangs $E_S = \{\}$. Sortiere E aufsteigend nach dem Gewicht.
>
> Mache für alle Kanten e=(u,v) in E (in aufsteigender Reihenfolge):
>
> - Falls u und v nicht in demselben Baum in B sind, dann nimm die Kante:
>   - Vereinige den Baum mit Knoten u und den Baum mit Knoten v zu einem einzigen Baum.
>   - Füge die Kante e zu $E_S$ hinzu

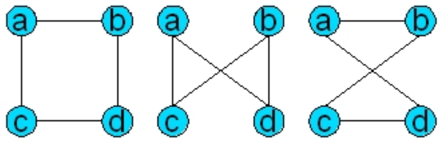

## Lösung 5

Es gibt verschiedene Möglichkeiten. Eine Möglichkeit ist: Nimm immer die kürzeste Kante, die keinen Kreis schliesst, bis alle Knoten verbunden sind.

Du kannst den Algorithmus auf ”Greedy-Heuristics” stellen und ihn dann Schritt für Schritt ausführen. Klicke auf ”Description” und ”Pseudocode” für mehr Information.

Ein Beispiel, bei dem dieser Algorithmus nicht funktioniert, findest du, wenn du beim Menu Examples auf Euclidian-Worst-Case klickst.


## Lösung 6




Allgemein gibt es für einen Graphen mit n Knoten:

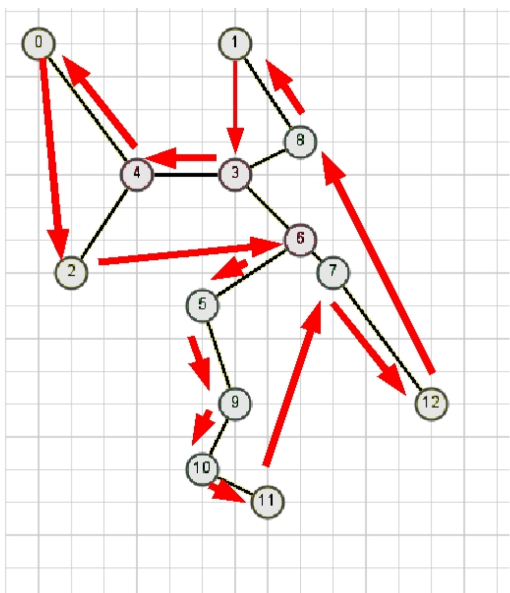
$$(n-1)!/2$$

Rundreisen.
Vom Startknoten gibt es n-1 Möglichkeiten für den zweiten Knoten. Dann gibt es nur noch n-2 Möglichkeiten für den 3. Knoten... Jeder Knoten darf nur einmal besucht werden. Das sind $(n-1)!$. Da es uns egal ist, in welche Richtung die Rundreise geht, gibt es nur noch halb so viele Möglichkeiten.

| Knoten: | 3 | 4 | 5 | 10 | 100 |
|---|---|---|---|---|---|
| Rundreisen: | 1 | 3 | 12 | 181440 | 4.6663e+155 |


## Lösung 7

Es gibt mehrere Lösungen. Eine ist hier abgebildet:




## Lösung 8

Eine mögliche Lösung: Speichere alle Kanten doppelt in einem Array E für beide Richtungen z.B.(a,b) und (b,a).

Starte mit einem beliebigen Knoten c. Wiederhole, bis alle Kanten markiert sind:

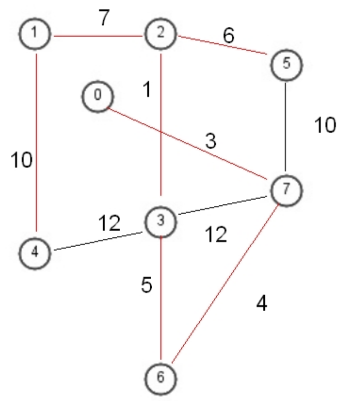
- Suche eine unmarkierte Kante e in E mit Knoten c als ersten Knoten.
- Füge c zur Lösung hinzu und markiere die Kante e.
- Setze c auf den zweiten Knoten von e.


## Lösung 9

Wenn man der optimalen TSP Lösung eine Kante entfernt, dann hat man einen Spannbaum. (siehe Hinweis) Falls dieser kleiner wäre als der minimale Spannbaum, dann wäre dieser nicht minimal! Also muss $OPTTour - irgendeineKante \geq MST$ gelten.


# Lösung Lernkontrolle

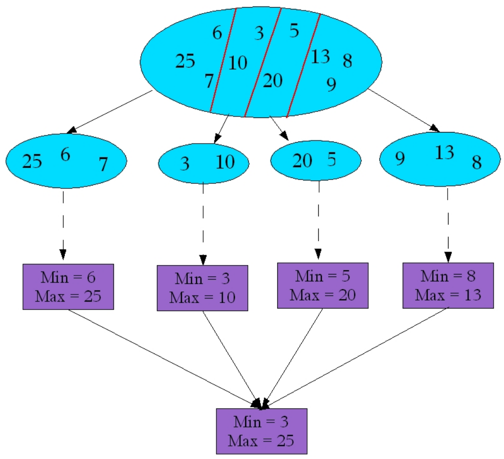
Die Lösung des minimalen Spannbaums eignet sich nicht, um TSP zu approximieren!




Eine Approximation mit dem minimalen Spannbaum aus der Lösung oben eignet sich nicht für diese TSP Lösung. Der minimale Spannbaum wurde für einen Graphen gemacht, bei dem die Gewichte nicht den Abständen entsprechen. Die Tour ist optimal für die Abstände im Graphen und nicht für Gewichte. Einige Kanten (die in der optimalen Tour vorhanden sind) gibt es nicht im ursprünglichen Graphen.

# Lösungen Kapitel 3


## Lösung 1





## Lösung 2

1. 7 Vergleiche
2. 10 Vergleiche
3. 14 Vergleiche im schlimmsten Fall


## Lösung 3

```java
/*
 * Berechnet das Minimum und das Maximum im Array daten im Bereich
 * zwischen anfang und ende
 */
public int[] minmax(int anfang, int ende){
  int res[] = new int[2];

  if((ende-anfang)<=1){
    res[0] = minimum2(daten[anfang],daten[ende]);
    res[1] = maximum2(daten[anfang],daten[ende]);
  }
  else if((ende-anfang)>1){

    int [] t1 = new int[2];
    int [] t2 = new int[2];

    t1 = minmax(anfang,anfang+(ende-anfang)/2);
    t2 = minmax(anfang+(ende-anfang)/2+1,ende);

    res[0] = minimum2(t1[0],t2[0]);
    res[1] = maximum2(t1[1],t2[1]);
  }
  return res;
}
```


## Lösung Wissenssicherung 2

Ja, es ist eine Teile-und-Herrsche Strategie. Die Frage "Ist die Zahl grösser als n?" teilt das Problem in zwei Teilprobleme:

1. Zahlen suchen, die grösser sind als n
2. Zahlen suchen, die kleiner oder gleich n sind

Das besondere hier ist, dass bei jeder Antwort nur noch das eine Teilproblem betrachtet werden muss. Ist die Antwort nein, dann muss man nur noch Fall 2 betrachten. (Bei ja Fall 1). Die Teillösung ist die gesuchte Zahl. Diese ist auch gleich der Gesamtlösung.


## Lösung 4

Bei 100 braucht es nur eine Frage ("Ist die Zahl grösser als 99?"). Bei 1 braucht es 99 Fragen.


## Lösung 5

Es werden immer etwa gleich viele Fragen gestellt. (7 oder 8) Es sind nicht immer genau gleich viele, weil die Mitte nicht immer eindeutig ist.


## Lösung 6

Listing 4.1: Strategie Mitte

```java
/*
 * c: gesuchte Zahl, max > min, c zwischen min und max
 */
public static int suche(int min,int max){
  //Abbrechen falls nur noch 1 Element
  if(max - min == 0){
    return min;
  }
//  wähle m: Mitte zwischen min und max
  int m = (max + min)/2;

//   Ist die gesuchte Zahl grösser als m?
  if(c > m) return suche(m + 1,max);
  else return suche(min,m);
}
```


## Lösung 7

Listing 4.2: Intervall

```java
/*
 * c: gesuchte Zahl, max > min
 */
public static int suche2(int min,int max, int abstand){
  if(max-min <= abstand){
    return min;
  }
//  wähle m: Mitte zwischen min und max
  int m = (max+min)/2;

//  Ist die gesuchte Zahl grösser als m?
  if(c>m) return suche(m+1,max);
  else return suche(min,m);
}
```


## Lösung Wissenssicherung 3

Schlage das Buch in der Mitte auf. Falls dein Name später kommt im Alphabet, dann suche zwischen Mitte und Ende des Buches. Sonnst zwischen Anfang und Mitte. Suche jetzt wieder gleich wie vorher: In der Mitte aufschlagen, ...


## Lösung 8

Listing 4.3: Variante 1 - iterativ

```java
private static void suche(String[] daten,String s,int links,int rechts){
  while(links <=rechts){
    //Mitte abgerundet
    int m = (links+rechts)/2;

    if(s.equals(daten[m])){
      System.out.println(s + " gefunden");
```

```
      return;
    }
    if(s.compareTo(daten[m])<0){
      rechts = m-1;
    }
    else if(s.compareTo(daten[m])>0){
      links = m+1;
    }
  }
  System.out.println(s + " nicht gefunden");
}
```

Listing 4.4: Variante 2 - rekursiv

```
private static void suche2(String[] daten,String s,int links,int rechts){
  if(rechts < links){
    System.out.println(s + " nicht gefunden");
    return;
  }
  //Mitte abgerundet
  int m = (links+rechts)/2;

  if(s.equals(daten[m])){
    System.out.println(s + " gefunden");
    return;
  }

  if(s.compareTo(daten[m])<0){
    suche2(daten,s,links,m-1);
  }

  else if(s.compareTo(daten[m])>0){
    suche(daten,s,m+1,rechts);
  }
}
```


# Lösung Lernkontrolle

1. Bei der binären Suche teilt man die Daten in der Mitte und sucht nur in der einen Hälfte weiter. Wenn die Objekte aber nicht geordnet sind, dann weiss man nicht, in welcher Hälfte sich das gesuchte Objekt befindet.

2. MinMax für 7 6 2 9 :

- Berechne MinMax für $M = \{7, 6, 2, 9\}$
- Teile in zwei Teilmengen $M_L = \{7, 6\}$, $M_R = \{2, 9\}$
- Berechne MinMax für $M_L$
  - Teile $M_L$ in zwei Teilmengen $\{7\}$, $\{6\}$
  - Berechne das Minimum und das Maximum: $min = 6$,$max = 7$
- Berechne MinMax für $M_R$.
  - Teile $M_R$ in zwei Teilmengen $\{2\}$, $\{9\}$
  - Berechne das Minimum und das Maximum: $min = 2$,$max = 9$
- Setze die Lösungen zusammen: $min = 6$, $max = 9$

# Lösungen Kapitel 4


## Lösung 1

1. Die kleinere der beiden ersten Karten der Reihen 1 und 2.
2. Immer die vordersten Karten der Reihen 1 und 2.
3. Wenn die erste Reihe leer ist, kommen die Karten der 2. Reihe nacheinander in die 3. Reihe. Es braucht keine Vergleiche mehr. (Für die 2. Reihe dasselbe umgekehrt.)


## Lösung 2

```
/**
 * Voraussetzung:
 * daten[] ist aufsteigend sortiert von links bis mitte-1 und
 * daten[] ist aufsteigend sortiert von mitte bis rechts
 *
 * Die Elemente von daten[] werden nun so in den Hilfsarray tmp eingefügt,
 * dass tmp[] aufsteigend sortiert ist.
 *
 * Am Schluss werden die Elemente von tmp[] wieder in daten[] gespeichert.
 *
 */
public static void merge(String[] daten, int links, int mitte, int rechts){
  int anzahlDaten = rechts - links +1;
```

```
  int l = links;
  int m = mitte;
  int k = 0;

  String[] tmp = new String[anzahlDaten];

  while((l < mitte) && (m <= rechts)){
    if(daten[l].compareTo(daten[m]) <= 0){
      tmp[k] = daten[l];
      l++; k++;
    }
    else{
      tmp[k] = daten[m];
      m++; k++;
    }
  }

  while(l < mitte){
    tmp[k] = daten[l];
    l++; k++;
  }

  while(m <= rechts){
    tmp[k] = daten[m];
    m++; k++;
  }
  for (int i = 0; i < anzahlDaten; i++)
    daten[links + i] = tmp[i];

}
```


## Lösung 3

```
/**
 * Sortiere einen Array daten[] von links bis rechts mit Merge-Sort
 */
static void sort(String daten[], int links, int rechts)
{
  int anzahlDaten = rechts-links +1;
  int mitte = links + anzahlDaten/2;

  if (anzahlDaten == 0) return; //keine Daten
  if (anzahlDaten == 1) return; //ein Element ist schon sortiert ;)

  sort(daten, links, mitte -1);
```

```
  sort(daten, mitte, rechts);
  merge(daten, links, mitte, rechts);
}
```

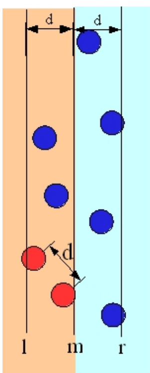
## Lösung 4

1. Wenn die Entfernung eines Punktes p zu m grösser als d ist, dann sind alle Nachbarn auf der anderen Seite von m weiter als d von ihm entfernt. Wir wollen aber nur die nächsten Nachbarn, bei denen ein Punkt p auf der einen Seite liegt und der andere (q) auf der anderen. Wenn p und q also einen Abstand grösser als d haben, dann können sie nicht die nächsten Nachbarn sein. Denn es gibt ein näheres Punktepaar mit Abstand d.

2. Wenn es zwischen m und d+m zwei Punkte hätte, dann hätten diese zwei Punkte einen kleineren Abstand als d. Sie liegen beide auf derselben Seite. d ist aber der kleinste Abstand zweier Punkte auf einer Seite. Das ist ein Widerspruch.

## Lösung 5

Alle Punkte können zwischen l und r liegen:




## Lösung 6

- Falls die Menge nur wenige Elemente hat, dann berechne das Punktepaar mit dem kleinsten Abstand direkt.
- Sonst:
  - Teile:
    - Wähle ein m möglichst in der Mitte
    - Teile die Menge auf in 2 Teilmengen L und R auf, so dass in L die x-Koordinaten aller Elemente kleiner m sind und in R alle grösser m.
  - Suche rechts:
    - Suche das Punktepaar mit kleinstem Abstand (dr) in R.
  - Suche links:
    - Suche das Punktepaar mit kleinstem Abstand (dl) in L.
  - Suche in der Mitte:
    - Berechne das Minimum d von dl und dr.
    - Suche alle Punkte, die höchstens Abstand d von m haben (in x-Richtung). Berechne ihren Abstand dm.
  - Setze die Lösung zusammen:
    - Berechne das Minimum d von dl, dr und dm und gib das entsprechende Punktepaar zurück.


## Lösung 7 + 8

```
/**
 * Berechnet diejenigen 2 Kreise zwischen links und rechts,
 * die den kleinsten Abstand haben.
 * Es wird Teile-und-Herrsche benützt.
 * Die Kreise sind im Array k gespeichert.
 */
```

```java
private Kreis [] naechsteNachbarn(int links ,int rechts){
  //2 Punkte
  Kreis [] res = new Kreis[2];
  if((rechts-links) == 1){
    res[0]=k[rechts];
    res[1]=k[links];
  }
  //3 Punkte
  else if ((rechts-links) == 2){
    int d1,d2,d3;
    d1 = abstand2(k[rechts],k[links]);
    d2 = abstand2(k[rechts],k[links+1]);
    d3 = abstand2(k[links+1],k[links]);

    if ((d1 < d2) && (d1 < d3)){
      res[0]= k[rechts];
      res[1]= k[links];
    }
    else if(d2<d3 && d2<d1){
      res[0]= k[rechts];
      res[1]= k[links+1];
    }
    else{
      res[0]= k[links+1];
      res[1]= k[links];
    }
  }
  // mehr als 3 Punkte
  else{
    //Punktemenge in der Mitte teilen
    int mitte = (rechts+links)/2;
    int xMedian = k[mitte].getPosX();

    res[0]= k[rechts];
    res[1]= k[links];

    Kreis [] l = new Kreis[2];
    Kreis [] r = new Kreis[2];

    //nächste Nachbarn der linken und der rechten Teilmenge berechnen
    l = naechsteNachbarn(links, mitte);
    r = naechsteNachbarn(mitte+1,rechts);

    int dl = abstand2(l[0],l[1]);
    int dr = abstand2(r[0],r[1]);
    int d;

    //die näheren Nachbarn in res speichern
    if(dl<dr){res = l; d = dl;}
    else{res = r; d = dr;}
```

```java
/*
 * Gibt es ein kleineres Punktepaar, bei dem einer in der linken Teilmenge und
 * einer in der rechten Teilmenge liegt?
 */
/*
 * Alle Punkte, die weiter als d von der Mitte entfernt sind wegwerfen,
 * die anderen in tmpl bzw. tmpr speichern
 */
    int j = 0;
    int tmpllength = 0;
    int i = links;
    while(i <= mitte && k[i].getPosX() < (xMedian-d)){i++;}
    while(i<=mitte){
      tmpl[j] = k[i];
      i++; j++;
      tmpllength++;
    }

    j = 0;
    i = mitte+1;
    int tmprlength = 0;
    while(i<rechts && k[i].getPosX() < (xMedian+d)){
      tmpr[j]=k[i];
      i++; j++;
      tmprlength++;
    }

    //tmpr und tmpl nach y sortieren
    Comparator<Kreis> cY = new KreisYComp();
    Arrays.sort(tmpl,0,tmpllength,cY);
    Arrays.sort(tmpr,0,tmprlength,cY);

/*
 * Für jeden Punkt in tmp: Distanz zu 6 Nachbarn messen und schauen,
 * ob sie kleiner ist als die kleinste bisherige Distanz.
 */
    int dm;
    for(i=0;i<tmpllength;i++){
      for(j=0;j<tmprlength;j++){
        dm = abstand2(tmpl[i],tmpr[j]);
        if(dm < d){
          d = dm;
          res[0]=tmpl[i];
          res[1]=tmpr[j];
        }
      }
    }
  }
  //nächste Nachbarn zurückgeben
  return res;
}
```


# Lösung Lernkontrolle

1. Das entspricht dem Problem der nächsten Nachbarn in 1D. Die Zahlen entsprechen den x-Koordinaten. Siehe Abschnitt 4.2.
2. (a) Teile 3,7,4,14,7 in 3,7,4 und 14,7.
   (b) Teile 3,7,4 in 3,7 und 4.
   (c) Sortiere 3,7 und 4 und setze die Lösung zusammen (merge): 3,4,7
   (d) Sortiere 14,7: 7,14
   (e) Setze 3,4,7 und 7,14 zusammen: 3,4,7,7,14

# Tests


# Test: Teil I

1. Beschreibe kurz mit eigenen Worten das Greedy Prinzip. (K2)

2. Ein fauler Kunde möchte ein Getränk kaufen bei einem Getränkeautomaten. Dieser Automat gibt leider kein Rückgeld. Er möchte möglichst wenige Münzen einwerfen. Wie geht er am besten vor? (K1)

3. Entwerfe einen Greedy Algorithmus für folgendes Problem: Ein Wanderer möchte Gegenstände in seinen Rucksack packen. Jeder Gegenstand hat für ihn einen bestimmten Wert. Der Rucksack hat ein Volumen V und kann höchstens ein Gewicht M tragen. (K3)

4. Eine Gruppe von Inseln soll mit Brücken verbunden werden. Wie gehst du vor, wenn die Kosten für den Bau der Brücken möglichst klein sein sollen? (Je länger eine Brücke ist, desto mehr kostet sie!) (K3)

5. Eine Reiseagentur plant Rundreisen mit einem Schiff für Touristen. Die Reise soll nicht zu lange dauern. Trotzdem ist genau ein Halt auf jeder Insel vorgesehen. Wie soll die Reiseagentur ihr Problem lösen? (K3)

6. Entwerfe einen Greedy Algorithmus für das folgende Problem: Mehrere Kunden möchten bedient werden. Jeder Kunde braucht eine gewisse Bedienungszeit. Wähle eine Reihenfolge der Kunden so, dass die totale Wartezeit möglichst klein ist. Z.B.

   - Kunde A 5min
   - Kunde B 12min
   - Kunde C 3min
   - Kunde D 18min

   Wenn die Kunden in der Reihenfolge D,A,B,C bedient werden, dann ist die totale Wartezeit $0 + 18 + 23 + 35 = 76$ (Wartezeit von D + Wartezeit von A + Wartezeit von B + Wartezeit von C ) (K3)


# Test: Teil II

1. Beschreibe mit eigenen Worten das Prinzip Teile-und-Herrsche! (K2)
2. Schreibe eine Teile-und-Herrsche Methode *min*, die das Minimum in einem unsortierten Array findet. (K3) *int min(int array[], int links, int rechts)*
3. Ein Patient liegt mit Schmerzen im Krankenhaus und kann im Moment nicht sprechen. Er versteht aber alles und ist in der Lage zu nicken und den Kopf zu schütteln. Du möchtest herausfinden, wo genau er Schmerzen hat. Welche Fragen kannst du ihm stellen, um dies mit binärer Suche herauszufinden? (K3)
4. Beschreibe mit wenigen, eigenen Worten den Algorithmus "Merge Sort". (K2)

# Lösungen der Tests


## Test: Teil 1: Lösung

1. Siehe Kapitel 1.2.

2. Das ist dasselbe Problem wie das Rückgeldproblem:

   - $B = B_0$, (noch keine Münzen eingeworfen)
   - wiederhole:
     - Falls $B = 0$, dann ist er fertig.
     - Sonst wirf die Münze M ein, die kleiner ist als $B$ und setze $B = B - M$

   Damit das so funktioniert muss der Kunde selbstverständlich von jeder Münzart genügend haben.

3. Es gibt mehrere Möglichkeiten. Die einfachste ist, immer den wertvollsten Gegenstand einzupacken, der weder das Maximalgewicht noch das Volumen des Rucksacks überschreitet. Man kann sich auch folgendes überlegen: Der Wert pro Gewicht und der Wert pro Volumen soll möglichst hoch sein. Also könnte man immer den Gegenstand nehmen, für welchen Wert/Gewicht + Wert/Volumen am grössten ist.

4. Suche den minimalen Spannbaum für einen Graphen mit den Inseln als Knoten und allen möglichen Verbindungen als Kanten. (Es ist möglich von jeder Insel zu jeder anderen eine Brücke zu bauen.) Die Kanten des minimalen Spannbaums sind dann die gesuchten Brücken. Den minimalen Spannbaum berechnet man am besten mit dem Algorithmus von Kruskal.

5. TSP muss gelöst werden. Z.B. indem man alle Rundreisen berechnet und die kleinste auswählt. Vielleicht genügt auch eine Greedy Approximation.

6. Wähle immer denjenigen Kunden, der noch nicht drangekommen ist und die kürzeste Bearbeitungsdauer hat.


## Test: Teil II: Lösung

1. Siehe Kapitel 3.1

2. 
```
int min (int array[], int links, int rechts) {
  if (links == rechts)
    return array[links];
  else {
    int mitte = (links + rechts) / 2;
    int min_l = min(array, links, mitte);
    int min_r = min(array, mitte+1, rechts);

    if (min_l < min_r) return min_l;
    else return min_r;

    }
}
```

3. Zum Beispiel: Tut es dir auf der linken Seite weh? Höher als der Bauch?

4. MergeSort:
   - Falls der Array genügend klein ist, sortiere ihn direkt.
   - Sonst:
     - Teile den Array in zwei Teile
     - Sortiere beide Teile rekursiv
     - Füge die beiden Teile so zusammen dass ein sortierter Array entsteht

# Literaturverzeichnis


[1] Travelling salesman problem. http://www.tsp.gatech.edu/.

[2] Graph theory applet. http://links.math.rpi.edu/applets/appindex/graphtheory.html, 1998.

[3] Markus Brändle. Graph bench. http://www.inf.ethz.ch/personal/braendle/graphbench-de/index.html.

[4] Juraj Hromkovic. *Algorithmics for hard problems*. Springer, Berlin, 2003.

[5] Christophe Rapine. Kruskal applet. http://gilco.inpg.fr/~rapine/Graphe/Arbre/kruskalApplet.html.

[6] Thomas Ottmann; Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum, Akad. Verl., Heidelberg, 2002.