Informatik II — *Max Schaldach*

1. Variablen

1.1. Zahlenwerte

Ein **Integer** ist eine positive oder negative ganze Zahl

- Bei der **Integerdivision** `//` werden die Nachkommastellen des Ergebnisses abgeschnitten
- Der **Modulo** `%` gibt den Restwert einer Integer-Division zurück
- Für Potenzrechnungen wird der Operator `**` verwendet

Ein **Float** ist eine Zahl mit Gleitkomma-Präzision. Floats können gerundet werden:

```
math.floor(3.2)  # 3
math.ceil(3.2)  # 4
```

Der Pseudowert `None` wird verwendet, um eine Variable zu kennzeichnen, die keinen Wert besitzt:

```
var = None
```

1.2. Strings

Strings sind Zeichenfolgen und werden mit Anführungszeichen initialisiert:

```
var = "string"  # äquivalent zu 'string'
var = """string"""  # wenn string mehrzeilig ist
```

Ein Zeichen kann über seinen Index abgerufen werden:

```
var[0]  # ruft erstes Element des Strings auf
var[1] = 'd'  # ERROR, Strings sind nicht veränderlich
```

Ein String kann durch Kombination von anderen Strings und numerischen Werten initialisiert werden:

```
var = f"{2} und 1"  # äquivalent zu "2 und 1"
```

Strings können mit Hilfe von `+` aneinandergehängt werden:

```
var = "…" + "…" + "…"
```

Die Länge eines Strings kann wie folgt bestimmt werden:

```
len("string")  # gibt 6 zurück
```

Strings können an bestimmten Zeichen getrennt werden:

```
var = "  You, rock"
var = var.split(',')  # ['  You','rock']
var = var.strip()  # 'You, rock'
```

1.3. Boolesche Ausdrücke

Boolesche Werte sind entweder `True` / `1` oder `False` / `0`

```
var = True  # äquivalent zu var = 1
```

Vergleichsoperatoren (`==`, `!=`, `>`, `<`, `>=`, `<=`) ermöglichen den Vergleich von Werten und liefern ein boolesches Ergebnis

Logische Operatoren kombinieren boolesche Ausdrücken:

```
False and …  # gibt unabhängig von … False zurück
True or …  # gibt unabhängig von … True zurück
not …  # invertiert Aussage
```

1.4. Umwandlung

Mit `type(…)` kann der Datentyp einer Variable abgefragt werden. Zur Konvertierung von Datentypen gibt es Helferfunktionen:

```
var = int("3")  # var = 3
var = str(2)  # var = '2'
var = float(3)  # var = 3.0
var = bool(6)  # var = True, denn alle Zahlenwerte ≠0
werden in True umgewandelt
```

1.6. Zeiger

In Python sind alle Variablen **Zeiger**. Das heisst, sie verweisen auf eine Speicheradresse, an denen ihr zugrundeliegendes Objekt abgespeichert ist

Wenn **unveränderbare Variablen** dupliziert werden (z.B. ein Integer, Float oder String), wird ein neues Objekt mit dem gleichen Wert erstellt:

```
var = 3
duplicate = var  # duplicate zeigt auf ein neues
Objekt mit dem gleichen Wert wie var
```

Wenn **veränderbare Variablen** dupliziert werden (z.B. eine Liste oder ein Dictionary), zeigen die Kopie und das Original auf dasselbe Objekt. Änderungen, die an einem der beiden vorgenommen werden, wirken sich auf das andere aus:

```
lis = [1, 2, 3, 4]
duplicate = lis  # zeigt auf dasselbe Objekt wie lis
```

Mit `copy()` wird eine neue Liste erstellt, die die gleichen Elemente wie die Originalliste enthält:

```
import copy
duplicate = lis.copy()  # zeigt auf ein neues Objekt
mit einer Kopie von lis
```

Wenn Unterlisten vorhanden sind, teilen sich die Original- und die Kopieliste jedoch immer noch dieselben Unterlisten. Um Unterobjekte zu kopieren, wird `deepcopy()` verwendet:

```
sublis = [1, 2]
lis = [sublis, 3, 4]
duplicate = lis.deepcopy(var)  # alle Elemente von
lis, auch Unterlisten, sind Kopien
```

1.7. User Input & Output

Mit `input(…)` wird der User zur Eingabe aufgefordert:

```
name = input("prompt")
```

wobei `prompt` ein String ist und in der Konsole erscheint

Achtung: Inputs werden immer als Strings abgespeichert

Mit `print(…)` können Werte in der Konsole ausgegeben werden:

```
print("Hello World")  # "Hello World"
print("Hello", "World")  # äquivalent zu oben
print("Hello" + "World")  # "HelloWorld"
```

Mit `print(…, end = " ")` können mehrere Ausdrücke in einer Zeile gedruckt werden

2. Container

2.1. Listen

Listen sind Sammlungen von Elementen (Datentypen sind variabel):

```
lis = ['string1',2,5.0,'string2']
lis = [0] * 5  # [0,0,0,0,0]
```

Listenelemente können durch Indexierung aufgerufen werden:

```
lis[0]  # ruft erstes Element der Liste auf
lis[-1]  # ruft letztes Element der Liste auf
```

Listen/Strings können auf ihren Inhalt überprüft werden:

```
… in lis  # True, wenn der Wert in der Liste vorkommt
```

Listen/Strings können mit **Slicing** `[start:stop:step]` geteilt werden:

```
lis = [2,1,2,3,7]
slice = lis[:2]  # [2,1]
slice = lis[1:-1]  # [1,2,3]
slice = lis[3:1:-1]  # [3,2]
slice = lis[-5:-1:1]  # [2,1,2,3]
slice = lis[-1:-6:-1]  # [7,3,2,1,2]
lis[3:6] = lis[2:5]  # lis = [2,1,2,2,3,7]
```

Wenn `start` oder `stop` ausserhalb des gültigen Bereichs der Liste liegen, werden die ungültigen Indizes ignoriert

`range(start,stop,step)` gibt einen Wertebereich zurück:

```
list(range(1,5)) # [1,2,3,4]
list(range(1,9,2)) # [1,3,5,7]
list(range(-5,0)) # [-5,-4,-3,-2,-1]
list(range(5,-1,-1)) # [5,4,3,2,1,0]
```

Es stehen eingebaute Hilfsfunktionen für Listen zur Verfügung:

```
len(lis)  # gibt Anzahl Elemente zurück
sum(lis)  # gibt Summe aller Elemente zurück
lis.append(element)  # fügt Element am Ende der Liste an
del lis[i]  # entfernt Element bei Index i
idx, vals = enumerate(lis)  # gibt die Werte und die
zugehörigen Indizes als separate Listen zurück
list(zip(lis1, lis2))  # kombiniert zwei Listen
```

Folgend zwei Beispiele zu den obigen Hilfsfunktionen:

```
list(zip([1,2], [3,4]))  # [(1,3),(2,4)]
for idx, val in enumerate([1,2,3,4]):
  print(idx, val)   # 0 1  1 2  2 3  3 4
```

Listen können auf unterschiedliche Weisen umgekehrt werden:

```
new_lis = lis[::-1]  # Rückgabewert ist die Liste
lis.reverse()  # besitzt keinen Rückgabewert
list(reversed(lis))  # reversed(…) gibt Iterator zurück
```

Mit **Listen Komprehension** kann man Listen kompakt erstellen:

```
[x**3 for x in range(10) if x > 5]
[x if x > y else y for (x, y) in zip(lis1, lis2)]
```

2.1. Tuples & Sets

Ein **Tuple** ist eine Liste, deren Werte unveränderlich sind:

```
tuple = ('string1',3,5.0,'string2')
tuple[1] = 3  # ERROR, Tuples sind nicht veränderlich
tuple([3,2,4])  # (3,2,4)
```

Ein **Set** ist eine Sammlung eindeutiger Elemente, bei der doppelte Elemente automatisch entfernt werden:

```
set = {1,2,3,True}  # 1 und True sind doppelt
set.add(5)  # fügt 5 zu set hinzu
set.remove(2)  # entfernt 2 aus set
set[0]  # ERROR, Elemente von Sets sind nicht aufrufbar
set([3,3,4,2])  # {3,4,2}, 3 kommt nur noch einfach vor
```

2.2. NumPy

NumPy ist eine Bibliothek für wissenschaftliches Rechnen, die multidimensionale Arrays beinhaltet:

```
import numpy as np
mat = np.array( [[1,2,3],[4,5,6]])  # erstellt Matrix
mat[0][1]  # 2
mat[0:2, 0:2]  # [[1,2], [4,5]]
mat[-2:2, ::2]  # [[1,3], [4,6]]
mat.size  # gibt Anzahl der Elemente zurück
```

Arrays sind Listen für die spezielle Operationen definiert sind:

```
arr = np.array([[1,2,3,7,8,9]])
arr.sum()  # gibt Summe aller Elemente zurück
arr.min()  # gibt kleinstes Element zurück
arr.max()  # gibt grösstes Element zurück
np.mean(arr)  # gibt Mittelwert zurück
np.std(arr)  # gibt Standardabweichung zurück
np.sort(arr)  # sortiert Array
```

NumPy enthält Funktionen zur Erstellung spezieller Arrays:

```
arr = np.zeros(5)  # [0,0,0,0,0]
arr = np.arange(5)  # [0,1,2,3,4]
np.linspace(5,15,3)  # [5,10,15]
np.random.randint(3)  # 3 Int-Zufallszahlen in [0,1]
np.random.randint(1,7,3)  # 3 Int-Zufallszahlen in [1,7]
np.random.random(3)  # 3 Float-Zufallszahlen in [0,1)
np.random.uniform(1,2,3)  # 3 Float-Zufallszahlen in
[1,2]
arr[arr % 2 == 0]  # [0,2,4]
```

Mit NumPy sind elementweise Operationen möglich:

```
mat + 1  # jedes Element wird iteriert
mat * 2  # jedes Element wird mit 2 multipliziert
mat1 ** 2  # jedes Element wird quadriert
mat1 * mat2  # Matrix-Elemente werden multipliziert
mat1 @ mat2  # Matrixmultiplikation, wobei Matrix-
Dimensionen valide Operation ergeben müssen
```

Achtung: Der logische Operator `or` entspricht in NumPy `|`

2.3. Dictionaries

Dictionaries speichern Schlüssel-Werte-Paare:

```
dic = {"key1": 1, "key2": 2}
```

Operationen für Dictionaries sind ähnlich wie jene für Listen:

```
dic["key1"]  # gibt den Wert von key1 aus
dic["key3"] = 3  # initialisiert key3 mit Wert 3
del dic["key2"]  # löscht Schlüssel-Werte-Paar
… in dic  # True, wenn Schlüssel in dic vorkommt
len(dic)  # gibt Anzahl Schlüssel-Werte-Paare zurück
```

Listen und Tuples lassen sich in ein Dictionary umwandeln:

```
lis = [["one",1],["two",2]]
dic = dict(lis)  # {"one": 1, "two": 2}
```

Bei Listen unterschiedlicher Länge, wird der Überhang ignoriert:

```
dic = dict(zip(["one","two"],[1,2,3]))
# {"one": 1, "two": 2}
```

Schlüssel und Werte können separat ausgelesen werden:

```
keys = list(dic.keys())  # ["one","two"]
values = list(dic.values())  # [1,2]
items = list(dic.items())  # [("one",1),("two",2)]
```

Mit **Dict Komprehension** kann man Dictionaries kompakt erstellen:

```
{2 * x : 3 * y for x,y in enumerate(list)}
{key : dic[key] for key in dic if dic[key] > 1}
{x : {z : y for z in range(5)} for x, y in
enumerate(list)}  # verschachteltes Dictionary
```

3. Codeblöcke

3.1. Anweisungen

Ein `if-else`-**Anweisung** führt eine Anweisung (`statement`) aus, wenn eine Bedingung (`condition`) erfüllt ist:

```
if condition:
    statement
elif condition:
    statement
else:
    statement
```

3.2. Schleifen

Eine `for`-**Schleife** wiederholt eine Anweisung eine spezifizierte Anzahl von Malen:

```
for i in range(n):  # n Iterationen
    statement
for elem in lis:  # iteriert durch Liste
    statement
for key, value in dic.items():  # iteriert durch Dict
    statement
```

Bleibt `i` ungenutzt, kann es durch `_` ersetzt werden

Eine `while`-**Schleife** wiederholt eine Anweisung, bis eine bestimmte Bedingung nicht länger erfüllt ist:

```
while condition:
    statement
```

`break` beendet eine Schleife vorzeitig, `continue` springt zur nächsten Iteration der Schleife

3.3. Try-Except-Anweisungen

Mit `try-except`-**Anweisungen** können Fehler abgefangen werden:

```
try:
    statement
except condition:
    statement
```

Wenn ein Fehler in `try` auftritt und `condition` zutrifft, wird `except` ausgeführt

3.3. Funktionen

Funktionen sind Codeblöcke:

```
def function_name(arg1, arg2 = 2):
    statement
    return …
```

`return` beendet die Ausführung einer Funktion und ermöglicht optional die Rückgabe eines Werts

Im obigen Beispiel besitzt `arg2` einen Default-Wert von 2

Im Programm werden Funktionen wie folgt aufgerufen. Dies kann, anders als in C++, auch vor ihrer Deklarierung geschehen:

```
function_name(1)
function_name(2, 4)  # Default-Wert für arg2 wird
überschrieben
```

Funktionen und Variablen, die innerhalb einer Funktion initialisiert werden, sind **lokal** und können nur dort aufgerufen werden:

```
def global_function(…):
    local_var = 10
    def local_function(…):
        …
global_function(…)
local_function(…)  # ERROR, unbekannte Funktion
local_var = 5  # ERROR
```

`map(…)` nimmt eine Funktion und ein Iterable (z.B. eine Liste) als Argument und wendet die Funktion auf alle Elemente im Iterable an:

```
def function_1(…):
    statement
map(function_1, lis)
```

Sowohl der erwartete Rückgabetyp einer Funktion als auch die erwarteten Datentypen der Argumente können explizit angegeben werden. Diese werden von Python allerdings nicht forciert:

```
def function_1(arg1: str, …) -> int:
    …
```

Lambda-Ausdrücke sind Funktionsausdrücke ohne Namen:

```
lambda arg: statement
```

Folgend ein Beispiel zu Lambda-Ausdrücken:

```
result = lambda var1, var2: var1 + var2
print(result(2, 3))  # 5
map(lambda var: var * 2, [1, 2]) # [2,4]
```

4. Klassen

Eine **Klasse** ist ein Datenkonstrukt, das Variablen (**Attributes**) und Funktionen (**Methods**) speichern kann:

```
class class_name:
    var1 = …
    var2 = …
    def method_name(self, …):
        statement
```

Ein Objekt einer Klasse wird folgendermassen erstellt:

```
object_name = class_name(…)
```

Dabei wird der **Konstruktor** der Klasse aufgerufen:

```
class class_name:
   def __init__(self, value1, …):
      self.var1 = value1
      …
```

wobei mit `self` auf das jeweilige Objekt verwiesen wird

Ausserhalb der Klasse geschehen Aufrufe wie folgt:

```
object_name.var1
object_name.method_name(…)
```

ausserhalb der Klasse ist `self` kein Funktionsargument, da das Objekt durch `object_name` spezifiziert wird

Versteckte Attribute/Methoden sind von ausserhalb der Klasse nicht aufrufbar und mit einem `__` gekennzeichnet:

```
class class_name:
   def __init__(self , …):
      self.__var1 = value1
      …
```

Folgend zwei Beispiele zu versteckten Attributen:

```
print(object_name.__var1)  # Error: No such attribute
object_name.varx = 2  # erstellt neues, lokales
Attribute varx in object_name
```

Klassen können vererbt werden:

```
class parent_class:
   def function_name(…):
      …
   …
class child_class(parent_class):
   …
```

Wodurch `child_class` durch **Inheritance** alle Attribute und Methoden von `parent_class` erbt:

```
object_name = child_class(…)
object_name.function_name(…)  # valide
```

Um gewisse Operationen für Klassen zu definieren, wird auf die **Magic Methods** zugegriffen. Einige Beispiele:

`<`	less than	`__lt__`
`>=`	greater than or equal	`__ge__`
`==`	equal to	`__eq__`
`+`	addition	`__add__`
`**`	exponentiation	`__pow__`
`print(…)`	printing	`__str__`

Folgend ein Beispiel zu den Magic Methods:

```
class class_name:
   def __init__(self, value1, value2):
      self.name = value1
      self.age = value2
   def __str__(self):
      return self.name + " is " + str(self.age)
   def __lt__(self, other):
      return self.age < other.age
person = class_name("Ralf", 20)
print(person)  # "Ralf is 20"
print(person < person)  # False
```

5. Matplotlib

Matplotlib ist eine Bibliothek zur Erstellung von Grafiken

```
import matplotlib.pyplot as plt
X, Y = [0,1,2,3],[0,2,4,6]
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(X, Y)  # lineare Funktion
plt.show()
```

```
fig = plt.figure()
ax = fig.add_subplot()
ax.scatter(X, Y, marker = 'x', c = 'r')  # Scatterplot
plt.show()
```

`marker` gibt das Symbol und `c` die Farbe der Datenpunkte an

```
import numpy as np
values = np.random.randint(1,100,1000)
fig = plt.figure()
ax = fig.add_subplot()
ax.hist(values, bins = 100)  # Histogramm
plt.show()
```

`bins` legt fest wie viele Säulen dargestellt werden

6. Pandas

Pandas ist eine Bibliothek für Datenanalyse. Ein Pandas **Dataframe** ist eine Tabelle mit Zeilen und Spalten (= **Series**):

Index	Month	Name	City	College	Age
0	January	Greg	Boston	BC	25
1	February	Max	San Diego	UCSD	30
2	March	Carlos	New York	NYU	21

Ein Dataframe wird wie folgt erzeugt:

```
df = pd.DataFrame()
```

Der Zugriff auf Daten in einem Dataframe erfolgt mit:

```
data["Month"]  # einzelne Spalte
data[["Name", "City"]]  # mehrere Spalten
data["Name"][2]  # gibt 'Carlos' zurück
data.iloc[1]  # Zeile mit Index 1
data[1:3]  # Zeilen mit Index 1 und 2
data.loc[1:3, "Month":"City"]  # Werte der Zeilen mit
Indizes 1 und 2 der Spalten "Month" und "City"
data.iloc[1:3, 0:2]  # äquivalent zu oben
```

Spalten von Dataframes können umbenannt werden:

```
data.set_index("Row")  # Index-Spalte
data.rename(columns = {"City":"Location"})
data.columns = ["Monat", "Name", "Stadt", …]  # alle
Spalten auf einmal
```

Dataframes können gefiltert werden:

```
data[data["Age"] > 30]  # nur Person über 30 Jahren
data["Name"][data["Age"] > 30]  # nur Namen von
Personen über 30
```

Zeilen und Spalten können selektiv gelöscht werden:

```
data.drop(columns = ["Age"])  # löscht Spalte
data.drop(data.index[0])  # löscht Zeile
data.dropna(axis = 0, how = "any")  # löscht Zeile,
wenn ein Wert in der Zeile NaN ist
data.dropna(axis = 0, how = "all")  # löscht Zeile,
wenn alle Werte in der Zeile NaN sind
data.dropna(axis = 1, how = "any")  # löscht Spalte,
wenn ein Wert in der Spalte NaN ist
```

Weitere Hilfsfunktionen für Dataframes:

```
data["Age"].sum()  # summiert alle Spaltenwerte
data["Age"].max()  # maximaler Spaltenwerte
data.fillna(…)  # ersetzt alle NaN mit …
```

7. Algorithmen

7.1. Laufzeitanalysen – Allgemeines

Ein **Algorithmus** ist eine Abfolge von Anweisungen zur Lösung eines bestimmten Problems. Algorithmen lassen sich über ihre Laufzeiten vergleichen. Die **Laufzeit** wird in der Regel asymptotisch für grosse Eingaben bestimmt. Wir betrachten die Funktion f, die die Laufzeit eines Algorithmus in Abhängigkeit von der Eingabegrösse n angibt:

$f \in \mathcal{O}(g)$	f wächst ab $n > n_0$ höchstens so schnell wie eine Referenzfunktion g: $$0 \leq f(n) \leq c \cdot g(n)$$ g ist eine **obere Schranke** (schlechtester Fall)
$f \in \Omega(g)$	f wächst ab $n > n_0$ mindestens so schnell wie eine Referenzfunktion g: $$0 \leq c \cdot g(n) \leq f(n)$$ g ist eine **untere Schranke** (bester Fall)
$f \in \Theta(g)$	Die Funktionen f und g sind asymptotisch, wachsen ab $n > n_0$ also genauso schnell: $$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$ g ist eine **obere und untere Schranke**

Wobei anstelle von $\in$ oft $=$ verwendet wird

Folgend ein Beispiel von aufsteigend sortierten Laufzeiten:

$$\mathcal{O}(1) < \mathcal{O}(\log(n)) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(n) < \mathcal{O}(n\log(n)) < \mathcal{O}(n^2) < \mathcal{O}(2^n) < \mathcal{O}(n!) < \mathcal{O}(n^n)$$

Wobei $\log(\dots)$ in den meisten Fällen der Logarithmus zur Basis 2 ist

Achtung: $n = \mathcal{O}(n^2)$ und $n^2 = \mathcal{O}(n^2)$, dies impliziert aber nicht, dass die Laufzeiten von n und n^2 asymptotisch sind

Für die Laufzeitenanalyse sind folgende Formeln hilfreich:

Grundlagen Logarithmus:	$c^a = n \longrightarrow a = \log_c(n)\,, \qquad \log(n^a) = a\log(n)$
Vereinfachungen Sigma:	$\sum_{i=0}^{n-1} c = \sum_{i=1}^{n} c\,, \qquad \sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i$
Asymptotik Sigma:	$\sum_{i=0}^{n^a} i^b = \Theta\left(n^{a(b+1)}\right), \qquad \sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

Für die Berechnungen sind Modelle notwendig, die einzelnen Aktionen Kostenpunkte zuweisen. Beispiel: Jede Operation, jeder Vergleich und jede Zuweisung besitzen einen Kostenpunkt von 1:

```
def sort(A):
  n = len(A)
  for i in range(n):
    mini = i
    for i in range(i + 1, n):
      if A[j] < A[mini]:
        mini = j
    A[mini], A[i] = A[i], A[mini]
```

Die obige Laufzeit beträgt $1 + \sum_{i=0}^{n-1}\left(1 + \sum_{j=i+1}^{n-1} 2 + 1\right) = \Theta(n^2)$

Weitere Beispiele: Jeder Aufruf von `f()` hat einen Kostenpunkt von 1

$\Theta(1)$:
```
def g(n):
  for i in range(500):
    f()
```

$\Theta(\sqrt{n})$: denn $i^2 = n$
```
def g(n):
  i = 1
  while i*i < n:
    i += 1
    f()
```

$\Theta(\log(n))$: denn $2^k = i < n$
```
def g(n):
  i = 1
  while i < n:
    i *= 2
    f()
```

$\Theta(n^4)$:
```
def g(n):
  for i in range(n * n):
    for j in range(i):
      f()
```

$\Theta(\log(n))$: denn $n \geq 2^i$
```
def g(n):
  i = 0
  while n // 2**i >= 1:
    i += 1
    f()
```

7.2. Laufzeitanalysen rekursiver Funktionen

Zur Laufzeitanalyse von rekursiven Funktionen wird das **Teleskopie-Verfahren** verwendet:

```
def g(n):
  f()
  if n > 1:
    g(n - 3)
```

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-3) + 1, & n > 1 \end{cases}$$

$$\Rightarrow \Theta(n)$$

Woraus logisch abgeleitet werden kann:

$$T(n) = T(n-3) + 1 = T(n-6) + 1 = T(n-3i) + i$$

Die Rekursion endet, wenn $n - 3i = 1$:

$$T(n) = T(1) + \frac{n-1}{3} = 1 + \frac{n-1}{3} \in \Theta(n)$$

Beispiele: Jeder Aufruf von `f()` hat einen Kostenpunkt von 1

```
def g(n):
  if n >= 1:
    for i in range(n):
      f()
    g(n // 2)
    g(n // 2)
```

$$T(n) = \begin{cases} 0, & n = 0 \\ 2T(n/2) + n, & n \geq 1 \end{cases}$$

$$\Rightarrow \Theta(n\log(n))$$

```
def g(n):
  if n > 1:
    g(n // 2)
    g(n // 2)
  else:
    f()
```

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2), & n > 1 \end{cases}$$

$$\Rightarrow \Theta(n)$$

```
def g(n):
  for i in range(n):
    f()
  g(n // 2)
```

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

$$\Rightarrow \Theta(n)$$

```
def g(n):
  while n > 1:
    n -= 1
    g(n)
  f()
```

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + \cdots + 1, & n > 1 \end{cases}$$

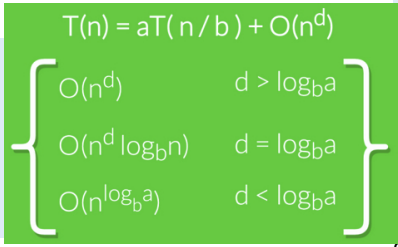
$$\Rightarrow \Theta(2^n)$$

7.4. Sortieralgorithmen

7.4.1. Selection Sort

Ein Array A wird sortiert, indem $A[i]$ mit dem Minimum von $A[i+1:]$ vertauscht wird:

```
def selection_sort(A):
  n = len(A)
  for i in range(0,n):
    min = i
    for j in range(i+1,n):
      if A[j] < A[min]:
        min = j
    A[min],A[i]=A[i],A[min]
```



Vergleiche: $\mathcal{O}(n^2)$. *Swaps*: $\mathcal{O}(1)$ bei sortierter Liste und $\mathcal{O}(n)$ im durchschnittlichen und schlechtesten Fall (invers sortiert)

7.4.2. Insertion Sort

Ein Array A wird sortiert, indem $A[i]$ an die richtige Position im sortierten Bereich $A[:i-1]$ eingefügt wird:

```
def insertion_sort(A):
  n = len(A)
  for i in range(1, n):
    key, j = A[i], i - 1
    while j >= 0 and A[j] > key:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = key
```

Vergleiche: $\mathcal{O}(n)$ bei sortierter Liste und $\mathcal{O}(n^2)$ im durchschnittlichen und schlechtesten Fall. *Swaps*: $\mathcal{O}(1)$ bei sortierter Liste und $\mathcal{O}(n^2)$ im durchschnittlichen und schlechtesten Fall (invers sortiert)

7.4.3. Merge Sort

Ein Array A wird sortiert, indem er in kleinere Subarrays geteilt wird (**Divide & Conquer** Prinzip), die dann einzeln sortiert und am Ende zusammengefügt werden:

```
def msort(A):
  n = len(A)
  if n <= 1:
    return A
  else:
    A1 = msort(A[:n // 2])
    A2 = msort(A[n // 2:])
  return merge(A1, A2)
```

```
# Alternative:
def msort(A, l, r):
  if (r - l) > 1:
    msort(A,l,(r+l)//2)
    msort(A,(r+l)//2,r)
    merge(A,l,(r+l)//2,r)
```

Vergleiche: $\mathcal{O}(n\log(n))$

7.4.3. Quick Sort

Ein Array A wird sortiert, indem er rekursiv bei einem **Pivot** q geteilt wird (Divide & Conquer). Die Hälfte $A[:q-1]$ enthält die Elemente, die kleiner als $A[q]$ sind und $A[q+1:]$ jene, die grösser sind:

```
def quick_sort(A, l, r):

  def partition(A, l, r):
    key = A[r]  # oder Pivot manuell festlegen
    i = l
    for j in range(l, r):
      if A[j] < key:
        A[i],A[j]=A[j],A[i]
        i += 1
    A[i],A[r] = A[r],A[i]
    return i

  if l < r:
    q = partition(A, l, r)
    quick_sort(A, l, q-1)
    quick_sort(A, q+1, r)

quick_sort(A,0,len(A)-1)
```

Vergleiche: $\mathcal{O}(n\log(n))$ im besten Fall (Median als Pivot) und durchschnittlichen Fall (willkürliches Pivot), $\mathcal{O}(n^2)$ im schlechtesten Fall (Minimum oder Maximum als Pivot). *Swaps*: $\mathcal{O}(n)$ im besten Fall und $\mathcal{O}(n\log(n))$ im durchschnittlichen und schlechtesten Fall

7.5 Listen & Suchalgorithmen

Der Zugriff auf ein Listen-Element per Index hat eine konstante Laufzeit von $\Theta(1)$. Die Laufzeit einiger Listen-Operationen hängt allerdings davon ab, ob eine Liste sortiert oder unsortiert ist:

Listenart	Suche	Einfügen	Löschen
Sortierte Liste	$\Theta(\log(n))$	$\mathcal{O}(n)$	$\Theta(\log(n))$
Unortierte Liste	$\Theta(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Laufzeit des Einfügens: Anzahl Vergleiche und Zuweisungen, um ein Element an richtige Position zu bringen. **Laufzeit des Suchens:** Anzahl Vergleiche, um ein gesuchtes Element zu finden

In einer **unsortierten Liste** wird nach Elementen linear gesucht:

```
def lin_search(A, s):
    for idx, val in enumerate(A):
        if val == s:
            return idx
```

In einer **sortierten Liste** können Elemente binär gesucht werden:

```
def bin_search(A, left, right, val):
    if right < left:
        return None
    else:
        mid = (left + right) // 2
        if A[mid] == val:
            return mid
        elif val < A[mid]:
            return bin_search(A, left, mid - 1, val)
        else:
            return bin_search(A, mid + 1, right, val)
```

Laufzeiten, wenn jeder *Vergleich* einen Kostenpunkt von 1 besitzt:

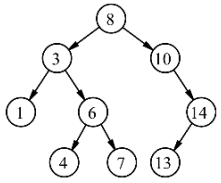
Algorithmus	Worst	Best	Average
Lineare Suche	$\mathcal{O}(n)$	$\Omega(1)$	$\Theta(n)$
Binäre Suche	$\mathcal{O}(\log(n))$	$\Omega(1)$	$\Theta(\log(n))$

8. Bäume

Min_Height:
Abrunden kleinere zweierPotenz: Z
2^(x-1) = Z

8.1. Binäre Suchbäume

Ein **binärer Suchbaum** ist ein Baum, der aus inneren Knoten besteht, von denen höchstens zwei Teilbäume abzweigen (**Ordnung** = 2). Der Schlüssel des linken Kinderknotens ist kleiner, der des rechten stets grösser als der Wert des Elternteils



Perfekt:	Alle Ebenen des Baumes sind voll
Vollständig:	Alle ausser der letzten Ebene sind voll
Komplett:	Jeder Knoten besitzt 0 oder 2 Kinderknoten
Balanciert:	Die Höhen des linken und rechten Teilbaums unterscheiden sich höchstens um 1
Degeneriert:	Jeder Knoten hat nur ein Kind

Knoten ohne Kinder werden als **Blätter** bezeichnet. Die **Höhe** h ist die maximale Pfadlänge von der Wurzel zum entferntesten Blatt

Folgend einige Beispiele für Suchbaum-Hilfsfunktionen:

```
def height(node):
    if node == None:
        return 0
    else:
        return 1 + max(height(node.right),height(node.left))

def find_node(root, key):
    n = root
    while n != None and n.key != key:
        if key < n.key:
            n = n.left
        else:
            n = n.right
    return n
```

$\Theta(n)$: Baum ist degeneriert, $\Theta(\log(n))$: Baum ist balanciert, $\Theta(1)$: key ist die Wurzel

```
def add_node(root, key):
    if root == None:
        root = Node(key)
    n = root
    while n.key != key:
        if key < n.key:
            if n.left == None:
                n.left = Node(key)
            n = n.left
        else:
            if n.right == None:
                n.right = Node(key)
            n = n.right
    return root
```

$\Theta(n)$: Baum ist degeneriert, $\Theta(\log(n))$: Baum ist balanciert, $\Theta(1)$: Baum ist leer

Ein Knoten ohne Kind kann gelöscht werden, bei einem Kind wird er durch dieses ersetzt. Sonst durch den **symmetrischen Nachfolger**:

```
def del_node(root, key):
    if root is None:
        return None
    if key < root.key:
        root.left = del_node(root.left, key)
    elif key > root.key:
        root.right = del_node(root.right, key)
    else:
        if root.left is None and root.right is None:
            root = None
        elif root.left is None:
            root = root.right
        elif root.right is None:
            root = root.left
        else:
            successor = find_successor(root.right)
            root.key = successor.key
            root.right = del_node(root.right, successor.key)
    return root
```

```
def find_successor(node):
    while node.left is not None:
        node = node.left
    return node
```

Die Laufzeit $\Theta(h)$ hängt von der Höhe des Baumes ab

Binäre Suchbäume können auf mehrere Arten traversiert werden:

In-order / symmetrische Reihenfolge (aus Reihenfolge kann *nicht* eindeutig auf Suchbaum zurückgeschlossen werden):

```
def inorder(node):
    if node is None:
        return []
    return inorder(node.left) + [node.key] \
                              + inorder(node.right)
```

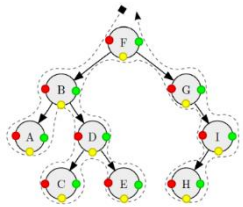
Pre-order / Hauptreihenfolge (aus Reihenfolge kann eindeutig auf Suchbaum zurückgeschlossen werden):

```
def preorder(node):
    if node is None:
        return []
    return [node.key] + preorder(node.left) \
                      + preorder(node.right)
```

Post-order / Nebenreihenfolge (aus Reihenfolge kann eindeutig auf Suchbaum zurückgeschlossen werden):

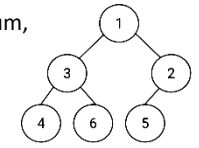
```
def postorder(node):
    if node is None:
        return []
    return postorder(node.left) + postorder(node.right) \
                                + [node.key]
```

Um die Traversierung eines Suchbaums zu vereinfachen, kann eine Kennzeichnung verwendet werden, bei der Rot, Gelb und Grün für die Reihenfolgen Pre-Order, In-Order und Post-Order stehen

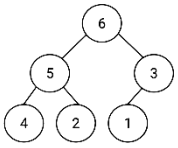


8.2. Heaps

Ein **Heap** ist ein binärer Baum, der bis auf die letzte Ebene voll ist und von links nach rechts aufgefüllt wurde.
Ein **Min-Heap** ist ein Heap, bei dem die Schlüssel der Kinder grösser als die des Elternknotens sind. Bei einem **Max-Heap** sind die Schlüssel der Kinder kleiner als die des Elternknotens



Min heap



Max Heap

Für die Indizes des i-ten Knoten gilt:

Indexierung:	Startend bei 0	Startend bei 1
Linkes Kind	$2i + 1$	$2i$
Rechtes Kind	$2i + 2$	$2i + 1$
Elternknoten	$(i - 1)\ //\ 2$	$i\ //\ 2$

Heaps werden als Listen abgespeichert. Die beiden vorherigen Beispiele entsprechen den Listen $[1,3,2,4,6,5]$ und $[6,5,3,4,2,1]$

Um einem Heap ein Element hinzuzufügen, platziert man es zunächst an der ersten freien Stelle der untersten Ebene, lässt es dann nach oben klettern, bis die Bedingungen für den Heap wieder erfüllt sind. In der Folge die Vorgehensweise bei einem Max-Heap:

```
def insert(heap, value):
    heap.append(value)
    sift_up(heap, len(heap) - 1)

def sift_up(heap, end):
    val = heap[end]
    k, m = end, k // 2
    # Bei Min-Heap das > zu < umkehren:
    while k > 0 and val > heap[m]:
        heap[k] = heap[m]
        k, m = m, k // 2
    heap[k] = val
```

Vorgehensweise zur Entfernung des Maximums eines Max-Heaps:

```
def remove_max(heap):
    if len(heap) == 0:
        return
    heap[0] = heap[-1]
    heap.pop()  # entfernt letztes Element einer Liste
    sift_down(heap, 0, len(heap))

def sift_down(heap, idx, end):
    while 2 * idx + 1 < end:
        j = 2 * idx + 1
        # Bei Min-Heap das zweite < zu > umkehren:
        if j + 1 < end and heap[j] < heap[j + 1]:
            j += 1
        # Bei Min-Heap das < zu > umkehren:
        if heap[idx] < heap[j]:
            heap[idx], heap[j] = heap[j], heap[idx]
            idx = j
        else:
            return
```

Eine Liste kann wie folgt zu einem Heap umgeformt werden:

```
def heapify(lis):
    n = len(lis)
    for i in range(n // 2 - 1, -1, -1):
        sift_down(lis, i, n)  # Initialisierung oben
```

$\mathcal{O}(\log(n))$ für das Einfügen und $\mathcal{O}(n)$ für das Entfernen eines Elements. $\mathcal{O}(n)$ um eine Liste in einen Heap umzuwandeln (Heapify)

Listen können sortiert werden, indem sie zuerst in einen Heap umgewandelt werden. Der Sortieralgorithmus nennt sich **Heap Sort**:

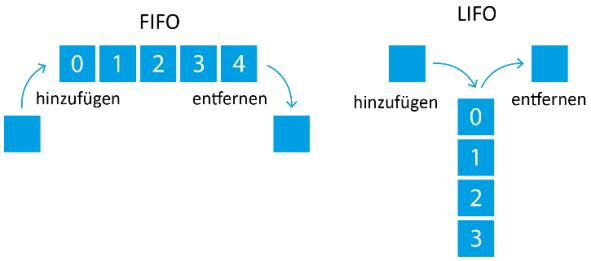
```
def heap_sort(lis):
    n = len(lis)
    heapify(lis)  # Initialisierung oben
    for i in range(n - 1, 0, -1):
        lis[0], lis[i] = lis[i], lis[0]
        sift_down(lis, 0, i)  # Initialisierung oben
```

Komplexität nicht Listenabhängig

$\mathcal{O}(n)$ wenn alle Elemente dieselben sind. Sonst $\mathcal{O}(n\log(n))$

8.3. Andere Datenstrukturen

Verkettete Listen bestehen aus Knoten, die jeweils auf den nächsten Knoten zeigen. Beim **FIFO**-Prinzip (**Queue**) werden neue Elemente am Ende der Kette hinzugefügt. Dadurch kann das zuerst hinzugefügte Element einfach abgerufen werden. Beim **LIFO**-Prinzip (**Stack**) werden neue Elemente am Anfang der Kette hinzugefügt und können somit schnell wieder abgerufen werden:



Bei verketteten Listen (FIFO/LIFO) gilt für die Laufzeiten:

Zugriff:	$\mathcal{O}(n)$	Einfügen:	$\Theta(1)$
Suche:	$\mathcal{O}(n)$	Löschen:	$\Theta(n)/\Theta(1)$

Hash-Tabellen sind Datenstrukturen, in denen eine `hash_function` einem Wert einen Index zuweist. Dadurch können Daten schnell aufgerufen werden. Der Index ist nicht eindeutig, weswegen es zu Kollisionen kommt. Diese kann man auf zwei Arten beheben:

- **Chaining**: am berechneten Index wird eine verkettete Liste von Elementen abgespeichert

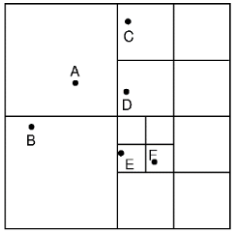
```
def create_hash(lis, size):
    hash_table = [ [] for x in range(size)]
    for elem in lis:
        index = hash_function(elem) % size
        if elem not in hash_table[index]:
            hash_table[index].append(elem)
```

- **Probing**: der nächstfreie Index in der Tabelle wird gewählt

```
def create_hash(lis, size):
    hash_table = [ None for elem in range(size)]
    for elem in lis:
        index = hash_function(elem) % size
        i = index
        while True:
            if hash_table[i] == elem:
                break
            if hash_table[i] == None:
                hash_table[i] = elem
                break
            i += 1
            i = i % M
            if i == index:
                break
```

Bei keinen Kollisionen betragen die Laufzeiten für das Suchen, Einfügen und Löschen von Elementen $\mathcal{O}(1)$. Im schlechtesten Fall gibt es n Kollisionen und somit eine Laufzeit von $\mathcal{O}(n)$

Quadtrees sind baumartige Datenstrukturen mit Ordnung 4. Jeder Knoten stellt ein Quadrat dar, das eine max. Anzahl an Punkten speichern kann. Wird die Kapazität überschritten, unterteilt man ihn in weitere 4 Quadrate, die dann zu Kindern werden. Jeder Punkt kann in $log(n)$ gefunden, gelöscht und eingefügt werden



Beispiel: Hinzufügen eines Punktes in einen Quadtree

```
class QuadTree:
    def __init__(self, l, u, max_cap)
        self.l, self.u, self.m = l, u, max_cap
        self.points = []
        self.children = None
        self.count = 0
    def insert(self, point):
        if self.children is not None:
            index = self.get_index(point)
        if index is not None:
            self.children[index].insert(point)
            self.count += 1
            return
        self.points.append(point)
        if len(self.points) > self.m:
            self.subdivide()
            for point in self.points:
                index = self.get_index(point)
                if index is not None:
                    self.children[index].insert(point)
                    self.count += 1
            self.count = []
```

wobei `subdivide()` und `get_index()` weitere Methoden der Quadtree-Klasse sind

9. Dynamische Programmierung

9.1. Memoization

Bei der **dynamischen Programmierung** wird ein Problem rekursiv in kleinere Subprobleme unterteilt. Die Ausgabe wird in einem **Memory** zwischengespeichert, um wiederholte Berechnungen zu vermeiden. Bei erneutem Aufruf der Funktion mit derselben Eingabe wird das zuvor berechnete Ergebnis aus dem Speicher abgerufen

Folgend die Berechnung der n-ten Fibonaccizahl mit Memoization:

```
def fib(n, memo):                    rec -> memo:
    if memo[n] != None:              if memo == None -> initialisierende
        return memo[n]               return zu memo[n] wechseln
    if n <= 1:                       am ende return memo [n]
        return n
    res = fib(n - 1, memo) + fib(n - 2, memo)
    memo[n] = res
    return res
fib(n, [None] * (n + 1))
```

Folgend ein Beispiel, bei dem eine Liste `coins` mit Münzenwerten eingegeben wird und geprüft werden soll, ob ein Zielwert `n` als Summe der anderen Münzwerte erreicht werden kann:

```python
def possible(n, coins):
  s = [False] * (n + 1)
  s[0] = True
  for val in range(1, n + 1):
    for coin in coins:
      if val - coin >= 0 and s[val - coin]:
        s[val] = True
  return s[n]
```

```
s = [False] * (n + 1)
s[0] = True
for i in range(1, n + 1):
  for b in blocks:
    if i - b >= 0 and s[i - b]:
      s[i] = True
      break
return s
```

Folgend ein Beispiel, bei dem eine Zahl `n` gegeben ist, die schnellstmöglich auf 1 reduziert werden soll. Dafür darf in jedem Schritt 1 subtrahiert oder durch 2 respektive 3 dividiert werden:

```python
from math import inf
def min_steps(n):
  table = [math.inf] * (n + 1)
  table[1] = 0
  for i in range(n + 1):
    if i + 1 <= n:
      table[i + 1] = min(table[i] + 1, table[i + 1])
    if i * 2 <= n:
      table[i * 2] = min(table[i] + 1, table[i * 2])
    if i * 3 <= n:
      table[i * 3] = min(table[i] + 1, table[i * 3])
  return s[n]
```

Sei `S` eine Matrix $m \times n$ Matrix mit positiven Zahlenwerten. Es soll der maximale Wert bestimmt werden, der erreicht werden kann, wenn `S` von $(0,0)$ bis $(m-1, n-1)$ traversiert wird wobei man sich nur nach Westen, Süden oder Südwesten bewegen darf:

```python
def bestway(a):
  m, n = len(a), len(a[0])
  s = [[None] * n for _ in range(m)]
  L = [[None] * n for _ in range(m)]
  for j in range(n - 1, -1, -1):
    for i in range(m):
      if j == n - 1:
        s[i][j] = a[i][j]
      else:
        n_east = s[i - 1][j + 1] if i > 0 else -1
        east = s[i][j + 1]
        s_east = s[i + 1][j + 1] if i < m - 1 else -1
        if n_east >= east and n_east >= s_east:
          s[i][j] = a[i][j] + n_east
          L[i][j] = "North east"
        elif east >= n_east and east >= s_east:
          s[i][j] = a[i][j] + east
          L[i][j] = "East"
        else:
          s[i][j] = a[i][j] + s_east
          L[i][j] = "South east"
  i, j, path = 0, 0, []
  while j < n - 1:
    path.append(L[i][j])
    if L[i][j] == "North east":
      i -= 1
    elif L[i][j] == "South east":
      i += 1
    j += 1
  return s[0][0], path
```

Gegeben ist eine Liste A von Sprunglängen A_i. Das Ziel ist es, die minimale Anzahl von Sprüngen zu berechnen, um die gesamte Liste zu durchlaufen:

```python
def jumps(A, pos, M):
  if pos >= len(A):
    return 0
  elif pos not in M:
    vals = [jumps(A,pos+i+1,M) for i in range(A[pos])]
    M[pos] = min(vals) + 1
  return M[pos]
result = jumps(A, 0, {})
```

Gegeben sei ein Dreieck, für den die maximale Summe von Werten entlang eines Pfades von der Spitze zur Basis gefunden werden soll:

```python
def best_path(triangle):
    n = len(triangle)
    if n == 0:
      return None
    s = [[None] * n for _ in range(n)]
    s[n-1] = triangle[n-1]
    for i in range(n - 2, -1, -1):
        for j in range(i + 1):
            left, right = s[i + 1][j], s[i + 1][j + 1]
            s[i][j] = triangle[i][j] + max(left, right)
    return s[0][0]
```

10. Maschinelles Lernen

10.1. Modelle

Machine Learning-Modelle erkennen Korrelationen zwischen Daten, um Muster zu identifizieren und Vorhersagen zu treffen

Lineare Regression: linearer Zusammenhang zwischen Variablen, um den Funktionswert eines neuen Datenpunkts vorherzusagen:

```python
import pandas as pd
from sklearn.linear_model import LinearRegression
# Daten aus .csv Datei einlesen:
df = pd.read_csv("data.csv")
X = df.drop(['target'], axis=1)
y = df['target']
# Modell trainieren:
linreg = LinearRegression()
linreg.fit(X, y)
# Modell für Vorhersagen zu neuen Datenpunkten nutzen:
X_test = pd.read_csv("X_final.csv")
y_pred = linreg.predict(X_test)
```

Manchmal muss man die Daten in eine NumPy Matrix umwandeln:

```python
X = df.drop(['target'], axis=1).to_numpy()
y = df['target'].to_numpy()
```

Logistische Regression: Prognose zur Wahrscheinlichkeit für ein Label basierend auf einer einzigen Input-Variable:

```python
from sklearn.linear_model import LogisticRegression
…
logreg = LogisticRegression()
logreg.fit(X, y)
…
```

Entscheidungsbaum: Entscheidungen in Form eines Baumdiagramms:

```python
from sklearn.tree import DecisionTreeClassifier
…
tree = DecisionTreeClassifier()
tree.fit(X, y)
…
```

Neuronale Netze: Schichten von Knotenpunkten (**Neuronen**), die Eingabeinformationen verarbeiten:

```python
from sklearn.neural_network import MLPClassifier
…                                    32,32,16,16,8
mlp = MLPClassifier(hidden_layer_sizes=(100,50),
max_iter=200, random_state=42)  # beliebige Werte
mlp.fit(X, y)    theroetisch für params: {'hidden_layer_sizes':
…                [(128,),(64,32),(64,128,32)]}
```

wobei der **Hyperparameter** `hidden_layer_sizes` die Anzahl der neuronalen Schichten und ihrer Knoten spezifiziert, `max_iter` die Anzahl an Iterationen des Algorithmus angibt, und `random_state` sicherstellt, dass das Resultat reproduzierbar ist

Um die beste Kombination an Hyperparametern für ein Modell zu finden, kann man `GridSearchCV` verwenden:

```python
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
…
pam = {  # folgend zwei Beispiele für Bäume
   "max_depth" : range(1, 4),
   "criterion" : ["squared_error", "friedman_mse"]}
mdl = DecisionTreeRegressor()  # Modell wählen
grid = GridSearchCV(estimator=mdl, param_grid=pam, cv=3)
grid.fit(X, y)  # Cross-Validierung mit Grid-Search
```

Ähnlich kann man für eine Punktemenge ein Polynom approximieren:

```python
…
poly = Pipeline([("transf", PolynomialFeatures()),
("lr", LinearRegression())])
pam = {"transf__degree": range(5)}  # Potenz
grid = GridSearchCV(estimator=poly, param_grid=pam, cv=3)
grid.fit(X, y)
…
```

`cv` gibt die Anzahl der Cross-Validierungen an. Mit einem höheren Wert steigt die Genauigkeit und sinkt die Modell-Geschwindigkeit

10.2. Validierung

Um ein Modell zu testen, kann der Datensatz in einen **Trainings**- und **Validierungsdatensatz** unterteilt werden:

```python
from sklearn.model_selection import train_test_split
…
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2, random_state=42)
…
```

Die Genauigkeit maschineller Modelle ergibt sich aus der Abweichungen ihrer Vorhersagen zu den tatsächlichen Werten:

- **Mean Squared Error**: Mittlere quadratische Abweichung der Vorhersagen und eigentlichen Werte (Best: 0, Worst: ∞)

- **R2-Score**: Genauigkeit im Vergleich zum Durchschnitt (Best: 1, Worst: $-\infty$)
- **Accuracy score**: Prozent der korrekt klassifizierten Datensätze (Best: 1, Worst: 0)

```
from sklearn.metrics import mean_squared_error,
r2_score, accuracy_score
…
mse = mean_squared_error(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
…
```

Wenn ein maschinelles Programm fehlerhafte Zusammenhänge herstellt, könnte dies an der Architektur des Modells liegen:

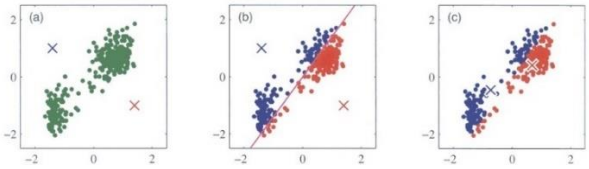
- **Overfitting:** Zu viele Knoten im Entscheidungsbaum, wodurch das Modell das Trainingsset auswendig gelernt hat
- **Underfiting:** Zu wenig Knoten, wodurch das Modell nicht fähig ist Zusammenhänge zu erkennen

Gewisse Datensätze sind für maschinelle Programme ungeeignet – z.B., wenn Labels keine Zahlenwerte sind. Mit Hilfe von **Encoding** Techniken können Strings in Zahlenwerte umgewandelt werden:

- **Ordinal encoding:** Jedes Label wird durch einen willkürlichen numerischen Wert ersetzt. Dies ist effizient, führt aber zu fälschlichen Zusammenhängen zwischen numerischen Werten
- **Mean encoding:** Der numerische Wert entspricht dem Durchschnitt für jedes Label in Bezug auf die Zielvariable
- **One-hot encoding:** Jedem Datenpunkt wird ein binärer Vektor mit 0 und 1 für jedes Feature zugewiesen. Dadurch wird die Unabhängigkeit der Labels erhalten, ist aber zeitaufwendig

10.3. Clustering

Clustering ordnet einem Datensatz unbeaufsichtigt Labels zu, um die Daten für maschinelle Programme nutzen zu können:



Anfangs werden k-Punkte – sogenannte **Zentroiden** – an zufälligen Punkten im Datenraum platziert. Jeder Datenpunkt wird anschliessend der nächsten Zentroide zugewiesen. Es bilden sich **Cluster**. Als nächstes wird jede Zentroide in den Mittelpunkt ihres Clusters verschoben und das Verfahren wiederholt, bis eine Zuweisungsrunde ohne Änderungen stattgefunden hat

k sollte der Anzahl der erwartenden Cluster entsprechen. Sind mehr Zentroiden als Cluster vorhanden, sind die Cluster umkämpft. Bei zu vielen Zentroiden kommt es zudem zu Overfitting

Clustering funktioniert gut für **sphärische Cluster**, bei denen Punkte eng um einen zentralen Punkt verteilt sind und die Varianz gering ist

Anhang - Aufgaben

A1. DP & Memoization – Seile

Gegeben ist eine Liste A von Stücklängen A_i eines Seils. Das Ziel ist es, das Seil in möglichst viele Teilstücke zu zerlegen, um eine bestimmte Gesamtlänge `n` zu erreichen

```
def solveRecursive(n, p):
  if n < 0:
    return -1
  elif n == 0:
    return 0
  else:
    max_val = -1
    for i in range(len(p)):
      max_val = max(max_val, solveRecursive(n-p[i], p))
    if max_val == -1:
      return -1
    return 1 + max_val

def solveDP(n, p):
  solution = [-1] * (n + 1)
  solution[0] = 0
  for i in range(1, n + 1):
    for j in range(0, len(p)):
      if i - p[j] >= 0:
        solution[i] = max(solution[i],solution[i-p[j]])
    if solution[i] != -1:
      solution[i] = 1 + solution[i]
  return solution[n]
```

A2. Memoization – Längste Teilfolge

Um die längste gemeinsame Teilfolge zweier Strings auf eine effiziente Weise zu finden, kann der folgende Memoization-Ansatz gewählt werden (als Memory kann auch eine $n \times m$ Matrix dienen):

```
def lcs(seq1, seq2, mem = None):
  n, m = len(seq1), len(seq2)
  if n == 0 or m == 0:
    return 0
  if mem is None:
    mem = dict()
  key = (n, m)
  if key not in mem:
    if seq1[-1] == seq2[-1]:
      mem[key] = 1 + lcs(seq1[:-1], seq2[:-1], mem)
    else:
      mem[key] = max(lcs(seq1[:-1], seq2, mem),
        lcs(seq1, seq2[:-1], mem))
  return mem[key]
```

A3. Algorithmus – Sliding Window

Um in einer Liste die längste konsekutive Teilliste zu finden, deren Maximum und Minimum sich genau um 1 unterscheiden, kann man eine simple `while`-Schleife verwenden:

```
def subsequence(nums: list) -> int:
  l, r, length = 0, 0, 0
  while r < len(nums):
    while nums[r] - nums[l] > 1:
      l += 1
    if nums[r] - nums[l] == 1:
      length = max(length, r - l + 1)
    r += 1
  return length
```

A4. K-kleinstes Element

```
def kth_smallest(arr, k):
  if len(arr) == 1:
    return arr[0]
  pivot = arr[len(arr) // 2]
  left = [x for x in arr if x < pivot]
  right = [x for x in arr if x > pivot]
  if k <= len(left):
    return kth_smallest(left, k)
  elif k > len(arr) - len(right):
    return kth_smallest(right,k-(len(arr)-len(right)))
  else:
    return pivot
```

A5. Memoization – Golomb-Zahlen

Die n-te **Golomb-Zahl** berechnet sich rekursiv:

```
def golomb(n, m = None):
  if m == None:
    m = dict()
  if n not in m:
    if n == 1:
      m[n] = 1
    else:
      m[n] = 1 + golomb(n-golomb(golomb(n-1, m), m), m)
  return m[n]
```

A6. DP – Aufgabenplanung

Berechne die maximale Belohnung, wenn man Werte aus zwei Listen `w1` und `w2` mit Zeitkosten aus `t1` und `t2` auswählt:

```
def tasks(w1, t1, w2, t2):
  n = len(w1)
  s = [None] * (n+1)
  s[-1] = 0
  for i in range(n-1, -1, -1):
    s[i] = max(w1[i]+s[i+t1[i]],w2[i]+s[i+t2[i]],s[i+1])
  return s[0]
```

A7. DP – Sprünge

Gegeben ist eine Liste A von Sprunglängen A_i. Es soll die minimale Anzahl an Sprüngen berechnet werden, um die Liste zu durchlaufen:

```
def jumps(a):
    s = [None] * (len(a)+1)
    s[n] = 0
    for i in range(len(a)-1, -1, -1):
      mini = s[i + 1]
      for j in range(1, a[i] + 1):
        if i + j <= n:
          mini = min(mini, s[i + j])
      s[i] = 1 + mini
    return s[0]
```