

# ProInformatik III

## Objektorientierte Programmierung

SoSe 2018

Oliver Wiese  
AG Sichere Identität

## Tutoren

Jakob Bode

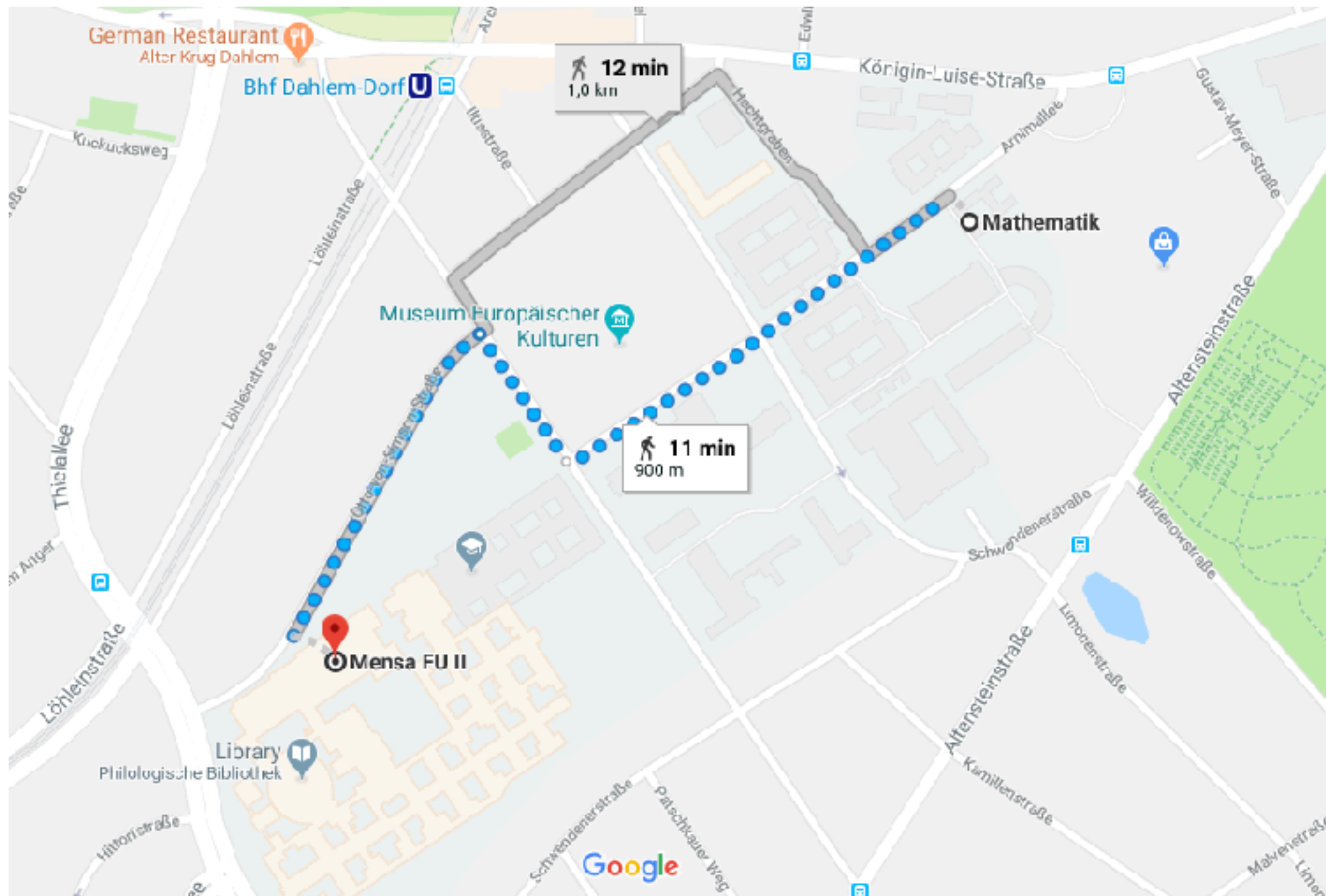
Fabian Halama

Alexander Korzec

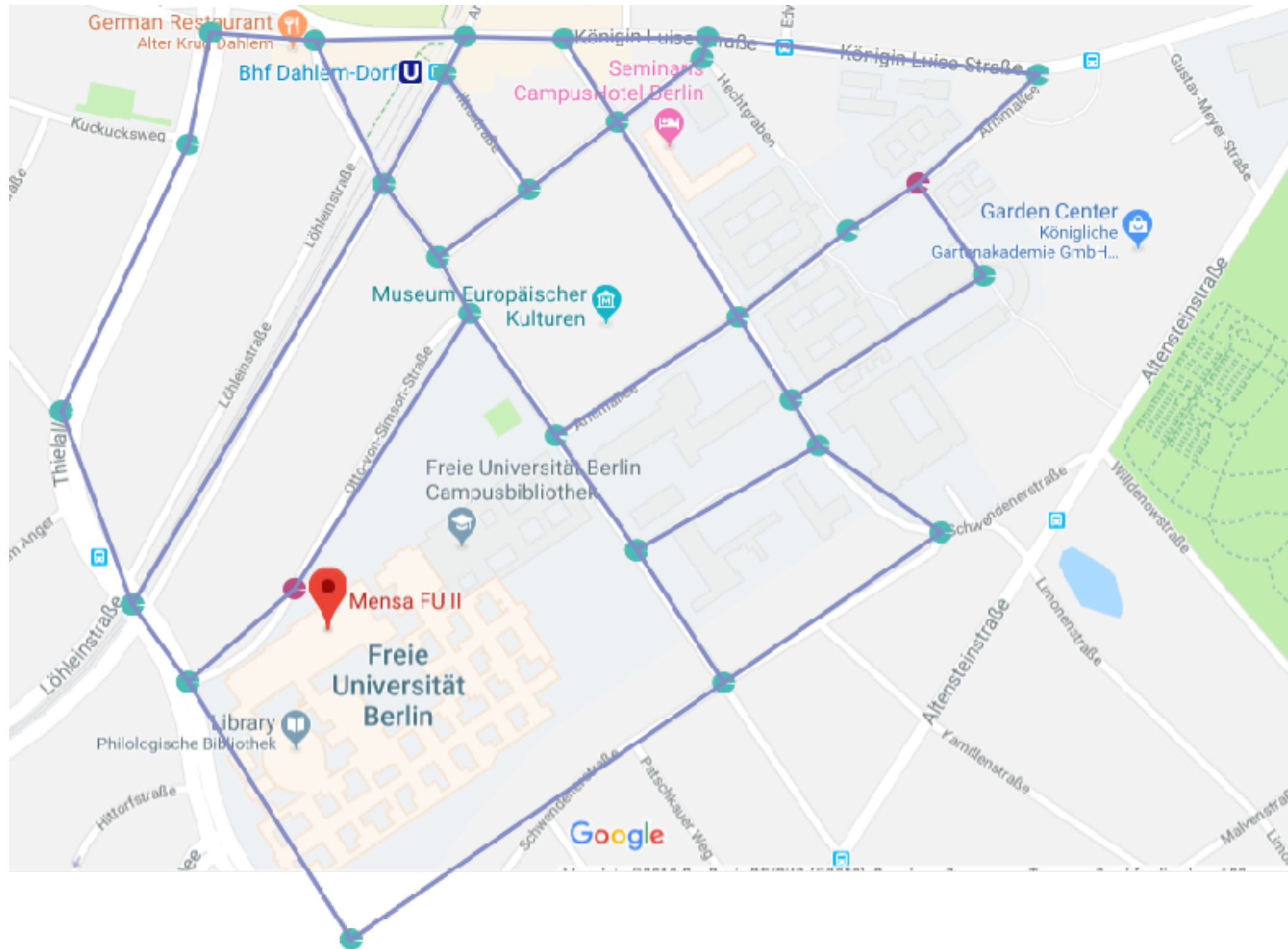
## Wie komme ich nachher in die Mensa?



## Lösung von Google

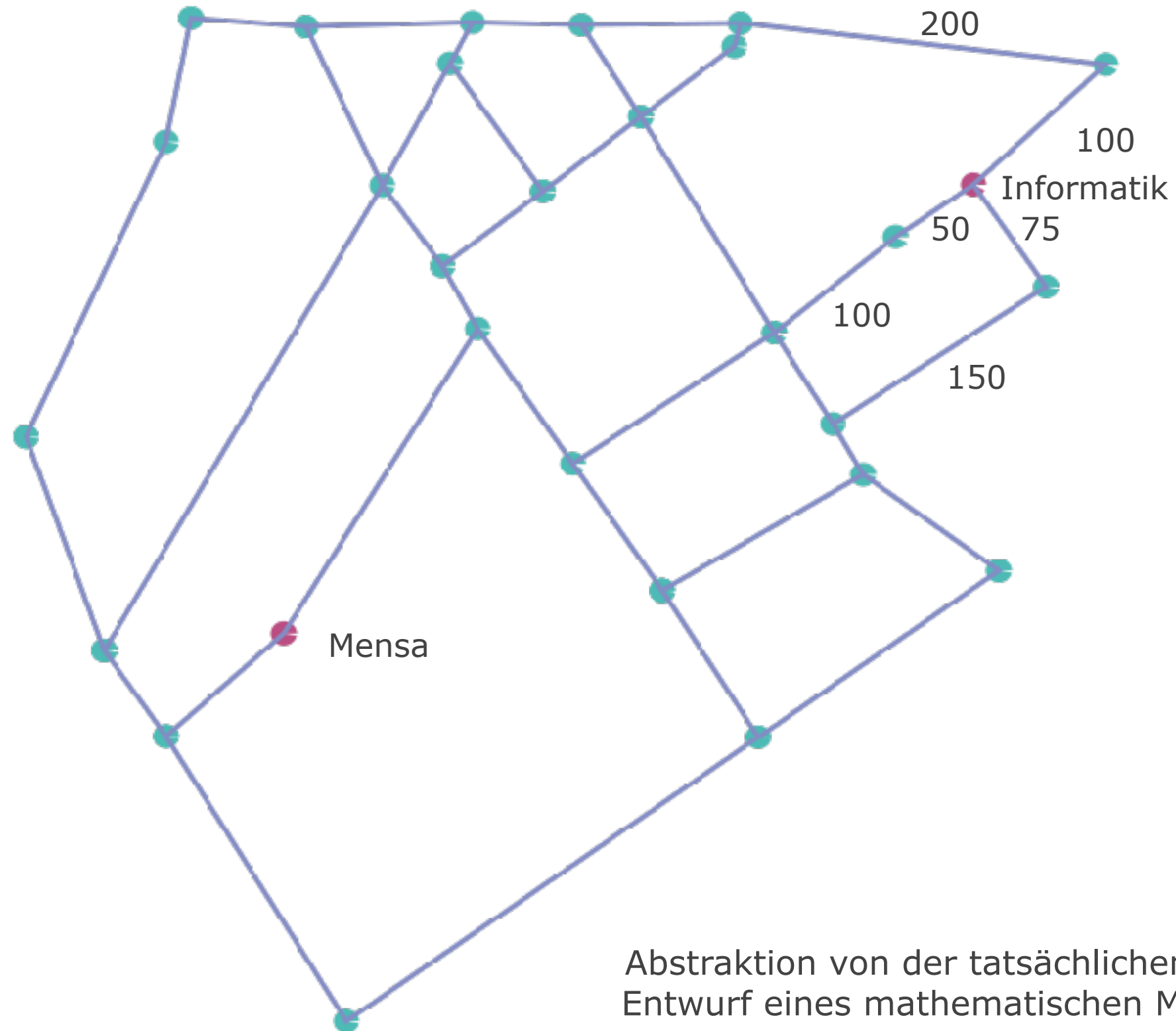


# Reiseproblem



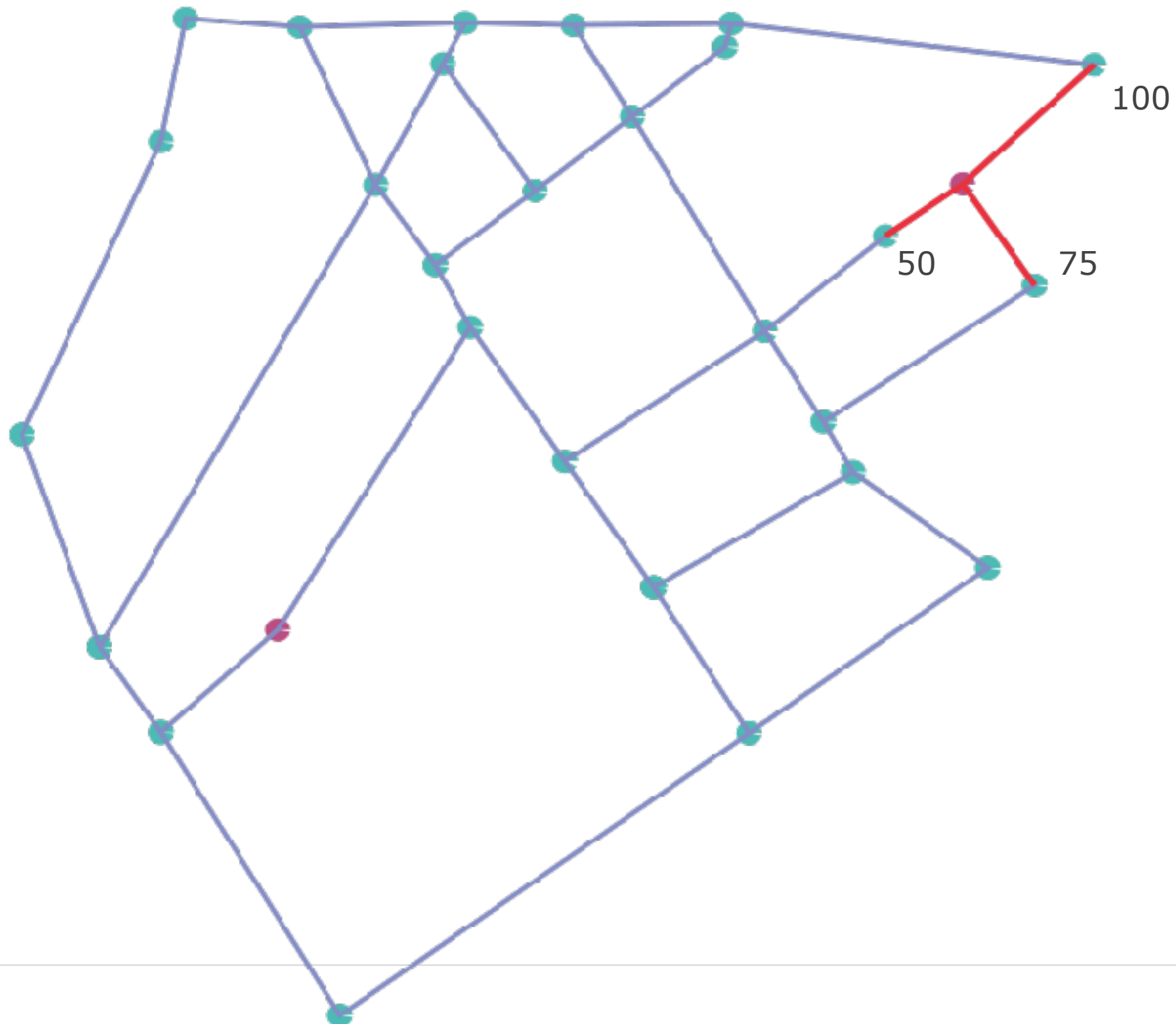


## Reiseproblem

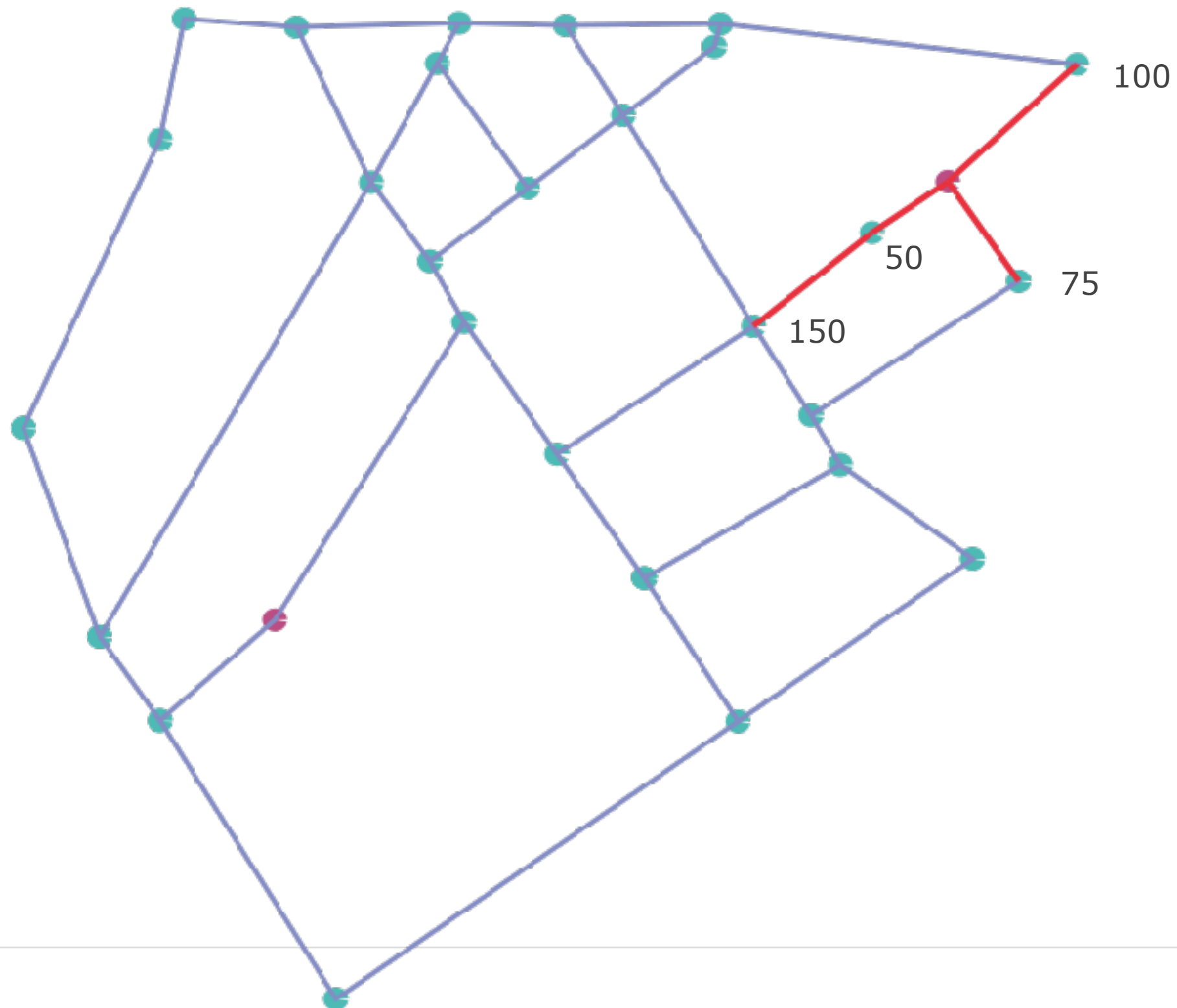


Abstraktion von der tatsächlichen Welt  
Entwurf eines mathematischen Modells

## Suche nach dem schnellsten Weg zur Mensa

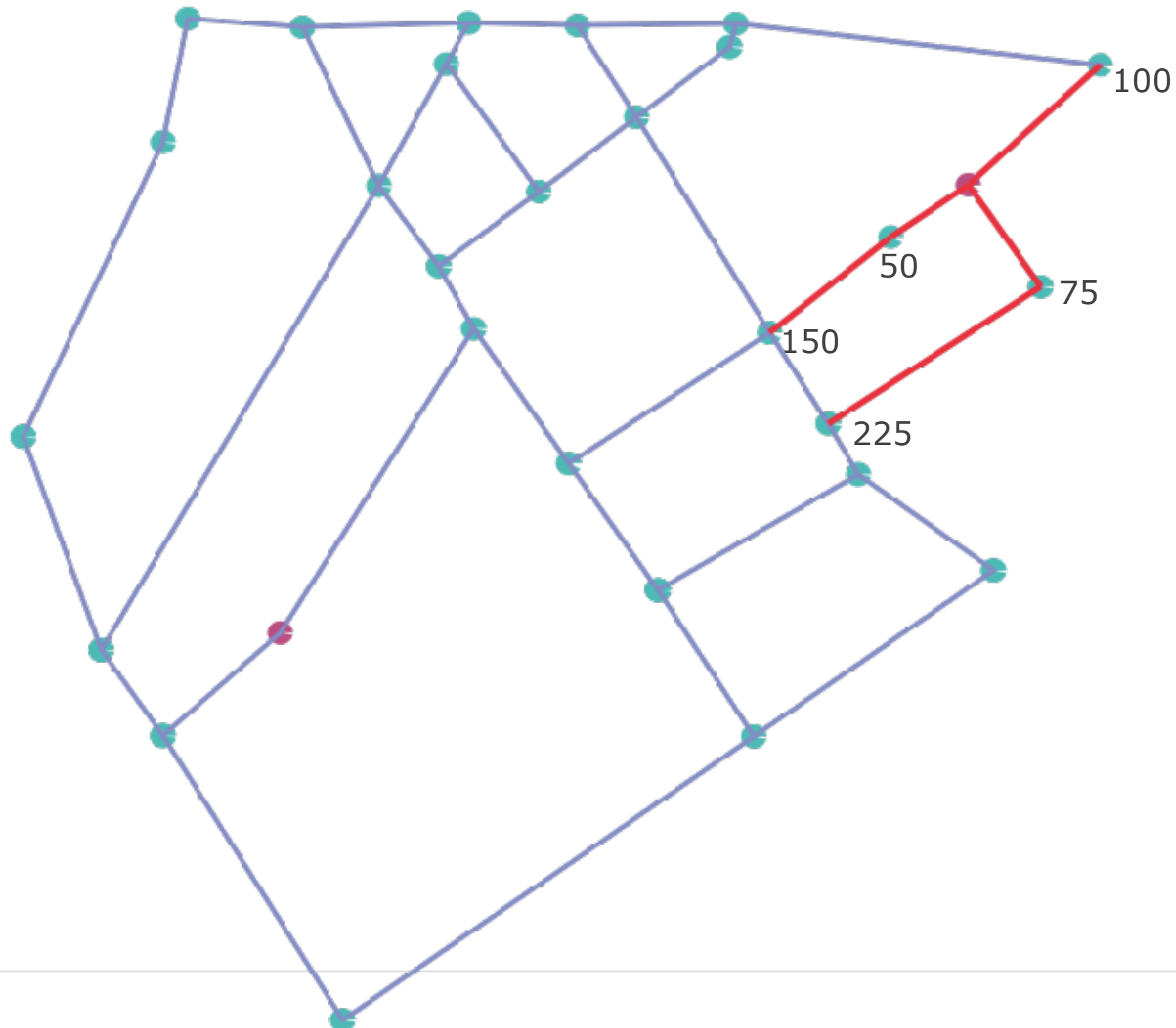


## Suche nach dem schnellsten Weg zur Mensa





## Suche nach dem schnellsten Weg zur Mensa



## Lösungsstrategie (informell):

1. Anfang: Startknoten hat die Distanz 0 und alle Knoten wurden noch nicht besucht
2. Gibt es noch nicht besuchte Knoten, wähle den mit der minimalen Distanz aus:
  1. Berechne die Distanz zu den Nachbarn
  2. Falls die Distanz zu Nachbarn kürzer ist als vorher, aktualisiere diesen

## Fragen:

1. Ist das verständlich und genau?
2. Ist das korrekt und allgemeingültig?
3. Wie effizient ist das?
4. Kann das auch ein Computer?

## Motto der Veranstaltung

- Interesse an Informatik bestärken
- Spaß und Interesse an Übung und Vorlesung
- Informatik an einer Universität kennenlernen
- Algorithmisches Denken schulen

Unsere Lernziele sind:

### **Programmieren im Kleinen**

Algorithmen entwerfen und implementieren

### **Grundkonzepte von imperativen Programmiersprachen beherrschen**

Datentypen, Anweisungen, Kontrollstrukturen

Unterprogramme, Funktionen, Parameterübergabe, Zeigertypen und Referenz-Typen

### **Korrektheit von Programmen analysieren**

Tests entwerfen und ausführen

### **Konzepte Objektorientierter Programmierung beherrschen**

Klassen, Objekte, Kapselung, Vererbung,

Polymorphie und Datenabstraktion

### **Theoretische Grundlagen kennen**

Registermaschinen als Berechenbarkeitsmodell und andere Berechnungsmodelle

Hoare-Kalkül zur Spezifikation / Verifikation

## 1. Teil

### Konzepte imperativer Programmierung

- Syntax und operationelle Semantik imperativer Programmiersprachen
- Python/Jython als Beispiel

## 2. Teil

### Formale Verfahren zur Spezifikation und Verifikation imperativer Programme

- Bedingungen im Zustandsraum (*assertions*)
- Hoare-Kalkül und partielle Korrektheit von Programmen
- Terminierung von Programmen

### Analyse von Laufzeit und Speicherbedarf

## **3. Teil**

**Konzepte objektorientierter Programmierung (Java)**

**Grundlegende Datenstrukturen und Datenabstraktion**

**Programmierungsmethodik**

**Berechenbarkeit**



## 3. Teil

### Konzepte objektorientierter Programmierung (Java)

- Primitive und Zusammengesetzte Datentypen,
- Methoden (Prozeduren und Funktionen), Parameterübergabe, Überladung
- Module, Klassen, Objekte
- Klassenhierarchien, Vererbung, abstrakte Klassen, Schnittstellen, Polymorphie
- Ausnahmebehandlung (Exceptions)

## 3. Teil

### Programmiermethodik

- Umwandlung von Rekursion in Iteration
- Teile und herrsche
- Rücksetzverfahren (*Backtracking*)
- Dynamische Programmierung

### Analyse von Laufzeit und Speicherbedarf

- O-Notation
- Analyse von Such- und Sortieralgorithmen
- Algorithmen und Datenstrukturen

### Berechenbarkeitsbegriff

- Universelle Registermaschinen

# Organisatorisches

## Ablaufplan

- 9:00 (s.t.!) bis 12 Uhr: Vorlesung mit Pausen
- Nach der Vorlesung wird der Übungszettel ins KVV gestellt
- 12:00 bis 14 Uhr: Mittagspause und selbstständiges Einarbeiten
- 14:00 bis 16 Uhr: Bearbeitung der Übungszettel im Tutorium
- Mittwochs **keine** Tutorien
- Mittwochsvorlesung dient zur Wiederholung und Vertiefung

## Räume

Vorlesung: Seminarraum 032 (Arnimallee 6)

Übungen:

1. Gruppe: Rechnerpoolraum K048 (Linux) (Takustraße 9)
2. Gruppe: Rechnerpoolraum K036 (Windows) (Takustraße 9)
3. Gruppe: Rechnerpoolraum 030 (Linux) (Arnimallee 6)

**Zuteilung erfolgt am Ende der Vorlesung!**

# Vorlesungen & Übungen

Grundlegende Regeln zum Erfolg sind:

1. Regelmäßige und aktive Teilnahme in der Vorlesung

**Fragen stellen!**

2. Regelmäßige und aktive Teilnahme an den Übungen

**Selbständige Lösung der Übungsblätter!**



## Kriterien für die Klausurteilnahme:

1. Eine **regelmäßige Teilnahme** an den Übungsterminen ist unerlässlich.
2. Jeder muss an **mindestens n-2 Übungsterminen anwesend** sein.
3. Jeder muss die täglichen Übungszettel bearbeiten und dem Tutor **vorstellen**.
4. Jeder Student muss **60% der maximal erreichbaren Punktzahl** aus allen abzugebenden Übungsblättern erreichen.
5. Abgabe der Übungszettel: Montags bis 8:55 im KVV
6. Bearbeitung eines kleinen Projekts in der fünften Woche

## Notenvergabe

Am Ende der Veranstaltung findet eine Klausur statt.

(Inhalt von Übungen und Vorlesungen sind relevant)

## Verbesserungsmöglichkeiten

- Zusätzlich ist es den Teilnehmern möglich durch besonders gute Projektarbeit die **Gesamtnote um maximal 0,7 zu verbessern**.
- Durch diese Verbesserung kann ein Bestehen der Klausur nicht erreicht werden.
- Die beste zu erreichende Note ist **1,0**.

Fragen?

Wir haben Fragen an euch...

... und wir gruppieren uns nach den Antworten.

1. Frage: Gehst du noch zur Schule oder schon in die Uni?  
Oder weder noch?

## 2. Hast du schon mal programmiert?



Kennst du Python?



Kennst du Java?



Wie viele Programmiersprachen hast du schonmal genutzt?

Ende der Einführung

## Warum Informatik studieren?

Pause  
(und danach geben wir dem Computer Befehle)



## Wir geben dem Computer Befehle!

```
Python 3.7.0 Shell
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> 40+2
42
>>> 10*2
20
>>> 9/3
3.0
>>> 42/5
8.4
>>> 12.5*12.5
156.25
>>>
```

Ln: 14 Col: 4

Interaktive Konsole mit Python-Interpreter

Erste Funktion: Taschenrechner

## Wir geben dem Computer Befehle!

```
>>> cakes = 2
>>> cakes + 2
4
>>> cups = cakes * 5
>>> cups
10
>>> cakse
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    cakse
NameError: name 'cakse' is not defined
>>>
```

Wir können uns Werte speichern

Unklare Befehle enden im Fehler

# Grundlegende Elemente von imperativen Programmen

- **Variablen**
- **Zuweisungen**
- **Ausdrücke**
- Definitionen von Datentypen
- Deklarationen von Variablen unter Verwendung vordefinierter Datentypen
- Anweisungen für den Kontrollfluss innerhalb des Programms
- Gültigkeitsbereich von Variablen (*locality of reference*)
- Definition von Prozeduren und Funktionen

# Variablen

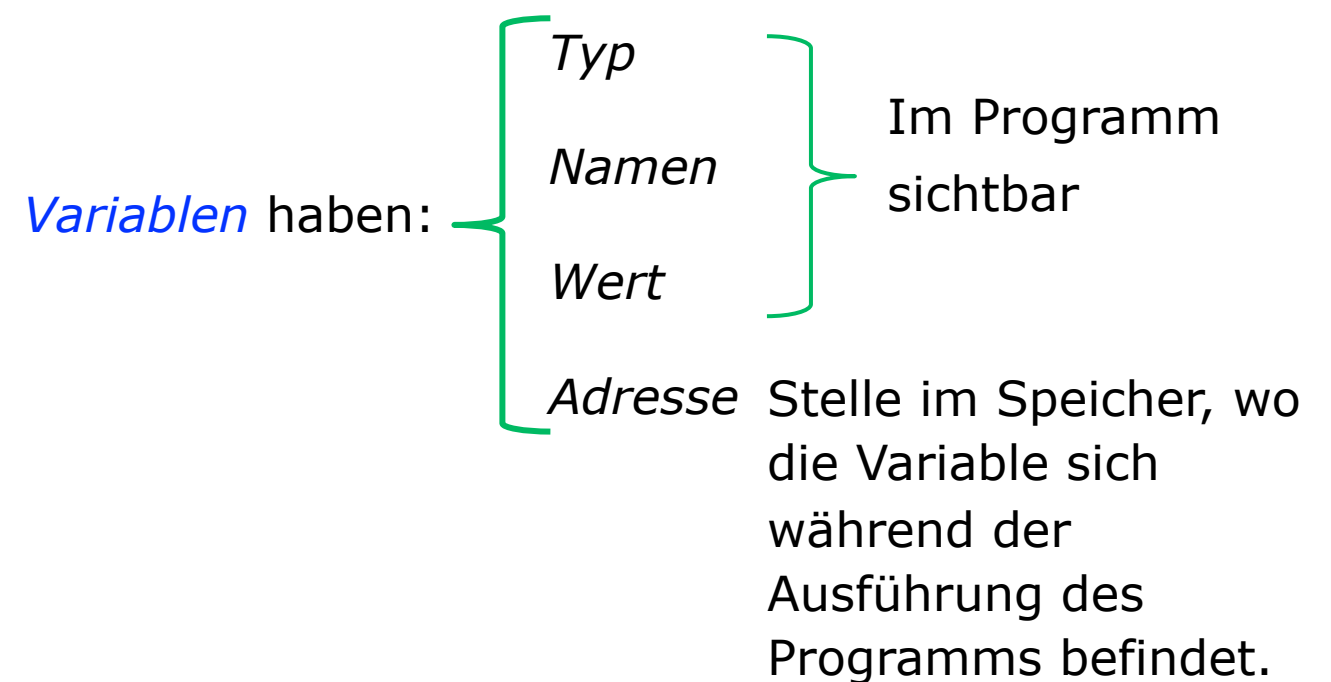
## Funktionale Programmiersprachen

Eine Variable stellt nur den symbolischen Namen von einem Wert oder einem Ausdruck, der zu einem Wert ausgewertet werden kann, dar.

**Der Wert einer Variablen kann nicht verändert werden.**

## Imperative Programmiersprachen

*Variablen* sind Stellen im Speicher, in denen Werte abgelegt werden können. Variablen speichern Zustände.



*Variablen* müssen normalerweise vor der erstmaligen Benutzung deklariert werden.

# In Python müssen Variable nicht deklariert werden

[illegible]

prog\_name.py

# Interpreter

>>>

```
100 99999999999999999999999999999999 True 3.4 hello
```

# Zuweisungen

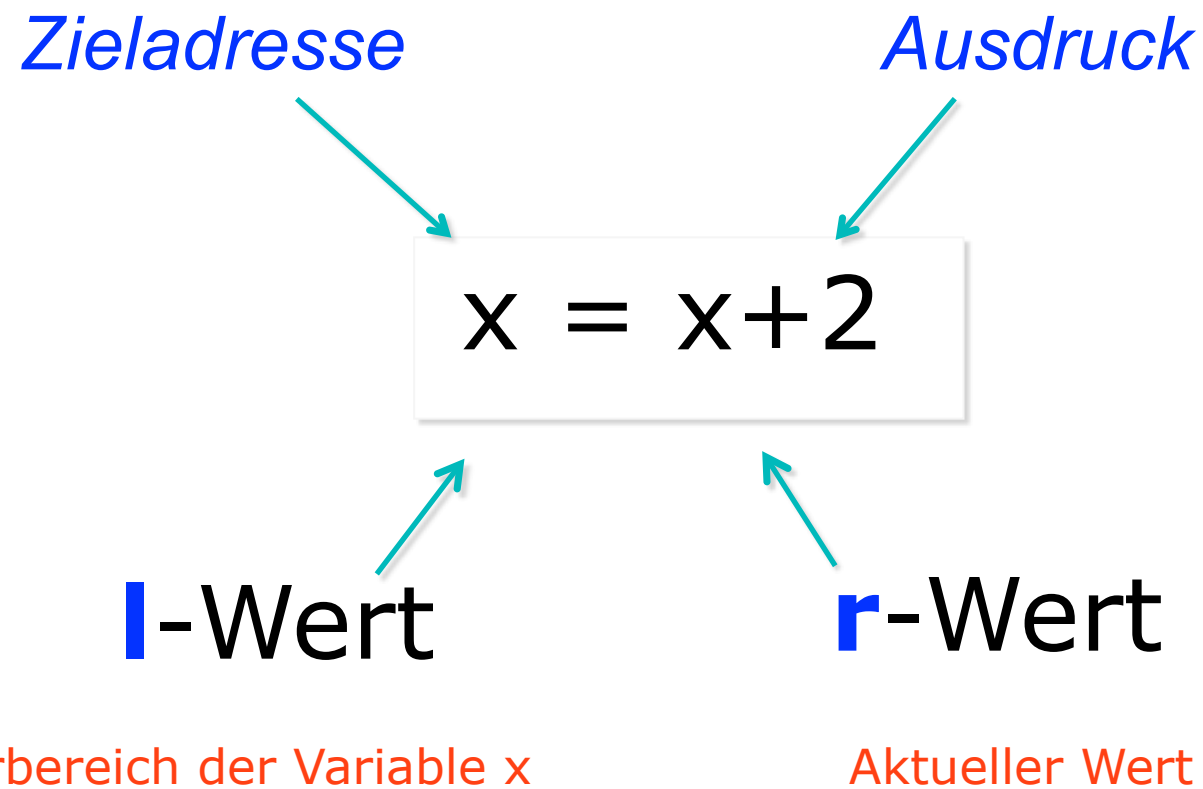


Zuweisungen sind destruktive Anweisungen.

Die Adresse mit dem symbolischen Namen **A** wird mit dem Wert des Ausdrucks **B** überschrieben.

Beispiel: `x = 4`

# Zuweisungen



Zwei verschiedene semantische Bedeutungen des gleichen Symbols. ☹

# Ausdrücke

Ein Ausdruck wird in der Informatik als eine Berechnungsvorschrift verstanden, die einen eindeutigen Wert darstellt.

Die einfachsten Ausdrücke in Python sind Konstanten beliebiger Datentypen

Beispiel: **1**    **3.4**    **"hello"**    (**Literale**)

Ausdrücke haben keinen **!**-Wert und können nicht auf der linken Seite einer Anweisung stehen.

Beispiele:

**a+b = c**    oder    **3 = c**



# Arithmetische Operatoren

## Operator

## Beschreibung

<b>+</b>	unär	ändert nichts an der Zahl
<b>-</b>	unär	ändert das Vorzeichen der Zahl
<b>**</b>	binär	Power-Operator
<b>*</b>	binär	Multiplikation
<b>+</b>	binär	Addition
<b>-</b>	binär	Subtraktion
<b>/</b>	binär	Division
<b>%</b>	binär	Modulo (Restbildung)
<b>//</b>	binär	Division (immer ganzzahlig)

## Typsystem in Python

```
>>> num = 5
>>> type(num)
<class 'int'>
>>> pi = 3.14
>>> type(pi)
<class 'float'>
>>> x = num * pi
>>> type(x)
<class 'float'>
>>> equal = num == pi
>>> equal
False
>>> type(equal)
<class 'bool'>
>>> num + equal
5
>>> equal = num == num
>>> equal
True
>>> num + equal
6
>>>
```

Es gibt verschiedene Typen von Zahlen und noch andere Typen

Mit == wird auf Gleichheit geprüft

num == pi, d.h. ist 5 gleich 3.14?

Warum ist num + equal erst 5 und später 6?

# Datentypen in Python

Datentyp	Beispiel	Bemerkungen
int	a = 3	
float	x = 1.0	
boolean	w = True	
Complex	c = 2 + 3j	
String	t = "Text" oder t = 'Text'	nicht veränderbar
Liste	l = [ 5, 3, 1, 6, 8 ]	veränderbar
Tuple	p = ( 35, 0, "Name")	nicht veränderbar
Dictionary	d = { 1:'a', 2:'b', 3:'c'}	veränderbar

## Datentyp einer Variablen

- \* Python hat **dynamische Datentypen**, ist aber streng typisiert.
- \* Der Datentyp einer Variable wird erst zur Laufzeit festgelegt.
- \* Im Gegensatz zur statischen Typisierung, bei der der Datentyp einer Variable explizit deklariert werden muss, wird der Typ einer Variablen aus dem Typ des Werts **zur Laufzeit abgeleitet**.
- \* **Quelle einiger schwierig zu findender Fehler.**
- \* Streng typisiert.

# Datentyp von Variablen

[illegible]

Eine Variable bekommt erst eine Speicheradresse, nachdem diese zum ersten mal einen Wert zugewiesen bekommt und **der Datentyp wird aus dem zugewiesenen Wert abgeleitet.**

## Ausgabe:

```
>>>
    <class 'int'>
    <class 'long'>
    <class 'float'>
    <class 'complex'>
>>>
```

# Dynamisches Typsystem

Python Virtuelle Maschine **PVM**

**y = 10.0**

**a = 10**

**b = 30**

**sum = a+b**

**mult = a\*y**

<b>y</b>	<b>10.0</b>
<b>a</b>	<b>10</b>
<b>b</b>	<b>30</b>
<b>sum</b>	<b>40</b>
<b>mult</b>	<b>100.0</b>

**Speicher**

# Ausdrücke haben einen **Wert** und einen **Datentyp**

Beispiele:

	Ausdruck	Wert	Type
type casting	<b>int</b> ("12345")	12345	<b>int</b>
	<b>int</b> (0xff)	255	<b>int</b>
	<b>complex</b> (3.14)	(3.14+0j)	<b>complex</b>
	(3*11//5)	6	<b>int</b>
	8%2	0	<b>int</b>
	8/3	2.66666666..	<b>float</b>
	<b>float</b> (3)	3.0	<b>float</b>
	2**(2+1)	8	<b>int</b>
	<b>pow</b> (2,3)	8	<b>int</b>

# Höhere Datenstrukturen

Python unterstützt drei höhere Datentypen

**Listen** (dynamic arrays)

**Tuples** (immutable lists)

**Dictionaries** (hash tables)



# Listen

Eine *Liste* ist eine **veränderbare** Sammlung von Objekten verschiedener Datentypen.

Beispiele:

**a = []** **# Leere Liste**

**b = [1, 3.5, "Zeichenkette", 10]**

**c = [ 'Hi', 4, 0.234 ]**

# Zeichenketten-, Listen- und Tupel-Operatoren

Operator	Funktion	Beispiel	Wert
<b>+</b>	Verkettung	"Eins" + " Zwei"	'Eins Zwei'
<b>*</b>	Wiederholung	2 * "Eins"	'EinsEins'
<b>in</b>	enthalten in	1 in [1,2]	True
<b>not in</b>	nicht enthalten in	3 not in [1,2]	True
<b>x[i]</b>	Indizierung, liefert das Element aus x an der Position i	x = "hallo" x[1]	a
<b>x[i:j]</b>	liefert aus x die Elemente von i bis j-1 zurück	x = "hallo" x[0:3]	'hal'

Beispiele:

```

>>> a = [1, 2, 3, 4]

```

```

>>> a[2]

```

```

3

```

```

>>> a[2] = 10

```

```

>>> a

```

```

[1, 2, 10, 4]

```

```

>>> b = a

```

```

>>> b

```

```

[1, 2, 10, 4]

```

```

>>> a[1] = 2000

```

```

>>> b

```

```

[1, 2000, 10, 4]

```

```

>>> a = [7, 6, 5, 4]

```

```

>>> a

```

```

[7, 6, 5, 4]

```

```

>>> b

```

```

[1, 2000, 10, 4]


```

Hier wird nur eine  
Speicheradresse  
kopiert.

Hier wird eine neue  
Liste erzeugt.

**b** zeigt aber auf die  
alte Liste.

Beispiele:

```
          -5 -4 -3 -2 -1  
          0  1  2  3  4  
>>> a = [1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
>>> a[2:4]  
[3, 4]  
>>> b = a[:]   
>>> b  
[1, 2, 3, 4, 5]  
>>> a[1] = 2000  
>>> b  
[1, 2, 3, 4, 5]  
>>> a  
[1, 2000, 3, 4, 5]  
>>> a[-1]  
5  
>>> a[-2]  
4
```

Hier wird die Liste  
vollständig kopiert.

# Tupel

Ein *Tupel* ist eine Sammlung von Objekten verschiedener Datentypen zu einem Objekt.

Im Gegensatz zu einer Liste ist die Folge der Elemente dabei **nicht veränderbar**.

Beispiele:	<code>()</code>	<b># Leeres Tupel.</b>
	<code>(1,)</code>	<b># Tupel mit einem Element.</b>
	<code>(1,2,3)</code>	<b># Tupel mit drei Elementen.</b>
	<code>(1, 'Eins')</code>	<b># Tupel mit zwei Elementen.</b>
	<code>number = (1, 'eins')</code>	

## Beispiele:

```
>>> a = (1,2,3)
```

```
>>> b = a
```

```
>>> b
```

```
(1, 2, 3)
```

```
>>> a[2]
```

```
3
```

```
>>> len(a)
```

```
3
```

```
>>> a == b
```

```
True
```

```
>>> a[0] = 5
```

Traceback (most recent call last):

File "<pyshell#76>", line 1, in <module>

a[0] = 5

TypeError: 'tuple' object does not support  
item assignment

```
>>> a = (5,6,7)
```

```
>>> b
```

```
(1, 2, 3)
```

```
>>> a+b
```

```
(5, 6, 7, 1, 2, 3)
```

```
>>> a = a+a
```

```
>>> a
```

```
(5, 6, 7, 5, 6, 7)
```

```
>>> b
```

```
(1, 2, 3)
```

```
>>> a*3
```

```
(5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7)
```

# Höhere Datenstrukturen

**Dictionaries** *Dictionaries* sind eine Sammlung von Schlüssel- und Wertpaaren.

Ein Dictionary ist also eine Liste aus Schlüsseln (*keys*), denen jeweils ein Wert (*value*) zugewiesen ist.

**Beispiele:** `{}` *# Leeres Wörterbuch (Dictionary)*

`{ 1:'Goethe', 2:'Schiller', 3:5.67 }`

`atomic_num = {'None' : 0, 'H' : 1, 'He' : 2}`

# Höhere Datenstrukturen

## Dictionaries

### Beispiele:

```
>>> synonyms = {}
>>> synonyms['pretty'] = 'beautiful'
>>> synonyms['shy'] = 'timid'
>>> synonyms['easy'] = 'facile'
>>> synonyms
{'shy': 'timid', 'easy': 'facile', 'pretty': 'beautiful'}
>>> synonyms['easy']
'facile'
>>> 'pretty' in synonyms
>>> True
```



# Höhere Datenstrukturen

## Dictionaries

- Beliebige Datentypen können kombiniert werden.
- Sehr effizient implementiert mit Hilfe von Hashtabellen.

### Beispiele:

```
>>> dic = {}  
>>> dic[1] = 'Hi'  
>>> dic['a'] = 'Hallo'  
>>> dic[3.1416] = 'pi'  
>>> dic[(1,2,3,4)] = 'Reihe'  
>>> dic  
{'a': 'Hallo', 1: 'Hi', (1, 2, 3, 4): 'Reihe', 3.1416: 'pi'}  
>>>
```

## weitere Operationen ...

```
>>> help ( int )
```

Methods defined here:

```
|  __abs__(...)
|      x.__abs__() <==> abs(x)
|  __add__(...)
|      x.__add__(y) <==> x+y
|  __and__(...)
|      x.__and__(y) <==> x&y
|  __bool__(...)
|      x.__bool__() <==> x != 0
|  __ceil__(...)
|      Ceiling of an Integral returns
|      itself.
|  __divmod__(...)
|      x.__divmod__(y) <==>
|      divmod(x, y)
|  __eq__(...)
|      x.__eq__(y) <==> x==y
|  __float__(...)
|      x.__float__() <==> float(x)
```

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.gcd(56,14)
14
>>> math.cos(3.0)
-0.9899924966004454
>>> math.factorial(33)
8683317618811886495518194401280000000
>>> help(math)
Help on module math:
NAME  math
MODULE REFERENCE
    https://docs.python.org/3.6/library/math
    The following documentation . . .
DESCRIPTION
    This module is always available. It provides . . .
FUNCTIONS
    acos(x) . . .
    . . .
```

# Kommentare in Python

```
""" Blockkommentare:
```

```
    können sich über mehrere Zeilen  
    erstrecken.
```

```
"""
```

```
# von hier aus bis Ende der Zeile wird dieser Text ignoriert
```

```
a = 4  # Kommentar
```

# Import-Anweisung

Die **import**-Anweisung ermöglicht in Python-Skripten den Zugang auf vorprogrammierte Module

Einige interessante Module sind:

<b>math:</b>	exp, sin, sqrt, pow
<b>string:</b>	digits, whitespace
<b>sys:</b>	stdin, stderr, argv
<b>os:</b>	system, path
<b>re:</b>	split, match

## Einrücken anstatt Klammern

- kein *begin* ... *end* wie in Pascal oder `{ ... }` wie in C
- Die Anweisungen innerhalb eines Blocks beginnen immer an der gleichen Zeilenspalte

Beispiel:

```
x = int ( input() )  
if x <= 0:  
    x = 0  
    print ('zero')  
else:  
    x = 1  
    print ('one')
```

# Was ist Python?

- eine **Script-Sprache**
- Anfang der **90er** Jahre entwickelt.
- Erfinder: Guido van Rossum an der **Universität von Amsterdam**
- Unterstützung des **strukturierten Programmierens** aus der **ABC-Sprache** übernommen.

ALGOL → ABC → Python

- **Philosophie:**
  - Simplizität, Lesbarkeit und Orthogonalität
  - **Schnelle Programmentwicklung** ist **wichtiger** als **schnelle Programme**

# Lesbarkeit

## Pseudocode

```
quicksort( A, p, r )  
  if p < r  
    then q ← partition( A, p, r )  
         quicksort( A, p, q-1 )  
         quicksort( A, q+1, r )  
  
partition( A, p, r )  
  x ← A[r]  
  i ← p-1  
  for j ← p to r-1  
    do if A[j] ≤ x  
       then i ← i+1  
          exchange A[i] ↔ A[j]  
  exchange A[i+1] ↔ A[r]  
  return i+1
```

## Python

```
def quicksort( A, p, r ):  
    if p < r:  
        q = partition( A, p, r )  
        quicksort( A, p, q-1 )  
        quicksort( A, q+1, r )  
  
def partition( A, p, r ):  
    x = A[r]  
    i = p-1  
    for j in range( p, r ):  
        if A[j] ≤ x:  
            i = i+1  
            exchange( A, i, j )  
    exchange( A, i+1, r )  
    return i+1
```

# Python unterstützt mehrere Paradigmen

- flexibel in der Handhabung verschiedener Programmier-Paradigmen
  - \* **Imperative Programmierkonzepte**
  - \* Objektorientierte Programmierung
  - \* Funktionale Programmierung
  - \* Aspektorientierte Programmierung
- einfaches Einsetzen verschiedener Programmier-Techniken
  - \* Strukturierte Programmierung
  - \* Entwurf gemäß Vertrag (DBC)



# Python-Interpreter

## Linux und Mac OS



Der Python-Interpreter  
ist Teil der Standard-  
Installation.



python

## Windows und Mac OS



IDLE

Integrierte DeveLopment Environment  
[www.python.org/idle](http://www.python.org/idle)

python 3.6.5

## Warum Python/Jython?

- Einfache Syntax und Semantik
- Einfache Programmierumgebung
- Hybride Programmiersprache (Multi-Paradigma)
- Ermöglicht zuerst nur das rein imperative Programmieren
- Höhere Datenstrukturen sind in der Sprache integriert
- Plattformunabhängig
- In der realen Welt verwendete Sprache
- Umfangreiche Standardbibliothek

# Welche Sprache ist besser?

Es gibt noch keine Programmiersprache, die für alle Anwendungen die beste ist.

- Java **sehr beliebt auf Grund der JVM**
- C eignet sich besonders gut für die **hardwarenahe Programmierung**
- SQL *Structure Query Language* ist eine **Datenbanksprache**
- Python **für die effiziente Entwicklung von Prototypen**
- Matlab **für numerische Berechnungen (Matrizen und Vektor-Rechnungen)**
- Elixir **für die Programmierung von nicht sequentiellen Prozessen**
- **USW.**

Es gibt keine beste Sprache

## Python:

- Invent your own computer games with python, Al Sweigart
- <https://www.python.org/>
- <https://wiki.python.org/moin/GermanPythonBooks>

## Java:

- Java-Intensivkurs, Marco Block
- <https://java.com/>

## Algorithmen:

- The Art of Computer Programming, Donald Knuth
- Data Structures and Algorithms in Java, Goodrich & Tamassia

## Zuweisung in Übungsgruppen & Fragen