Termin 14:

I) Web-Datenbank-Interaktion
II) Theorie der Algorithmen

Grundlagen der Informatik
Wintersemester 2006/07

Prof. Bernhard Jung

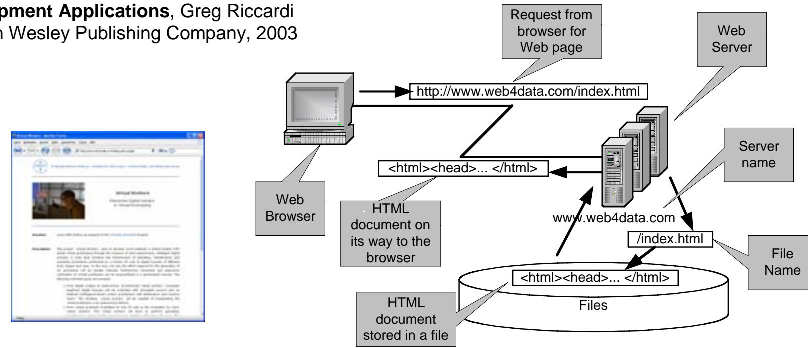
Übersicht

- Generierung dynamischer Web-Seiten (mit Python Server Pages)
  - Verarbeitung von Web-Formular-Eingaben
  - Zugriff auf Datenbankinhalte
- Theorie der Algorithmen
  - Zeitkomplexität von Algorithmen
  - Berechenbare und durchführbare Algorithmen

Literatur
Gumm & Sommer. *Einführung in die Informatik*. Oldenbourg. 2004.
H. Ernst. *Grundlagen und Konzepte der Informatik*. 2. Auflage. Vieweg. 2000.
Goldschlager, Lister. *Informatik – Eine moderne Einführung*. Hanser. 3. Auflage. 1990.
Online
http://www.apachefriends.org
http://de.wikipedia.org/wiki/Sortierverfahren
http://math.hws.edu/TMCM/java/xSortLab/

Architektur statischer Web-Seiten

Bild: **Database Management with Web Site Development Applications**, Greg Riccardi
Addison Wesley Publishing Company, 2003



- HTML-Seiten im Datei-System des Web-Servers gespeichert

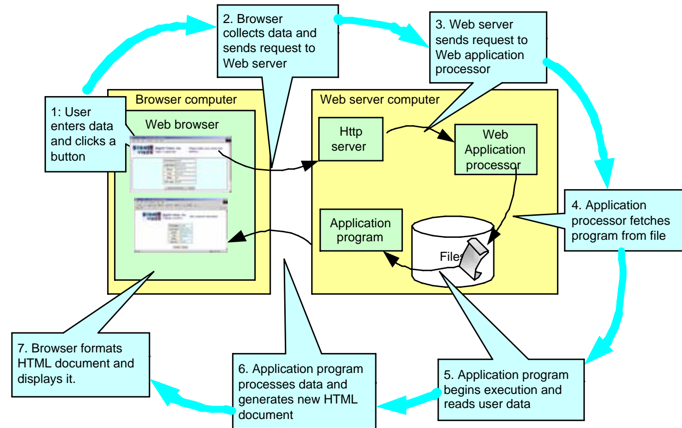
Dynamische Web-Seiten

Eingabe-Formulare

Inhalte z.Tl. aus Datenbank generiert

Architektur dynamischer Web-Seiten

Bild: **Database Management with Web Site Development Applications**, Greg Riccardi Addison Wesley Publishing Company, 2003



- Formulareingaben in Web-Browser triggern Ausführung von Anwendungsprogrammen (z.B. geschrieben in Python) auf Server:
  - Verarbeitung von Benutzer-Eingaben
  - Zugriff auf Datenbank-Inhalte
- Ausgabe der Anwendungsprogramme ist HTML-Code

Generierung dynamischer Web-Seiten

- Wozu?
  - Verarbeitung von Benutzereingaben
  - Anbindung von Datenbanken
- Programmausführung auf dem Server (server side scripting)
  - oft benutzte Programmiersprachen z.B. PHP, Java, Python
  - Einbettung der Programm-Skripte in HTML-Seiten oder Ausführung von CGI (Common Gateway Interface) Skripten, die HTML-Seiten erzeugen
- Im folgenden:
  - Server Side Scripting mit Python
  - "Python Server Pages (PSP)"
    - Einbettung von Python-Code in HTML-Seiten
    - ähnlich zu Java Server Pages (JSP), Microsoft Active Server Pages (ASP), PHP
  - Benutzte Umgebung: Xampp von www.apachefriends.org
    - Apache Web Server, MySQL Datenbank, PSP
    - alles Open Source

Generierung dynamischer Web-Seiten (1)

HTML-Seite mit Login-Formular

```
<html>
<head>
<link href="design.css"
        rel="stylesheet" type="text/css">
<body>

<h1>Login-Formular</h1>
<hr>

 <form action="process.psp" method="POST" >
 Eingabe Name: <input type="text" name="username" />
 Eingabe Passwort: <input type="password" name="passwd" />
 <input type="submit" value="login" />
</form>
<p/>
Hinweis: Passwort ist "TU-BAF"

</body>
</html>
```

Formular mit benannten Eingabefeldern, Name 'process.php' der Datei, die Eingabedaten verarbeiten soll

Generierung dynamischer Web-Seiten (2)

Python-Server Page zur Verarbeitung der Login-Eingaben

```
<html>
 <head><link href="design.css" rel="stylesheet" type="text/c
<body>
<h1>Login-Verarbeitung</h1>
<hr>
<p/>
<%
if form['passwd'] == 'TU-BAF':
   print >> req, 'Willkommen, ', form['username']
   print >> req, ', dein Passwort war richtig'
else:
   print >> req, 'Login fehlgeschlagen', '<p/>'
   print >> req, 'Hinweis: Passwort ist "TU-BAF"'

%>
<hr/>
</body>
</html>
```

process.psp

Eingebettetes Python Skript

Einbettung in HTML-Code mittels <% …%>

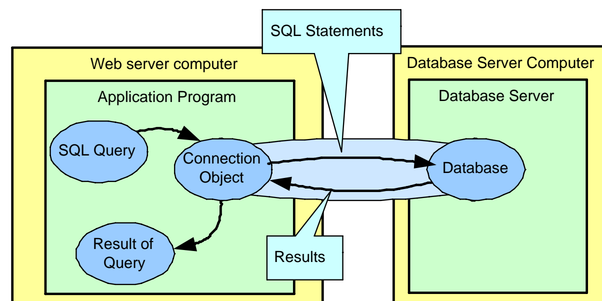
Zugriff auf Formulardaten über Namen der Eingabefelder

Ausgaben mittels print >> req in zurückgeliefertes HTML (req ist in PSP vordefinierte Variable)

Generierung dynamischer Web-Seiten (3)

Zugriff auf Datenbanken mit Python

- Schritte
  - Verbindung zu Datenbank aufbauen
  - SQL-Anfrage an Datenbank-Server schicken
  - Verarbeitung der zurückgelieferten Ergebnistabelle
- Dazu: Import und Verwendung von Funktionen des Moduls MySQLdb



Generierung dynamischer Web-Seiten (4)

Python-Server Page mit Zugriff auf Datenbank

```
<h1> Unsere Universität</h1>
(demonstriert Datenbank-Zugriff mit Python Server
Pages)
<p/>
<hr>
<h3>Unsere Studenten</h3>
<%
import MySQLdb

db = MySQLdb.connect(host="localhost",
      user="user1", passwd="", db="universitaet")
cursor = db.cursor()
cursor.execute("SELECT name FROM student
ORDER BY name")
result = cursor.fetchall()

for record in result:
    print >> req, record[0]
    print >> req, '<br/>'

%>
```

Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://localhost/test/db.psp

Unsere Universität

(demonstriert Datenbank-Zugriff mit Python Server Pages)

Unsere Studenten

Aristoxenos
Carnap
Feuerbach
Fichte
Jonas
Schopenhauer
Theophrastos
Xenokrates

db.psp

Theorie der Algorithmen
Zeitkomplexität

- Bewertung von Algorithmen
  - Wie viele Computer-Ressourcen benötigt Ausführung des Algorithmus?
  - hauptsächlich: Zeit, Speicher; auch Anzahl Prozessoren
- Maß für Zeitkomplexität (Ausführungszeit)
  - Anzahl Operationen der Algorithmenausführung
    - in Proportion zur Größe der Eingaben
  - Grobe Abschätzung dabei pragmatisch ausreichend
    - z.B. "proportional zu n", "proportional zu n^2" mit n Größe der Eingabe
  - d.h. Abstraktion von Ausführungsgeschwindigkeit von Programmen auf spezifischer Hardware

Theorie der Algorithmen
Zeitkomplexität

- Beispiel: Zeitkomplexität des Standard-Multiplikations-Algorithmus

```
1984 * 6713
-----------
   11904
    13888
      1984
       5952
-----------
   13318592
```

Multiplikation zweier n-stelligen Zahlen:

jede Zeile kann in n Schritten (Zeiteinheiten) berechnet werden; es gibt n Zeilen

es müssen n Zeilen mit jeweils n (oder n+1) Ziffern addiert werden

→ Ausführungszeit proportional zu n*n oder n^2

Sortier-Algorithmus 1: Bubble-Sort

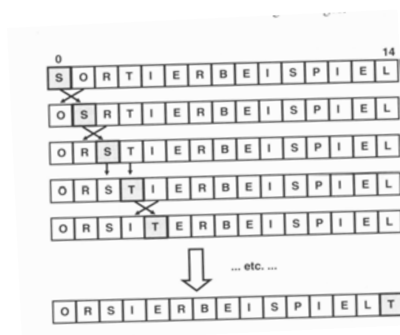
- Sortieren von Listen
  - z.B. [5,3,4,1,2] → [1,2,3,4,5]
- Idee Bubble-Sort
  - Wiederholtes Vertauschen von benachbarten Elementen, die in falscher Reihenfolge stehen; dazu i.a. mehrere Durchläufe notwendig
  - Name rührt daher, dass in jedem Durchlauf große Elemente "wie Blasen im Wasser" ihrer endgültigen Position entgegenstreben

Algorithmus in Pseudo-Code

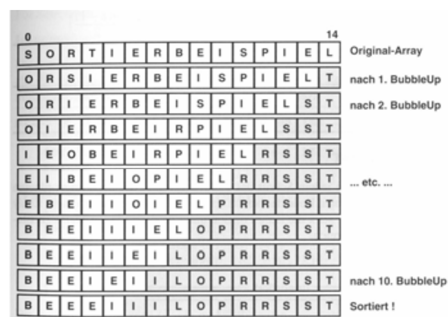
```
Schleife über Durchlauf = 1 .. N-1
  Schleife über Position = 0 .. N-1-Durchlauf
    Falls x[Position] > x[Position+1]
      dann Vertausche x[Position] und x[Position+1]
```


Sortier-Algorithmus 1: Bubble-Sort

Veranschaulichung



1. Durchlauf
danach: größtes Element in endgültiger Position



nach 10 Durchläufen:
10 größte Elemente in endgültiger Position

allg: nach (n-1) Durchläufen ist Liste sortiert

Sortier-Algorithmus 1: Bubble-Sort
Zeit-Komplexität

- Anzahl Vergleichsoperationen
  - Durchgänge äußere Schleife:
    - (n-1)-mal (n = Größe der Liste)
  - Durchgänge innere Schleife (= Anz. Vergleichsoperationen):
    - $(n-2) + (n-3) + (n-4) + .. + 3 + 2 + 1 \sim n^2 / 2$
  - → Ausführungszeit des Alg. proportional zu n^2

Algorithmus in Pseudo-Code

```
Schleife über Durchlauf = 1 .. N-1
  Schleife über Position = 0 .. N-1-Durchlauf
    Falls x[Position] > x[Position+1]
      dann Vertausche x[Position] und x[Position+1]
```


Sortier-Algorithmus 2: Merge-Sort

- Beispiel für Teile-und-Herrsche (Divide and Conquer) Methode
- Idee
  - Sortieren ist einfacher für kurze als für lange Listen
  - insbesondere sind Listen mit nur einem Element schon sortiert
- konkret
  - Teile Liste in zwei kleinere Teil-Listen
  - sortiere die Teil-Listen (evtl. durch weiteres Teilen)
  - Verschmelze die sortierten Teil-Listen zum Endergebnis

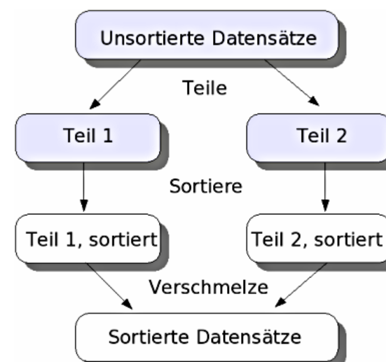


Bild: wikipedia.de

Sortier-Algorithmus 2: Merge-Sort

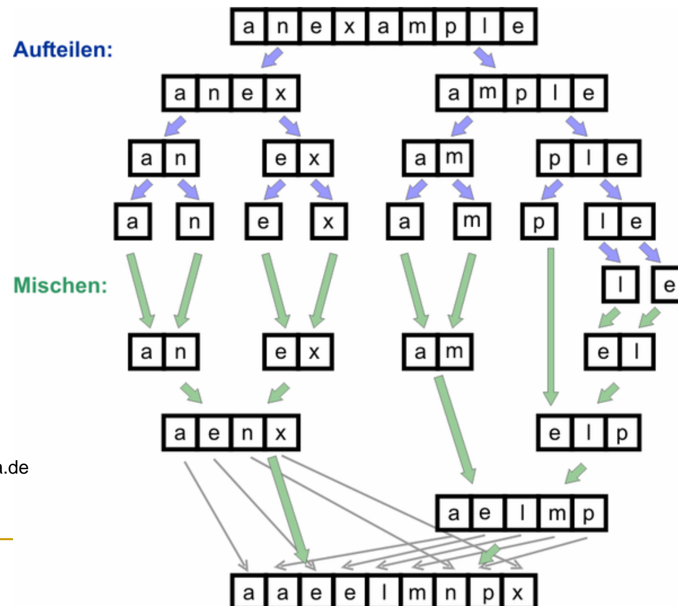


Bild: wikipedia.de

Sortier-Algorithmus 2: Merge-Sort
Zeitkomplexität

Pseudocode für Algorithmus Merge-Sort

```
Funktion sort(liste)
  falls len(liste) > 1
     teil1, teil2 ← split(liste)
     teil1 = sort(teil1)
     teil2 = sort(teil2)
     liste = merge(teil1, teil2)
```

- kann gezeigt werden:
  - Ausführungszeit von Merge-Sort proportional zu *n * log n*, mit *n* Größe der Eingabeliste
  - → bessere Zeitkomplexität als Bubble-Sort
- kann auch gezeigt werden:
  - minimale Ausführungszeit für beliebige Algorithmen zur Lösung des *Problems* "Sortieren von Listen der Länge *n*" ist proportional zu *n log n*
  - → **Merge-Sort ist optimaler Sortieralgorithmus** (bzgl. Zeitkomplexität)

Theorie der Algorithmen: Berechenbare und durchführbare Probleme

- Berechenbares (computable) Problem
  - Aufgabenstellung, die *im Prinzip* durch Algorithmen gelöst werden kann
- Durchführbares (feasible) Problem
  - berechenbares Problem, das durch Algorithmen mit vertretbarem Aufwand an Ressourcen (insbesondere Zeit) gelöst werden kann
  - z.B. Algorithmen mit Ausführungszeit proportional zu $n \log n$, n, n^2, n^3, n^{1000}, … (*polynomiale Algorithmen*)
  - aber nicht Algorithmen mit Ausführungszeit proportional zu 2^n, 3^n, … (*exponentielle Algorithmen*)
  - nicht durchführbare Probleme sind in Praxis nur für kleine Eingaben (durch Algorithmen) lösbar

Ausführungszeiten für Algorithmen unterschiedlicher Zeitkomplexität

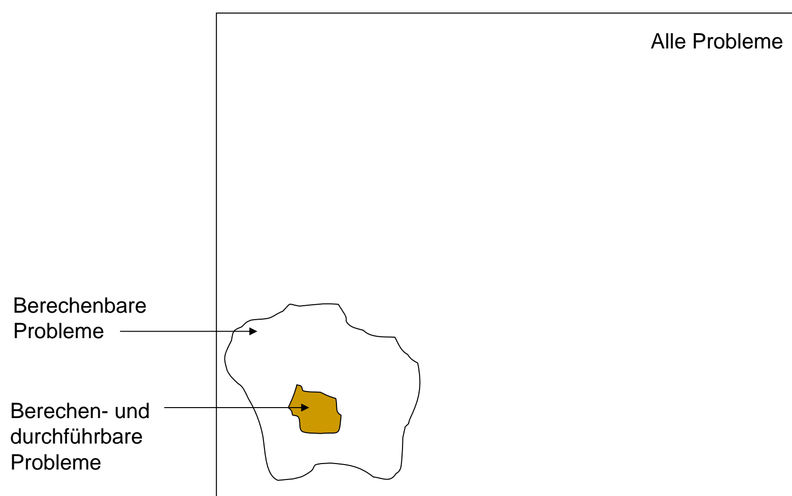
Größe n der Eingabedaten	$\log_2 n$ Mikrosekunden	n Mikrosekunden	n^2 Mikrosekunden	2^n Mikrosekunden
10	0.000003 Sekunden	0.00001 Sekunden	0.0001 Sekunden	0.001 Sekunden
100	0.000007 Sekunden	0.0001 Sekunden	0.01 Sekunden	10^{14} Jahrhunderte
1000	0.00001 Sekunden	0.001 Sekunden	1 Sekunde	astronomisch
10000	0.000013 Sekunden	0.01 Sekunden	1.7 Minuten	astronomisch
100000	0.000017 Sekunden	0.1 Sekunden	2.8 Stunden	astronomisch

(Praktisch) nicht durchführbare Algorithmen Beispiele

- Problem des Handlungsreisenden
  - gegeben: Karte mit n Städten
  - gesucht (Variante als *Entscheidungsproblem*):
    - Gibt es eine Rundreise, die jede Stadt genau einmal besucht, unter Einhaltung einer vorgegeben Kilometerbeschränkung?
  - gesucht (Variante als *Optimierungsproblem*):
    - was ist die kürzeste Rundreise?
- Kryptographische Probleme
- …

- Bemerkungen
  - exakte Lösungen obiger Probleme erfordern i.a. Ausprobieren aller möglichen Kombinationen
  - z.Tl. existieren approximative Algorithmen, die in Praxis zufrieden stellende Ergebnisse liefern

Berechenbare und durchführbare Probleme



Nicht berechenbare Probleme

- kann gezeigt werden:
  - Anzahl aller möglichen Algorithmen ist abzählbar unendlich
  - Anzahl aller Probleme ist überabzählbar unendlich
  - → es gibt (sehr viele) Probleme, die algorithmisch nicht lösbar sind
- Beispiel für nicht berechenbares Problem: Halteproblem
  - Eingabe: beliebiger Algorithmus / Programm
  - Frage: stoppt Algorithmus / Programm für beliebige Eingaben?
  - Relevanz für Programmierung:
    - i.a. ist es unbestimmbar, ob ein beliebiges Programm Endlosschleifen enthält

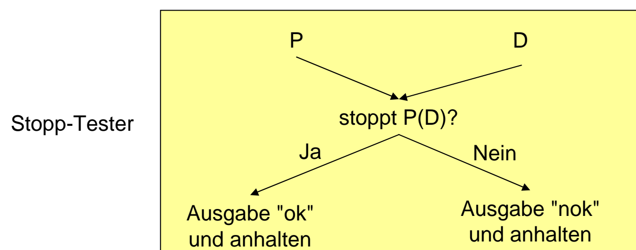
Nicht berechenbare Probleme
Halteproblem (1)

- Beweisskizze: Es gibt keinen Algorithmus, der Halteproblem löst

1. Nehme an, dass ein Programm *Stopp-Tester* zur Lösung des Halteproblems geschrieben werden kann
2. Benutze *Stopp-Tester*, um ein anderes Programm *Spaßig* zu bilden (mittels eines Zwischenprogramms *Stopp-Tester-Neu*)
3. Zeige, dass das Programm *Spaßig* eine undenkbare Eigenschaft hat (es kann weder stoppen noch endlos laufen)
4. Folgere, dass Annahme in Schritt 1 falsch ist

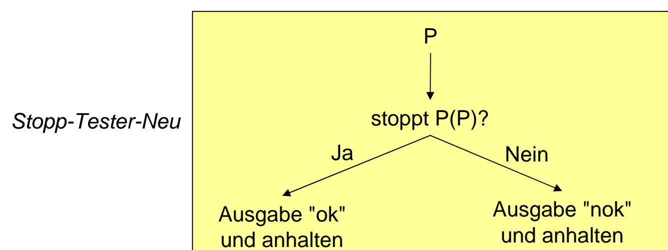
Nicht berechenbare Probleme
Halteproblem (2)

- Funktionsweise eines hypothetischen Programms Stopp-Tester, welches das Halteproblem löst:
  - Eingaben: Programm P, Eingabedaten D
  - Ausgabe:
    - ok, falls P mit Eingabe D stppt
    - nok, sonst



Nicht berechenbare Probleme
Halteproblem (3)

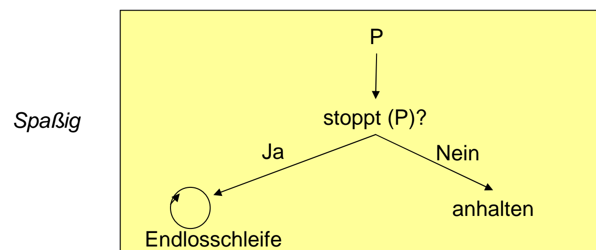
- Konstruktion einer Variante *Stopp-Tester-Neu* von *Stopp-Tester*
- Eingabe: Programm P
- Ausgabe:
  - ok, falls P mit Eingabe P (als Programmtext) stoppt
  - nok, sonst



Nicht berechenbare Probleme
Halteproblem (4)

- Konstruktion einer Variante *Spaßig* von *Stopp-Tester-Neu*
- Eingabe: Programm P
- Verhalten:
  - Endlosschleife, falls *Stopp-Tester-Neu(P)* Ergebnis "ok" liefert
  - anhalten, sonst

Spaßig

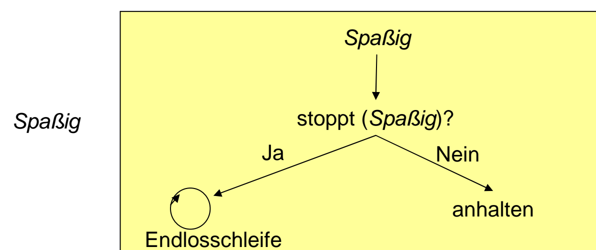


Nicht berechenbare Probleme
Halteproblem (5)

- Verhalten von *Spaßig* mit Eingabe *Spaßig*
  - falls *Spaßig* stoppt, dann geht *Spaßig* in Endlosschleife
  - falls *Spaßig* nicht stoppt, dann stoppt *Spaßig*

Widerspruch!

Spaßig



Nicht berechenbare Probleme
Halteproblem (6)

- Beispiel für Algorithmus, für den bisher nicht klar ist, ob er für beliebige Eingaben stoppt

Ulam-Algorithmus