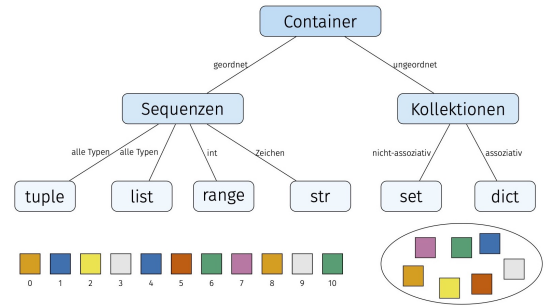
Container



Tuples (Tupel)

Tuples sind wie Listen, aber unveraenderlich (immutable).

```
# Initialisierung
my_tuple = (1, 2, 3)
single_element = (5,)  # Achtung: Einzeltupel braucht Komma!
```

Wichtige Methoden:

```
my_tuple.count(2)  # Anzahl der Vorkommen von 2
my_tuple.index(3)  # Index des ersten Vorkommens von 3
```

Tuples unterstuetzen Entpackung:

```
a, b, c = (10, 20, 30)  # a=10, b=20, c=30
```

Listen

Listen sind Sammlungen von Elementen mit variablen Datentypen und werden mit eckigen Klammern [] definiert.

```
# Initialisierung einer Liste
lis = [2, 1, 2, 3, 7]

# Leere Liste erstellen
empty_list = []
another_list = list()

# Liste mit mehreren gleichen Elementen
lis = [None] * 5  # [None, None, None, None, None]
```

Listenelemente koennen ueber Indizes aufgerufen werden:

```
first = lis[0]  # Ruft das erste Element auf
last = lis[-1]  # Ruft das letzte Element auf
```

Listen koennen auf ihren Inhalt ueberprueft werden:

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

Listen koennen mit Slicing [start:stop:step] geteilt werden:

```
slice1 = lis[:2]  # [2, 1]
slice2 = lis[1:-1]  # [1, 2, 3]
slice3 = lis[::-1]  # [7, 3, 2, 1, 2] (umgekehrt)
slice4 = lis[::2]  # [2, 2, 7]
```

Listen enthalten eingebaute Methoden:

```
len(lis)  # Gibt die Anzahl der Elemente zurueck
sum(lis)  # Gibt die Summe aller Elemente zurueck
max(list) # Gibt das Maximum der Liste zurueck
lis.append(4)  # Fuegt ein Element am Ende hinzu
lis.insert(1, 10)  # Fuegt 10 an Index 1 ein
lis.pop()  # Entfernt das letzte Element und gibt es zurueck
lis.remove(2)  # Entfernt das erste Vorkommen von 2
lis.index(3)  # Gibt den Index des ersten Vorkommens von 3 zurueck
lis.count(2)  # Gibt die Anzahl der Vorkommen von 2 zurueck
lis.sort()  # Sortiert die Liste aufsteigend
lis.reverse()  # Kehrt die Reihenfolge der Liste um (kein Rueckgabewert)
```

Listen koennen durch Iteration verarbeitet werden:

```
for element in lis:
    print(element)  # Gibt jedes Element aus

for idx, val in enumerate(lis):
    print(idx, val)  # Gibt Index und Element aus
```

Zwei Listen koennen kombiniert werden:

```
combined = list(zip([1, 2], [3, 4]))  # [(1,3), (2,4)]
```

Ergänzung zu sort() und sorted()

In Python gibt es zwei Methoden zur Sortierung von Listen:

- sorted() gibt eine neue sortierte Liste zurück, ohne die ursprüngliche Liste zu verändern.
- .sort() hingegen sortiert die Liste direkt in-place.

Grundlegende Sortierung:

```
a = [4, 2, 7, 5]

sorted_a = sorted(a)
# sorted_a = [2, 4, 5, 7], a bleibt unveraendert: [4, 2, 7, 5]

a.sort()
# a = [2, 4, 5, 7]
```

Absteigende Sortierung: Um eine Liste in absteigender Reihenfolge zu sortieren, kann das Argument reverse=True verwendet werden:

```
reverse_sorted_a = sorted(a, reverse=True)
# reverse_sorted_a = [7, 5, 4, 2], a bleibt unveraendert

a.sort(reverse=True)
# a = [7, 5, 4, 2]
```

Sortierung von Tupellisten: Falls eine Liste von Tupeln sortiert wird, erfolgt die Sortierung standardmäßig nach dem ersten Element jedes Tupels. Falls mehrere Tupel mit demselben ersten Wert existieren, erfolgt die Sortierung nach dem zweiten Element:

```
t = [(2, 3), (1, 2), (2, 2), (1, 3)]
t_sorted = sorted(t)
# t_sorted = [(1, 2), (1, 3), (2, 2), (2, 3)]
```

Sortierung nach einem spezifischen Kriterium: Falls man die Sortierung nach einem anderen Element der Tupel steuern möchte, kann man den Parameter key verwenden. Das folgende Beispiel sortiert die Tupel nach dem zweiten Element:

```
t = [(3, 'b'), (1, 'c'), (2, 'a')]
t_sorted = sorted(t, key=lambda x: x[1])
# t_sorted = [(2, 'a'), (3, 'b'), (1, 'c')]
```

List Comprehension

```
# Liste Filtern
[x for x in arr if cond]

# Funktion auf Liste anwenden
[f(x) for x in arr]

#Funktion und Filter
[f(x) for x in arr if cond]

# Ternaeroperator
[f(x) if cond else g(x) for x in arr]

# mehrere Listen zusammen verarbeiten
[f(x, y, z) for x, y, z in zip(arr1, arr2, arr3)]

# verschachtelte Listen erstellen
[[x*y for ]]
```

Strings

Strings sind Zeichenketten und unveraenderlich (immutable).

```
var = "string"  # Initialisierung
var[0]  # ruft erstes Zeichen auf
var[-1]  # ruft letztes Zeichen auf
```

Strings koennen miteinander verknuepft werden (Konkatenation):

```
var = "..." + "..." + "..."
```

Strings koennen auf ihren Inhalt ueberprueft werden:

```
"c" in var  # True, wenn der Buchstabe in der Zeichenkette vorkommt
```

Strings koennen mit **Slicing** [start:stop:step] geteilt werden:

```
var = "abcdef"
slice = var[:3]  # 'abc'
slice = var[1:-1]  # 'bcde'
slice = var[::-1]  # 'fedcba' (umgekehrt)
slice = var[::2]  # 'ace'
```

Strings enthalten eingebaute Methoden zur Verarbeitung:

```
len(var)  # gibt Laenge des Strings zurueck
var.strip()  # entfernt fuehrende und nachfolgende Leerzeichen
var.split(',')  # trennt Zeichenkette an ',' und erstellt eine Liste
''.join(["a", "b", "c"])  # verbindet eine Liste zu 'abc'
var.upper()  # wandelt in Grossbuchstaben um
var.lower()  # wandelt in Kleinbuchstaben um
```

Sets (Mengen)

Sets sind ungeordnete Sammlungen einzigartiger Elemente.

```
# Initialisierung
my_set = {1, 2, 3, 4}
empty_set = set()  # {} waere ein Dictionary!
```

Wichtige Methoden:

```
my_set.add(5)  # Fuegt ein Element hinzu
my_set.remove(3)  # Entfernt ein Element
my_set.clear()  # Leert das Set
x in my_set # Element suchen
```

Mithilfe von Sets können wir eine Liste effizient auf Duplikate untersuchen:

```
def check_duplicates(arr):
    s = set()
    for x in arr:
        if x in s:
            return True
        s.add(x)
    return False
```

Dictionaries

Dictionaries sind Sammlungen von Schluessel-Wert-Paaren und werden mit geschweiften Klammern {} definiert.

```
# Initialisierung eines Dictionaries
dict1 = {:}

# Leeres Dictionary erstellen
dict2 = {}
dict3 = dict()
```

Dictionaries koennen Werte ueber Schluessel abrufen:

```
name = dict1["name"]  # "Alice"
alter = dict1.get("alter")  # 25
```

Dictionaries koennen Werte aendern oder hinzufuegen:

```
dict1["stadt"] = "Hamburg"  # Aendert bestehenden Wert
dict1["beruf"] = "Ingenieur"  # Fuegt neues Schluessel-Wert-Paar hinzu
```

Dictionaries koennen auf ihre Schluessel oder Werte ueberprueft werden:

```
"name" in dict1  # True, wenn der Schluessel existiert
25 in dict1.values()  # True, wenn der Wert existiert
```

Dictionaries enthalten eingebaute Methoden:

```
len(dict1)  # Gibt die Anzahl der Paare zurueck
dict1.keys()  # Gibt alle Schluessel zurueck
dict1.values()  # Gibt alle Werte zurueck
dict1.items()  # Gibt alle Schluessel-Wert-Paare zurueck
dict1.pop("alter")  # Entfernt einen Schluessel und gibt seinen Wert zurueck
dict1.clear()  # Entfernt alle Elemente aus dem Dictionary
```

Dictionaries koennen durch Iteration verarbeitet werden:

```
for key in dict1.keys():
    pass

for value in dict1.values():
    pass

for key, value in dict1.items():
    pass
```

Dict Comprehension

```
# Dictionary filtern
{key:value for key, value in d.items if value < 10}

# zwei Listen als dict zusammenfuehren
{x: y for x, y in zip(arr1, arr2)}

# Verschachteltes Dict verwenden, nicht-verschachteltes erstellen
{k: v for subdict in d.values() for k, v in subdict.items()}
```

Klassen

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

```
class class_name:
    attribute1 = None
    attribute2 = None
    def method1(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, value1, ...):
        self.__var1 = value1
        ...
```

Klassen können vererbt werden:

```
class parent_class:
    def function_name(...):
        ...

class child_class(parent_class):
    ...
```

Wodurch child_class alle Attribute und Methoden von parent_class erbt:

```
object_name = child_class(...)
object_name.function_name(...) # moeglich, obwohl nur in der Elterklasse definiert
```

Beispiel für Vererbung:

```
class Measurement: # Elterklasse
    def __init__(self, date, time, location):
        self.date = date
        self.time = time
        self.location = location
    def __str__(self):
        return self.date + 'at' + self.time + 'in' + self.location

class Temperature(Measurement): #Kindesklasse
    def __init__(self, date, time, location, intensity):
        Measurement.__init__(self, date, time, location)
        self.intensity = intensity
    def __str__(self):
        return Measurement.__str__(self) + ':' + str(self.intensity)
```

Um gewisse Operationen für Klassen zu definieren, wird auf die Magic Methods zugegriffen (links: Vergleiche, rechts: relationale Operatoren):

Operator	Bedeutung	Magische Methoden
<	kleiner als	__lt__
<=	kleiner gleich	__le__
>	grösser als	__gt__
>=	grösser gleich	__ge__
==	gleich	__eq__
!=	ungleich	__ne__

Operator	Bedeutung	Magische Methode
+,+=	Addition	__add__,__iadd__
-	Subtraktion	__sub__
*	Multiplikation	__mul__
/	Division	__truediv__
//	Ganzzahldivision	__floordiv__
%	Modulo (Rest)	__mod__
**	Exponentiation	__pow__

Beispiel zu den Magic Methods:

```
class class_name:
    ...
    def __lt__(self, other): # kleiner als
        return ...

    def __add__(self, other): # Addition
        return ...
```

Pandas

Pandas ist eine Bibliothek für Datenanalyse. Ein Pandas Dataframe ist eine Tabelle mit Zeilen und Spalten.
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' zurueck
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
data.iloc[1:3, 0:2]  # aequivalent zu oben
```

Spalten in DataFrames umbenennen

Spalten von DataFrames können umbenannt werden:

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

Filtern von DataFrames

DataFrames können gefiltert werden:

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

Löschen von Zeilen und Spalten

Zeilen und Spalten können selektiv gelöscht werden:

```
data.drop(columns=["Age"])  # loescht Spalte
data.drop(data.index[0])    # loescht Zeile
data.dropna(axis=0, how="any")  # loescht Zeilen mit mind. einem NaN
data.dropna(axis=0, how="all")  # loescht Zeilen mit nur NaN
data.dropna(axis=1, how="any")  # loescht Spalten mit mind. einem NaN
```

Hilfsfunktionen für DataFrames

Weitere nützliche Funktionen für DataFrames:

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

Laufzeitanalyse

Grössenordnung von Funktionen

$$\log(n) < \sqrt{n} < n < n \cdot \log(n) < n^2 < 2^n < n! < n^n$$

Rechenregeln

Summen:

$$\sum_{i=0}^{n-1} 1 = n \in \Theta(n)$$

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta(n^3)$$

$$\sum_{i=0}^{n^2} i = \frac{n^2(n^2-1)}{2} \in \Theta(n^4)$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \in \Theta(2^n)$$

Logarithmusfunktionen:

$$\log n^n = n \log n \in \Theta(n \log n)$$
$$\log n! = n \log n - n \in \Theta(n \log n))$$

Funktionsklassen

Beispiel 1:

$$n \sum_{i=1}^{\log n} i = n \frac{\log n(\log n + 1)}{2} \in \Theta(n \log^2 n)$$

Wir verwenden die zweite Summe aus den Rechenregeln.
Beispiel 2:

$$\sum_{i=0}^{n} \log\left(n \sum_{j=0}^{n} j\right) = \sum_{i=0}^{n} \log\left(\frac{n(n+1)}{2}\right)$$
$$= (n+1) \log\left(\frac{n(n+1)}{2}\right) \in \Theta(n \log n)$$

Wobei wir auch hier die zweite Summe verwendet haben.
Beispiel 3:

$$\frac{(2^n)!}{(2^n-1)!} = \frac{2^n(2^n-1)!}{(2^n-1)!} = 2^n \in \Theta(2^n)$$

Code-Snippets

Nicht-rekursive Snippets

Beispiel 1:

```
def run(n):
    for m in range(0, n):
        for k in range(0, m):
            op()
```

Laufzeit $\in \Theta(n^2)$
Beispiel 2:

```
def run(n):
    count = 0
    while n // (2 ** count) >= 1:
        op()
        count += 1
```

Laufzeit $\in \Theta(\log n)$
Beispiel 3:

```
def run(n):
    l = 0
    r = n
    while l < r:
        op()
        m = (l + r) // 2
        if h(m): # function that returns a boolean
            l = m + 1
        else:
            r = m - 1
```

Laufzeit $\in \Theta(\log n)$
Beispiel 4:

```
def run(n):
    for m in range(0, n):
        for k in range(0, m * m):
            op()
```

Laufzeit $\in \Theta(n^3)$
Beispiel 5:

```
def g(n):
    i = n
    while i > 1:
        for j in range(0, n):
            f()
        i = i // 2
```

Laufzeit $\in \Theta(n \log n)$.

Rekursive Snippets

Beispiel 1:

```
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        count += 1
```

Laufzeit $\in \Theta(\log n)$.
Beispiel 2:

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

Laufzeit $\in \Theta(\log n)$. Beispiel 3:

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

Laufzeit $\in \Theta(n)$
Beispiel 4:

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

Laufzeit $\in \Theta(n \log n)$
Beispiel 5:

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

Wir haben folgende Rekurrenzrelation:

$$T(n) = \begin{cases} 1, & \text{falls } n = 1 \\ 4T(n/2) + 1, & \text{falls } n > 1 \end{cases}$$

Wir teleskopieren:

$$T(n) = 1 + 4T(n/2) = 1 + 4(1 + 4T(n/4))$$
$$= 1 + 4 + 4^2 T(n/2^2) = 1 + 4 + 4^2 + \ldots + 4^k T(n/2^k)$$

Die Rekursion stoppt, wenn $n = 1$, also nach k-Schritten, wobei gilt $k = \log_2 n$. Setzen wir dies ein erhalten wir folgende Summe:

$$T(n) = \sum_{i=0}^{\log n} 4^k = \frac{1 - 4^{\log n + 1}}{1 - 4}$$

Da der dominante Term $4^{\log n}$ ist können wir approximieren:

$$T(n) \approx 4^{\log n} = (2^{\log n})^2 = n^2$$

Daher ist die Laufzeit $\in \Theta(n^2)$

Sortieralgorithmen

Bubble Sort

Beschreibung: Der Algorithmus vergleicht benachbarte Elemente im Array und vertauscht sie, falls sie in der falschen Reihenfolge stehen. Dieser Vorgang wird wiederholt, bis das gesamte Array sortiert ist.
Invariante: Nach dem k-ten Durchlauf befinden sich die k größten Elemente an ihren endgültigen Positionen am Ende des Arrays.
Speicherplatz: Es wird kein zusätzlicher Speicherplatz benötigt.
Parallelisierung: Die Vergleiche und Vertauschungen können parallel auf verschiedenen Teilen des Arrays durchgeführt werden.

```
def bubble_sort(a):
    n = len(a)
    for i in range(n):
        for j in range(0, n - i - 1):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
```

	Untere Schranke	Obere Schranke
Vergleiche	$\Theta(n^2)$	$\Theta(n^2)$
Sequenz	beliebig	beliebig
Vertauschungen	0	$\Theta(n^2)$
Sequenz	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Laufzeit	$\Theta(n)$	$\Theta(n^2)$
Beschreibung	$1, 2, \ldots, n$	beliebig

Die Laufzeit hängt von der anfänglichen Sortierung ab.

Selection Sort

Beschreibung: In jedem Durchgang wird das kleinste verbleibende Element, mit dem Element beim aktuellen Index vertauscht.
Invariante: Nach dem k-ten Durchlauf sind die ersten k Elemente des Arrays immer die kleinsten und bereits korrekt sortiert.
Speicherplatz: Es wird kein zusätzlicher Speicherplatz benötigt.
Parallelisierung: Die Suche nach dem kleinsten Element in jeder Iteration kann parallelisiert werden.

```python
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]
```

	Untere Schranke	Obere Schranke
Vergleiche	$\Theta(n^2)$	$\Theta(n^2)$
Sequenz	beliebig	beliebig
Vertauschungen	0	$\Theta(n)$
Sequenz	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Laufzeit	$\Theta(n)$	$\Theta(n^2)$
Beschreibung	$1, 2, \ldots, n$	beliebig

Die Laufzeit hängt nicht von der anfänglichen Sortierung ab.

Insertion Sort

Beschreibung: In jedem Durchgang wird ein Element aus dem unsortierten Teil des Arrays entnommen und an der richtigen Position im bereits sortierten Bereich eingefügt.
Invariante: Nach dem k-ten Durchlauf sind die ersten k Elemente des Arrays immer sortiert.
Speicherplatz: Es wird kein zusätzlicher Speicherplatz benötigt.
Parallelisierung: Mehrere Elemente können gleichzeitig in den sortierten Bereich eingefügt werden.

```python
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
```

	Untere Schranke	Obere Schranke
Vergleiche	$\Theta(n)$	$\Theta(n^2)$
Sequenz	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Vertauschungen	0	$\Theta(n^2)$
Sequenz	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Laufzeit	$\Theta(n)$	$\Theta(n^2)$
Beschreibung	$1, 2, \ldots, n$	beliebig

Die Laufzeit hängt von der anfänglichen Sortierung ab.
Vorsicht: Im ersten Durchgang des insertion sort bleibt die Reihenfolge im Array immer gleich.

Merge Sort

Beschreibung: Der Algorithmus teilt das Array rekursiv in zwei Hälften, sortiert diese unabhängig voneinander und fügt sie anschliessend in der richtigen Reihenfolge wieder zusammen.
Invariante: Nach jedem vollständigen Zusammenfügen (Merge-Schritt) sind die bereits kombinierten Teilarrays sortiert.
Speicherplatz: Es wird zusätzlicher Speicherplatz für die temporären Teilarrays benötigt $O(n)$.
Parallelisierung: Die Aufteilung des Arrays und das Sortieren der Teilhälften können parallel ausgeführt werden.

Alorithmus `merge` und `merge_sort`:

```python
def merge(a1, a2):
    b, i, j = [], 0, 0
    while i < len(a1) and j < len(a2):
        if a1[i] < a2[j]:
            b.append(a1[i])
            i += 1
        else:
            b.append(a2[j])
            j += 1
    b += a1[i:]
    b += a2[j:]
    return b

def merge_sort(a):
    if len(a) <= 1:
        return a
    else:
        sorted_a1 = merge_sort(a[:len(a) // 2])
        sorted_a2 = merge_sort(a[len(a) // 2:])
        return merge(sorted_a1, sorted_a2)
```

	Untere Schranke	Obere Schranke
Vergleiche	$\Theta(n \log n)$	$\Theta(n \log n)$
Sequenz	beliebig	beliebig
Vertauschungen	$\Theta(n \log n)$	$\Theta(n \log n)$
Sequenz	beliebig	beliebig
Laufzeit	$\Theta(n \log n)$	$\Theta(n \log n)$
Beschreibung	beliebig	beliebig

Die Laufzeit hängt nicht von der anfänglichen Sortierung ab.

Quicksort

Beschreibung: Der Algorithmus wählt ein Pivot-Element, teilt das Array in zwei Teilarrays (kleinere und grössere Elemente) und sortiert diese rekursiv, bevor sie zusammengefügt werden.
Invariante: Nach jeder Partitionierung befinden sich alle Elemente welche kleiner sind links vom Pivot und alle Elemente, die grösser sind rechts davon.
Speicherplatz: Es wird kein zusätzlicher Speicherplatz benötigt.
Parallelisierung: Die Partitionierung und das rekursive Sortieren der Teilarrays können parallel ausgeführt werden.

Algorithmus `partition`:

```python
def partition(a, l, r):
    p = a[r]
    j = l
    for i in range(l, r):
        if a[i] < p:
            a[i], a[j] = a[j], a[i]
            j += 1
    a[j], a[r] = a[r], a[j]
    return j
```

Algorithmus `quicksort`:

```python
def quicksort(a, l, r):
    if l < r:
        k = partition(a, l, r)
        quicksort(a, l, k - 1)
        quicksort(a, k + 1, r)
```

	Untere Schranke	Obere Schranke
Vergleiche	$\Theta(n \log n)$	$\Theta(n^2)$
Sequenz	(*)	$1, 2, \ldots, n$
Vertauschungen	$\Theta(n)$	$\Theta(n \log n)$
Sequenz	$1, 2, \ldots, n$	(*)
Laufzeit	$\Theta(n \log n)$	$\Theta(n^2)$
Beschreibung	(*)	$1, 2, \ldots, n$

(*) Das Pivot wird immer so gewählt, dass es den Sortierbereich halbiert. Die Laufzeit hängt von der anfänglichen Sortierung ab.
Vorsicht: Die relative Reihenfolge der Elemente links und rechts vom Pivot ändert sich nicht.

Heapsort

Heapsort

Beschreibung: Sei $A[1, \ldots, n]$ ein Heap. Solange $n > 1$:

1. Vertausche $(A[1], A[n])$
2. Versickere $(A, 1, n-1)$
3. $n \leftarrow n - 1$

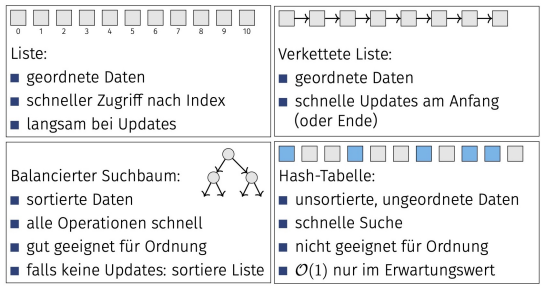
Zunächst wird ein Max-Heap (für aufsteigende Sortierung) oder ein Min-Heap (für absteigende Sortierung) aufgebaut. Anschließend wird das größte (oder kleinste) Element extrahiert und an das Ende der Liste verschoben. Danach wird der Heap wiederhergestellt (`heapify`). Diese beiden Schritte werden wiederholt auf den verbleibenden unsortierten Teil des Arrays angewendet, bis das gesamte Array sortiert ist.
Invariante: Nach jeder Extraktion bleibt die Heap-Eigenschaft erhalten, d.h. jedes Elternknoten-Element ist größer (bei Max-Heap) bzw. kleiner (bei Min-Heap) als seine Kindknoten.
Speicherplatz: Es wird kein zusätzlicher Speicherplatz benötigt, da die Umstrukturierung direkt in der Liste erfolgt (In-Place-Sortierung).
Parallelisierung: Der Aufbau des Heaps und das erneute Heapify-Verfahren können in parallelen Threads implementiert werden, was die Performance verbessert.
Vorsicht: Beim Sift-Down wird das Elter in einem Max-Heap mit dem grösseren der beiden Kinder und beim Min-Heap mit dem kleineren vertauscht.

```python
def swap(list_a, i, j):
    list_a[i], list_a[j] = list_a[j], list_a[i]

def sift_down(list_a, index, size):
    while 2 * index + 1 < size:
        j = 2 * index + 1
        if j + 1 < size and list_a[j] < list_a[j + 1]:
            j += 1

        if list_a[index] < list_a[j]:
            swap(list_a, index, j)
            index = j
        else:
            return

def heapify(list_a):
    n = len(list_a)
    for i in range(n // 2 - 1, -1, -1):
        sift_down(list_a, i, n)

def sort(list_a):
    n = len(list_a)
    heapify(list_a)
    for i in range(n - 1, 0, -1):
        swap(list_a, 0, i)
        sift_down(list_a, 0, i)
```

Die Laufzeit ist in allen Fällen $\Theta(n \log n)$. Die Laufzeit hängt also nicht von der anfänglichen Sortierung ab.

Datenstrukturen

Übersicht:



Listen (und Vektoren)

Operation	Laufzeit
Index-Zugriff	$\mathcal{O}(1)$
Suche (**in**)	$\mathcal{O}(n)$
Sortierte Suche	$\mathcal{O}(\log n)$
Einfügen	$\mathcal{O}(n)$
Entfernen	$\mathcal{O}(n)$

Anmerkungen:

- Das Entfernen des letzten Element ist $\mathcal{O}(1)$.
- In den meisten Fällen ist das Einfügen am Ende $\mathcal{O}(1)$. Ist rechts der Liste jedoch kein Speicherplatz mehr übrig, muss die gesamte Liste verschoben werden, um neuen Speicherplatz zu generieren. In diesem Fall ist die Laufzeit $\mathcal{O}(n)$.

Verkettete Listen

Definition

```python
class Node:
    def __init__(self, value, Next = None):
        self.value = value
        self.Next = Next

class Linked_List:
    def __init__(self, head):
        self.head = head
```

Eine verkettete Liste besteht aus dem Zeiger auf das erste Element.

Traversieren

```python
def print_elements(l):
    """Print all elements in linked list l."""
    current = l.head
    while current != None:
        print(current.value)
        current = current.Next
```

Laufzeit $\in \mathcal{O}(n)$

Letzter Knoten

```python
def last_node(l):
    """Return last node in linked list l."""
    current = l.head
    if current == None:
        return None
    else:
        while current.Next != None:
            current = current.next
        return current
```

Laufzeit $\in \mathcal{O}(n)$

Suche

```python
def search(l, v):
    """Return first node in l with value v or else None."""
    current = l.head
    while current != None:
        if current.value == v:
            return current
        current = current.Next
    return None
```

Laufzeit $\in \mathcal{O}(n)$

Einfügen nach Knoten

```python
def insert_after_node(node, value):
    """Insert new node with value after node."""
    new_node = Node(value, node.Next)
    node.Next = new_node
```

Laufzeit $\in \mathcal{O}(1)$

Einfügen nach Wert

```python
def insert_after_value(l, value, new_value):
    """Insert node with new_value after node with value."""
    node = search(l, value)
    if node != None:
        insert_after_node(node, new_value)
```

Laufzeit $\mathcal{O}(n) + \mathcal{O}(1) \in \mathcal{O}(n)$

Entfernen nach Knoten

```python
def remove_after_node(node):
    """Remove node after node (if it exists)."""
    to_remove = node.Next
    if to_remove == None:
        return
    else:
        node.Next = to_remove.Next
```

Laufzeit $\in \mathcal{O}(1)$

Entfernen nach Wert

```python
def remove_after_value(l, value):
    """Remove node after node with value (if it exists)."""
    node = search(l, value)
    if node != None:
        remove_after_node(node)
```

Laufzeit $\mathcal{O}(n) + \mathcal{O}(1) \in \mathcal{O}(n)$

Erstellen (ohne Hilfszeiger)

```python
def create_from_list(a):
    """Create and return linked list from list a.
    """
    l = Linked_List(Node a[0]))
    for v in a[1:]:
        last = last_node(l)
        last.Next = Node(v)
    return l
```

Problem: Das letzte Element wird immer wieder neu gesucht. Wir haben folglich eine Laufzeit in $\mathcal{O}(n^2)$.

Erstellen (mit Hilfszeiger)

```python
def create_from_list(a):
    """Create and return linked list from list a.
    """
    l = Linked_List(Node(a[0]))
    last = l.head
    for v in a[1:]:
        last.Next = Node(v)
        last = last.Next
    return l
```

Laufzeit in $\mathcal{O}(n)$

(Binäre) Bäume

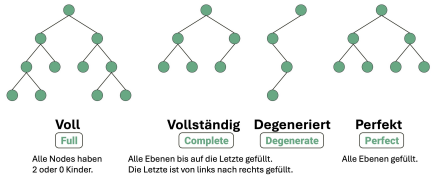
(Die Laufzeiten sind immer bezüglich balancierten Suchbäumen gegeben) Ein binärer Suchbaum erfüllt folgende Eigenschaften:

- Jeder Knoten `v` speichert einen Schlüssen
- Schlüssem im linken Teilbaum `v.left` sind kleiner als `v.key`
- Schlüssen im rechten Teilbaum `v.right` sind grösser als `v.key`

Einige Begriffe:

- Ordnung: maximale Anzahl Kinderkonten
- Höhe: maximale Pfadlänge Wurzel zu Blatt
- Balanciert: die Höhe der linken und rechten Teilbäume an jedem Knoten unterscheiden sich höchstens um 1.
- Degeneriert: Jeder Knoten hat höchstens 1 Kind, sodass der Baum einer verketteten Liste ähnelt.

Aufbaukomplexität: (Worst: $\Theta(n^2)$, Best: $\Theta(n)$)



Die minimale Baumtiefe ist gegeben durch $H_{min} = \lceil \log(n+1) \rceil$ und die maximale Baumtiefe durch $H_{max} = n$.

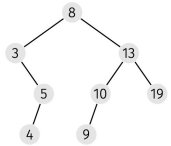
Definition

```python
class Node:
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right

class Tree:
    def __init__(self, root):
        self.root = root
```

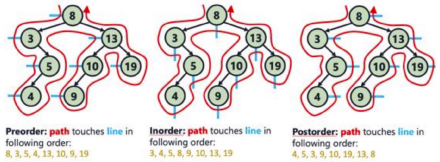
Der Baum ist der Zeiger auf die Wurzel.

Traversierungsarten

- ***Hauptreihenfolge*** (*preorder*):
  v, dann $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- ***Nebenreihenfolge*** (*postorder*):
  $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$, dann v.
  4, 5, 3, 9, 10, 19, 13, 8
- ***Symmetrische Reihenfolge*** (*inorder*):
  $T_{\text{left}}(v)$, dann v, dann $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19



Trick um Traversierungsreihennfolge schneller zu bestimmen:



Traversieren (Preorder)

```python
def visit(node):
    print(node.value)
    if node.left != None:
        visit(node.left)
    if node.right != None:
        visit(node.right)

def visit_tree(tree):
    if tree.root != None:
        visit(tree.root)
```

Für die Postorder Traversal kommt `print(node.value)` nach den beiden `if`-Anweisungen. Für die Inorder Traversal kommt `print(node.value)` zwischen den beiden `if`-Anweisungen.
Laufzeit $\in \mathcal{O}(n)$ für alle Traversierungsarten.

Suche (iterativ)

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

Suche (rekursiv)

```python
def search(node, v):
    if node.value == v:
        return True
    l, r = False, False
    if node.left != None:
        l = search(node.left, v)
    if node.right != None:
        r = search(node.right, v)
    return l or r
```

Laufzeit $\in \mathcal{O}(\log n)$

Knoten einfügen

```python
def addNode(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
```

Laufzeit $\in \mathcal{O}(\log n)$

Knoten entfernen

```python
def symmetric_desc(start_node: Node):
    if start_node.left is None:
        return start_node.right

    if start_node.right is None:
        return start_node.left

    parent = None
    node = start_node.right
    while node.left is not None:
        parent = node
        node = node.left

    if parent is not None:
        parent.left = node.right
        node.right = start_node.right
    node.left = start_node.left

    return node

def remove(self, key: int) -> bool:
    # case when we are deleting the root node
    if self.root is not None and self.root.key == key:
        self.root = symmetric_desc(self.root)
        return True

    node = self.root
    while node is not None:
        # key as the left or right child of current code
        if node.left is not None and node.left.key == key:
            node.left = symmetric_desc(node.left)
            return True

        if node.right is not None and node.right.key == key:
            node.right = symmetric_desc(node.right)
            return True

        # have not found the key, keep searching
        if key < node.key:
            node = node.left
        else:
            node = node.right
        return False
```

Der symmetrische Nachfolger ist der kleinste Knoten im rechten Teilbaum (oder der grösste Knoten im linken Teilbaum) des Knoten, welcher entfernt werden soll.
Laufzeit $\in \mathcal{O}(\log n)$

Heaps

Ein Heap ist ein binärer Baum mit folgenden Eigenschaften:

- vollständig bis auf die letzte Ebene
- Lücken des Baumes in der letzten Ebene höchstens rechts
- **Heap-Bedingung**: Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Elternknotens

Implementierung eines Heaps als Array:

- Arrays mit Indexierung die bei 0 beginnt: Kinder$(i) = \{2i+1, 2i+2\}$ und Elter$(i) = \lfloor (i-1)/2 \rfloor$
- Arrays mit Indexierung die bei 1 beginnt: Kinder$(i) = \{2i, 2i+1\}$ und Elter$(i) = \lfloor i/2 \rfloor$

Die Höhe eines Heaps mit n Knoten ist $H(n) = \lceil \log_2(n+1) \rceil$.
Aufbaukomplexität: (Worst: $\Theta(n \log n)$, Best: $\Theta(n)$)

Hash-Tabellen

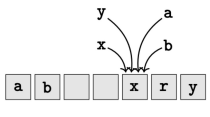
Tabelle (Array) der Grösse M. Eine mathematische Hash-Funktion berechnet für ein Element den entsprechenden Index.
Was machen wir, wenn zwei Element denselben Index haben?

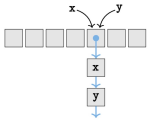
Probing

- Einfach rechts / links vom Index versuchen, solange bis ein freier Platz kommt.

Chaining

- Aus jedem Eintrag in der Tabelle eine linked list machen.





Vorsicht: Kollisionen können mittels probing in beide Richtungen gelöst werden. Schaue was in der Aufgabenstellung spezifiziert wird.

- mit Kollisionen unter Annahme von guter Hash-Funktion

Operation	erwartete Laufzeit	worst-case-Laufzeit
Suche	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Einfügen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Entfernen	$\mathcal{O}(1)$	$\mathcal{O}(n)$

$\Rightarrow$ in der Praxis normalerweise ziemlich schnell

Andere Datenstrukturen

Stack

Ein Stack ist eine lineare Datenstruktur, die nach dem LIFO-Prinzip (Last In, First Out) arbeitet. Das bedeutet, dass das zuletzt hinzugefügte Element als erstes wieder entfernt wird – ähnlich einem Stapel von Tellern.

Operation	Beschreibung
`push(x)`	Fügt das Element `x` oben auf den Stack.
`pop()`	Entfernt das oberste Element und gibt es zurück.
`is_empty()`	Prüft, ob der Stack leer ist.

Eigenschaften:

- LIFO-Prinzip: (Last In, First Out)
- Konstante Zeit für `push()` und `pop()`
- Begrenzter Zugriff (man kann nur das oberste Element direkt manipulieren)

Anmerkung: Die Elemente werden vorne angehängt und vorne entfernt.
Anwendungen: Speicherverwaltung (Aufrufstapel in Programmen), Browser-Zurück-Funktion (History-Stack)

Queue

Eine Queue ist eine lineare Datenstruktur, die nach dem FIFO-Prinzip (First In, First Out) arbeitet. Das bedeutet, dass das zuerst eingefügte Element auch als erstes wieder entfernt wird – ähnlich einer Warteschlange an der Kasse.

Operation	Beschreibung
`enqueue(x)`	Fügt das Element x hinten in die Queue ein.
`dequeue()`	Entfernt das vorderste Element und gibt es zurück.
`front()`	Gibt das erste Element der Queue zurück, ohne es zu entfernen.
`is_empty()`	Prüft, ob die Queue leer ist.

- FIFO-Prinzip: (First In, First Out)
- Konstante Zeit für `enqueue()` und `dequeue()`
- Beschränkter Zugriff (Elemente können nur von vorne entfernt und hinten hinzugefügt werden).

Anmerkung: Die Elemente werden hinten angehängt und vorne entfernt.

Quad Tree

Ein Quadtree ist eine hierarchische Baumstruktur, die rekursiv ein zweidimensionales Gebiet in vier gleich große Teilbereiche (Quadranten) unterteilt. Diese Datenstruktur wird häufig für räumliche Partitionierung verwendet.

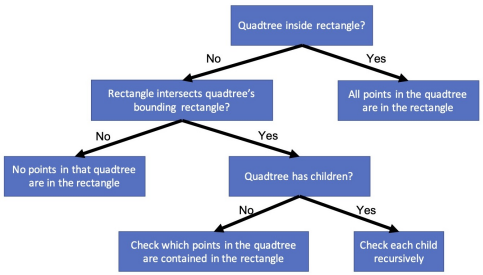
Grundprinzip:

- Jeder innere Knoten hat genau vier Kinder (Quadranten).
- Die Unterteilung erfolgt rekursiv, bis eine bestimmte Bedingung erfüllt ist (z. B. eine maximale Anzahl an Objekten pro Bereich).
- In den Blattknoten werden die Daten gespeichert.

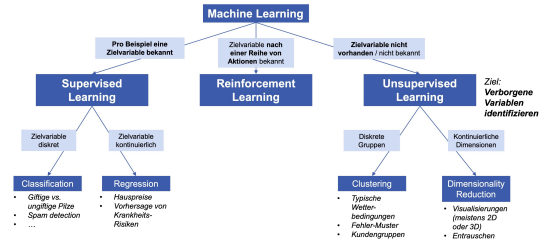
Eigenschaften:

- Effiziente Partitionierung für 2D-Raum.
- Speicherplatz-Optimierung: Weniger Speicherbedarf als ein Gitter (Grid) bei dünn besetzten Bereichen
- Schnelle Suche in $\mathcal{O}(\log n)$ bei gut balancierten Bäumen.

Algorithmus zur Bereichssuche in einem Quadtree:



Machine Learning: Theorie



Classification:

- Decision Trees
- Logistische Regression
- Neuronale Netzwerke

Regression:

- Lineare Regression
- Decision Trees
- Neuronale Netzwerke

Confusion Matrix

Normalisierte Verwechslungsmatrix

		Predicted Label	
		Ungiftig	Giftig
True Label	Ungiftig	1.0	0.0
	Giftig	1.0	0.0

Balanced accuracy rate BAR:

- Durchschnittliche Genauigkeit über alle Klassen
- Mittelwert der Diagonale der normalisierten Verwechslungsmatrix

Hier: BAR = 50%

Wir lesen die Verwechslungsmatrix wie folgt: 100 Prozent der Ungiftigen werden als ungiftig vorhergesagt. 100 Prozent der Giftigen werden als ungiftig vorhergesagt. Des weiteren ist die Summe pro Zeile immer 1 und der Mittelwert der Diagonale die Genauigkeit.

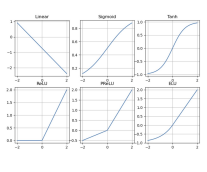
Modelle

Entscheidungsbäume

Entscheidungen in Form eines Baumdiagramms. Durch rekursive binäre splits findet eine Paritionierung des Merkmal-Raums (feauture space) statt. Wir teilen den Raum solange auf, bis wir nur Gruppen noch haben, die gleichförmig bezüglich des Ziellabels sind (dh. Gini-Koeffizient $= 0$).

Neuronale Netzwerke

Schichten von Knotenpunkten (Neuronen), die Eingabeinformationen verarbeiten. Ein Neuron ist eine Linearkombination der Eingaben in der Schicht davor, gefolgt von einer Aktivierungsfunktion. Nichtlineare Aktivierungsfunktionen ermöglichen die Modellierung komplexer abhängigkeiten:



Name in PyTorch	Equation
`Linear`	$f(x) = a \cdot x + b$
`Sigmoid`	$f(x) = \frac{1}{1+e^{-x}}$
`Tanh`	$f(x) = \tanh(x)$
`ReLU` (Rectified Linear Unit)	$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$
`PReLU` (Parametric ReLU)	$f(x) = \begin{cases} a \cdot x, & x < 0 \\ x, & x \geq 0 \end{cases}$
`ELU` (Exponential L.U.)	$f(x) = \begin{cases} a(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$

Wegen der nichtlinearen Aktivierungsfunktion ist keine analytische Lösung des neuronalen Netzwerk möglich. Wir führen eine iterative Optimierung durch.
Convolutional Filter: Kleine Matrix, die über eine größere Eingabematrix gleitet und Merkmale extrahiert, indem sie eine gewichtete Summe der überlagerten Werte berechnet. Beispiel:

$$A = \begin{bmatrix} 5 & 2 & 3 \\ 0 & 8 & -1 \\ -1 & 2 & 0 \end{bmatrix} \qquad f = \begin{bmatrix} 2 & -1 \\ 1 & -2 \end{bmatrix}$$

Der Filter f kann in vier Positionen über die Matrix A platziert werden (oben links, oben rechts, unten links unten rechts). Sind die Matrizen an einer Position überlagert, so werden die entsprechenden Einträge miteinander multipliziert und dann aufsummiert. Schliesslich wird noch die ReLU-Funktion angewandt (siehe oben), um die entsprechenden Einträge von a zu erhalten. Wir betrachten die Position oben Links: $a[0][0] = \text{ReLU}(10 - 2 - 16) = 0$. Analog $a[1][0] = 11$, $a[0][1] = 0$, und $a[1][1] = 19$.

Nicht-numerischer Attribute mittels One-hot encoding

Daten enthalten oft nicht-numerische Angaben. Um diese für Berechnungen nutzbar zu machen, werden sie mithilfe von One-hot encoding wie folgt transformiert:

- Definiere für jeden möglichen Wert eine entsprechende binäre Variable.
- Die Variable ist 1 (oder True) genau dann, wenn das Attribut den entsprechenden Wert hat, sonst 0.

Eigenschaften von One-hot encoding sind:

- Enthält keine Information über Ähnlichkeit. Zwei Objekte sind bezüglich einer Variable entweder gleich oder verschieden.
- Bei N verschiedenen Attribut-Werten entstehen aus 1 Attribut N Attribute. Beispiel: Es sei ein Datensatz mit der Dimension (10×1) gegeben (10 Beispiele und 1 Feature). Für dieses Feature gibt es vier mögliche Werte. Durch One-hot encoding durch erhält man ein Datensatz mit der Dimension (10×4). Dimension der Beispiele bleibt gleich, Dimension der Feautres wird verändert.
- Die Darstellung ist redundant, da immer genau 1 der neuen Attribute 1 ist, alle anderen sind 0. Ein Attribut bzw. eine Spalte kann somit immer weggelassen werden.

Beispiel:
Gegeben sei ein Datensatz mit 3 Spalten:

- `name`: String-Werte mit 35 verschiedenen Kategorien
- `married`: Boolscher Wert (Ja/ Nein)
- age: anzzahlige Werte mit 12 verschiedenen Altersstufen

Ziel ist es, den Datensatz in ein Format zu bringen, das nur numerische Werte enthält mit der minimalen Anzahl von Spalten. Die minimale Anzahl von Spalten im neuen Datensatz ist 36. Wie kommt man darauf? `married` ist ein boolscher Wert, der als numerische Spalte codiert werden kann (0 oder 1). Die Werte in der Spalte age sind bereits numerische Werte. Für `name` wenden wir folgende Regel an: One-Hot Encoding für eine Kategorie mit k Werten erzeugt $k - 1$ Spalten (da eine Spalte redundant ist). Wir erhalten für die 35 Namen also 34 Spalten. Somit ist die minimale Anzahl an Spalten 36. Die maximale Anzahl an Spalten wäre 49.

Modellkomplexität

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

- **Underfitting:** Das gelernte Modell ist sehr einfach und kann die Struktur der Daten nicht (vollständig) abbilden.
- **Overfitting:** Das gelernte Modell ist sehr komplex und passt sich übermässig an die Trainingsdaten und das darin enthaltene Rauschen ("noise") an.

Bemerkungen:

- Over- und Underfitting können nur aus dem Vergleich der Performance auf den Trainings- und Testdaten identifiziert werden.
- Die Baumtiefe ist ein Hyperparameter, welcher die Komplexität des Modells definiert. Innerhalb des Modells sind die Hyperparameter fix, sie werden ausserhalb festgesetzt, z.B. mittels Crossvalidation.

Dimensionsreduktion (PCA)

Ab einem Punkt nimmt die Performance von Modellen bezüglich der Loss-Funktion mit zunehmender Dimensionalität bzw. mit einer höheren Anzahl Merkmalen ab. Principal component analysis **(PCA)**: Die Dimension des Datensatz wird in die Richtungen reduziert, in der die Streuung (Varianz) der Daten maximal ist. Diese Richtungen werden auch Hauptkomponenten genannt. Die Projektion auf die Hauptkomponenten ist linear. Dies ist wünschenswert, da in der Streuung Information enthalten ist. Alle Projektionsrichtungen (Hauptkomponenten) sind orthogonal zueinander.

Grid Search via Cross Validation (CV)

Systematische Durchsuchung einer vordefinierte Menge an Hyperparameter-Kombinationen wobei jede Kombination anhand von Kreuzvalidierung bewertet wird, um die beste Konfiguration für ein Modell zu finden.

- Typischerweise wird k-fache Kreuzvalidierung mit $k = 5$ oder $k = 10$ verwenden. Dabei sind 20 bzw. 10% der Gesamtdaten jeweils Validierungsdaten.
- Spezialfall: Leave-one-out (LOO) Kreuzvalidierung: jeder Datenpunkt genau 1 mal als Validierungsset (der Grösse 1) verwendet. Entspricht bei n Datenpunkten k-facher Kreuzvalidierung mit $k = n$.
- **Vorsicht:** Kreuzvalidierung findet immer auf den Trainingsdaten statt. Es gibt weiterhin eine Trennung zwischen Trainings- und Testdaten. Wir benötigen die Testdaten, um das schlussendliche Modell testen zu können.

Clustering / K-Means

Ein Clustering-Modell teilt Daten unbeaufsichtigt in Gruppen, indem es die Summe der quadratischen Abstände der Punkte zu ihren zugewiesenen Zentren minimiert. Der entsprechende Trainingsalgorithmus (auch Lloyd Algorithmus) funktioniert wie folgt:

1. Zufällige Initialisierung der Zentren
2. Aktualisierung der Zuweisung
3. Aktualisierung der Zentroide: Jedes Zentroid wird auf den Durchschnitt aller Punkte gesetzt, die diesem Zentroid zugewiesen sind.

Wobei die Schritte 2 und 3 iterativ wiederholt werden.
Anmerkungen:

- Das Modell bestimmt nicht die Anzahl K von Clustern. Dies ist ein Hyperparameter und man muss es selbst festlegen.
- Die Cluster Zentren müssen nicht Datenpunkte sein.
- Das Modell ist empfindlich gegenüber der Initialisierung.
- Der Algorithmus konvergiert immer in einer endlichen Anzahl Schritten, kann aber in einem lokalen Optimum hängen bleiben.

Machine Learning: Coding Aufgaben

Daten vorbereiten

Daten einlesen von `"data.csv"`:

```
import pandas as pd

df = pd.read_csv("data.csv")

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

Mit `df.drop(['target], axis = 1)` werden alle Spalten ausser der Spalte `'target'` extrahiert und in `X` gespeichert. Dies sind die Eingabevariablen (Features). Die Spalte `'target'`, die das vorherzusagende Ergebnis enthält, wird separat in `y` gespeichert.

Daten in Trainings- und Testdaten aufteilen:

```
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 = 0)
```

Der Code teilt die Daten zufällig in ein Trainings- und Testset auf, wobei 80% der Daten für das Training und 20% für das Testen verwendet werden. Dies dient dazu, ein Machine-Learning-Modell auf dem Trainingsset zu trainieren und seine Leistung auf dem Testset zuevaluieren.

Modell wählen und trainieren

Wir können von den folgenden Modellen wählen:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
```

DecisionTreeClassifier

Entscheidungen in Form eines Baumdiagramms:

```
# Ohne GridSearch:
model = DecisionTreeClassifier(max_depth = 5,
      random_state = 0)

# Mit GridSearch:
model = DecisionTreeClassifier()
parameters = {"max_depth" : range(1, 10)}

grid = GridSearchCV(model, parameters, cv = 10)
grid.fit(X_train, y_train)
```

Der Hyperparameter für den DecisionTreeClassifier ist die maximale Tiefe des Baums. In der zweiten Zeile wird eine Parameter-Raster für den Hyperparameter im Bereich von 1 bis 9 festgelegt. `GridSearch` führt eine systematische Suche über die angegebenen Hyperparameter durch, wobei cv = 10 bedeutet, dass eine 10-fache Kreuzvalidierung verwendet wird.

DecisionTreeRegressor

```
model = DecisionTreeRegressor()
parameters = {
    "max_depth" : range(1, 10),
    "criterion" : ["squared_error", "friedman_mse"]
}
grid = GridSearchCV(model, parameters, cv = 10)
grid.fit(X_train, y_train)
```

LinearRegression

```
linreg = LinearRegression()
linreg.fit(X_train, y_train)
```

Wir benötigen keinen Hyperparameter, um ein Modell mit linearer Regression zu trainieren.

LogisticRegression

```
logreg = LogisticRegression(max_iter = 1000)
logreg.fit(X_train, y_train)
```

`max_iter` = 1000 legt die maximale Anzahl Iterationen für den Optimierungsalogorithmus auf 1000 fest, um eine bessere Konvergenz sicherzustellen.

MLPClassifier

Wir verwenden den MLPClassifier für diskrete Labels.

```
mlp = MLPClassifier(hidden_layer_sizes = (32, 16,
      8), max_iter = 5000, random_state = 0)
mlp.fit(X_train, y_train)
```

Der Hyperparameter des MLPClassifier ist die `hidden_layer_sizes`.
Modellinitialisierung:

- `hidden_layer_sizes` = (32, 16, 8): Definiert die Struktur des neuronalen Netzwerks mit jeweils 32, 16 und 8 Neuronen. Mehr Schichten und Neuronen können die Modellleistung erhöhen, aber auch das Risiko von Overfitting erhöhen.
- `max_iter` = 5000: Legt fest, dass das Modell bis zu 5000 Iterationen für die Optimierung (Training) durchführt. Eine höhere Anzahl kann helfen Konvergenzprobleme zu vermeiden.
- `random_state` = 0: Stellt sicher, dass die Zufallsinitialisierung der Gewichte reproduzierbar ist.

MLPRegressor

Wir verwenden den MLPRegressor für kontinuierliche Werte.

```
mlp = MLPRegressor(hidden_layer_sizes = (32, 16, 8)
      , max_iter = 5000, random_state = 0)
mlp.fit(X_train, y_train)
```

Modell testen

Wir messen die Qualität des Testers anhand der Testdaten:

```
from sklearn.metrics import mean_squared_error,
      r2_score, accuracy_score

y_pred = grid.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print("MSE on test data: {:.3f}".format(mse))
print("R2 on test data: {:.3f}".format(r2))
print("Accuracy on test data: {:.3f}".format(
      accuracy))
```

Die Genauigkeit maschineller Modelle ergibt sich aus der Abweichung ihrer Vorhersagen zu den tatsächlichen Werten:
Mean Squared Error: Mittlere quadratische Abweichung der Vorhersagen und eigentlichen Werten (Best: 0, Wost: $+\infty$). Wann verwenden?

- Wenn du große Fehler stärker bestrafen willst (durch das Quadrieren der Fehler).
- Wenn du möchtest, dass das Modell keine großen Abweichungen macht.

R2-Score: Genauigkeit im Vergleich zum Durchschnitt (Best: 1, Worst: $-\infty$). Wann verwenden? Wenn du wissen willst, wie gut dein Modell die Variabilität der Daten erklärt.
Accuracy score: Prozent der korrekt klassifizierten Datensätze (Best: 1, Worst: 0). Wann verwenden?

- Wenn die Klassen gleichmässig verteilt sind.
- Wenn Fehlklassifikationen in jeder Klasse gleich schlimm sind.

Balanced Accuracy Rate: (Best: 1, Worst: 0). Wann verwenden?

- Wenn die Klassen unausgewogen sind.
- Wenn jede Klasse gleich wichtig ist.
- Wenn du verhindern möchtest, dass das Modell eine Klasse bevorzugt.

Die Standard-Fehlerfunktion für Klassifizierung ist der `accuracy_score` und für Regression der `r2_score`

Vorhersage machen

Wir verwenden den Schätzer, um Vorhersagen auf Grundlage neuer Daten zu treffen.

```
X_final = pd.read_csv("X_final.csv")
y_final = grid.predict(X_final)
return y_final
```

Coding Beispiele

Dynamic Programming

- zu kleine Tabelle? `T = [None] * n` vs `T = [None] * (n+1)`
- off by one? `T[n]` vs `T[n-1]` und `T[0]` vs `T[1]`
- falsche Initialisierung der Tabelle?
  `T = [ [None] * m ] * n` vs `T = [ [None] * m for i in range(n) ]`
- Memoisierung:
  - Tabelle nicht übergeben bei rekursivem Aufruf? `f_mem(i)` vs `f_mem(i, T)`
  - Wert berechnet aber nicht abgespeichert?
    `return x` vs `T[i] = x; return T[i]`
  - vergessen zu überprüfen, ob Wert bereits berechnet wurde?
- DP:
  - nicht genügend Randfälle?
  - falsche Berechnungsreihenfolge (Zugriff auf uninitialisierte Einträge)

Tribonacci

Für Tribonacci-Zahlen gilt: $a_0 = a_1 = 0$, $a_2 = 1$ und für $n \geq 3$: $a_n = a_{n-1} + a_{n-2} + a_{n-3}$.

```
def tribonacci(n, memo=None):
    if memo is None:
        memo = [None] * (n + 1)

    if memo[n] is None:
        if n <= 1:
            memo[n] = 0
        elif n == 2:
            memo[n] = 1
        else:
            memo[n] = tribonacci(n - 1, memo) +
     tribonacci(n - 2, memo) + tribonacci(n - 3,
     memo)

    return memo[n]
```

Schnellster Weg zur anderen Seite

Ziel: Spielfeld von links nach rechts mit minimalen Schritten zu durchqueren, wobei die Zahl auf jedem Feld angibt, wie weit man sich maximal nach vorne bewegen darf.
Rekurrenz:

$$S(i) = \begin{cases} 0 & i = n \\ 1 + \min\{S(i+1), \ldots, S(i+a[i])\} & 0 \leq i < n \end{cases}$$

Code:

```
def jumps(a):
    n = len(a)
    s = [None] * (n + 1)

    for i in range(n, -1, -1):
        if i == n:
            s[i] = 0
        else:
            s[i] = 1 + min([s[i + j]
                    for j in range(1, a[i] + 1)])

    return s[0]
```

Aufgabenplanung

Ziel: Eine Menge von Aufgaben so wählen, dass die Gesamtbelohnung maximiert wird, wobei jede Aufgabe eine bestimmte Dauer und eine Frist hat und sich Aufgaben nicht überlappen dürfen.
Rekurrenz:

$$S(i) = \begin{cases} 0 & i = n \\ \max\{S(i+1), w[i] + S(i+t[i])\} & 0 \leq i < n \end{cases}$$

Code:

```
def tasks(w, t):
    n = len(w)
    s = [None] * (n + 1)

    for i in range(n, -1, -1):
        if i == n:
            s[i] = 0
        else:
            finish_time = min(n, i + t[i])
            s[i] = max(w[i] +
                    s[finish_time], s[i + 1])

    return s[0]
```

Aufgabenplanung V2)

Ziel: Optimale Auswahl an Aufgaben zu treffen, um die maximale Gesamtbelohnung zu erzielen, wobei zu jedem Zeitpunkt nur eine von zwei möglichen Aufgaben gewählt werden kann und deren Dauer berücksichtigt werden muss.

```
def tasks(w1, t1, w2, t2):
    n = len(w1)
    s = [None] * (n + 1)
    s[n] = 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]
```

Blöcke

Gibt eine Liste boolescher Werte zurück, die annotieren, ob es möglich ist, einen Block der Länge i zu erstellen (i = Index des booleschen Werts in der Liste) mit i ¡ n und Blöcken als Liste der verfügbaren Unterblocklängen.

```
def what_is_possible(n, blocks):
    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
```

Schneiden von Eisenstäben

Ziel: Eisenstab in Stücke schneiden, um den maximalen Gesamtwert zu erzielen. Jedes mögliche Stück eine bestimmte Länge und einen zugehörigen Preis. Die Herausforderung besteht darin, die optimale Zerlegung des Stabs zu finden, sodass die Summe der Preise der Teilstücke maximiert wird.
Code (Rekursiv):

```
def max_val_cuts(w, n):
    if n == 0:
        return 0

    vals = [w[j] + max_val_cuts(w, n-j)
                for j in range(1, n + 1)]
    return max(vals)
```

Code (DP):

```
def max_value_cuts(w, n):
    s = [None] + (n + 1)
    s[0] = 0
    for i in range(1, n + 1):
        values = [w[j] + s[i - j]
                    for j in range(1, i + 1)]
        s[i] = max(values)
    return s[n]
```

Code (Konstruktion des optimalen Schnittes)

```
def max_value_cuts_with_cut(w, n):
    s = [None] * (n + 1)
    L = [None] * (n + 1)
    s[0], L[1] = 0, 1
    for i in range(1, n + 1):
        values = [(w[j] + s[i - j], j)
                    for j in range(1, i + 1)]
        maxi = max(values)
        s[i], L[i] = maxi
    i, cut = n, []
    while i >= 1:
        cut.append(L[i])
        i -= L[i]
    return (s[n], cut)
```

Triangle

Gibt den maximal möglichen Wert entlang eines Pfads vom Scheitelpunkt zurück zur Basis eines gegebenen Dreiecks.

```
    def best_path(triangle):
    m = len(triangle)

    s = [[None for j in range(m)] for i in range(m)
     ]

    for i in range(m - 1, -1, -1):
        for j in range(i, -1, -1):
            if i == m - 1:
                s[i][j] = triangle[i][j]
            else:
                s[i][j] = triangle[i][j] + max(s[i
     + 1][j], s[i + 1][j + 1])

    return s[0][0]
```

Knights Way

```
def best_way(a):
    return dp_best_way(a)

def rec_best_way(a, i=0, j=0):
    n, m = len(a), len(a[0])

    reward1, reward2, reward3, reward4 = 0, 0, 0, 0
    if i - 2 >= 0 and j + 1 < m:
        reward1 = rec_best_way(a, i - 2, j + 1)
    if i - 1 >= 0 and j + 2 < m:
        reward2 = rec_best_way(a, i - 1, j + 2)
    if i + 1 < n and j + 2 < m:
        reward3 = rec_best_way(a, i + 1, j + 2)
    if i + 2 < n and j + 1 < m:
        reward4 = rec_best_way(a, i + 2, j + 1)

    return a[i][j] + max(reward1, reward2, reward3,
      reward4)

def dp_best_way(a):
    n, m = len(a), len(a[0])

    s = [[None for _ in range(m)] for _ in range(n)
     ]

    for i in range(n - 1, -1, -1):
        for j in range(m - 1, -1, -1):
            reward1, reward2, reward3, reward4
                    = 0, 0, 0, 0
            if i - 2 >= 0 and j + 1 < m:
                reward1 = s[i - 2][j + 1]
            if i - 1 >= 0 and j + 2 < m:
                reward2 = s[i - 1][j + 2]
            if i + 1 < n and j + 2 < m:
                reward3 = s[i + 1][j + 2]
            if i + 2 < n and j + 1 < m:
                reward4 = s[i + 2][j + 1]

            s[i][j] = a[i][j] + max(reward1,
     reward2, reward3, reward4)

    return s[0][0]
```

Vorsicht: Beim Kopieren des rekursiven Code muss man aufpassen, dass die Einrückung korrekt ist. Ansonsten treten Fehler auf.

Längste gemeinsame Teilfolge

Der **Longest Common Subsequence (LCS)**-Algorithmus bestimmt die Länge der längsten gemeinsamen Teilsequenz zweier Zeichenketten. Er kann entweder rekursiv oder mit dynamischer Programmierung berechnet werden.

```
def S(seq1, i, seq2, j): # rekursive Funktion
    if i == len(seq1) or j == len(seq2):
        return 0
    elif seq1[i] == seq2[j]:
        return 1 + S(seq1, i + 1, seq2, j + 1)
    else:
        return max(S(seq1, i + 1, seq2, j), S(seq1,
     i, seq2, j + 1))

def lcs(seq1, seq2): # dynamische Programmierung
    m = len(seq1)
    n = len(seq2)

    s = [[None for j in range(n + 1)] for i in
     range(m + 1)]  # s[i][j]

    for i in range(m, -1, -1):
        for j in range(n, -1, -1):
            if i == m or j == n:
                s[i][j] = 0
            elif seq1[i] == seq2[j]:
                s[i][j] = 1 + s[i + 1][j + 1]
            else:
                s[i][j] = max(s[i + 1][j], s[i][j +
     1])

    return s[0][0]
```

Anmerkung: Im Gegensatz zum `Knights Way`-Problem muss hier die geeignete Datenstruktur selbst festgelegt werden. Dafür verwenden wir eine **Matrix** der Größe $(\text{len(seq1)}+1) \times (\text{len(seq2)}+1)$. Der Grund für die zusätzlichen Zeilen und Spalten liegt in der Rekursionsstruktur: Der Basisfall tritt auf, wenn $i = \text{len(seq1)}$ oder $j = \text{len(seq2)}$. Damit diese Basisfälle korrekt in der dynamischen Programmierung (DP)-Tabelle abgebildet werden können, muss sie einen zusätzlichen Index enthalten. Dies stellt sicher, dass auch die Randwerte richtig verarbeitet werden.

Strings

Palindrome Checker

```
def is_palindrome(word):
    for i in range(0, len(word) // 2):
        if word[i] != word[-1 - i]:
            return False
    return True
```

String Reverse

Gibt einen String zurück, dessen Längenintervallabschnitte umgekehrt sind.

```
def reverse_string(text, interval):
    n = len(text)
    text_list = list(text)
    # Konvertiere den String in eine Liste

    for i in range(0, n, interval):
        text_list[i:i + interval] =
        text_list[i:i + interval][::-1]
        # In-Place Reverse

    return ''.join(text_list)
    # Konvertiert Liste in String
```

Wir konvertieren den String in eine Liste, da man diese veränderlich ist und somit direkt bearbeitet werden kann. Am Schluss fügen wir die einzelnen Charakter in der Liste wieder zusammen.

Dutch Flag

```
def dutch_flag(arr):
    i = 0
    j = 0
    k = len(arr) - 1

    while j <= k:
        if arr[j] == 'R':
            arr[j], arr[i] = arr[i], arr[j]
            i += 1
            j += 1
        elif arr[j] == 'B':
            arr[j], arr[k] = arr[k], arr[j]
            k -= 1
        else:  # arr[j] == 'W'
            j += 1

    return arr
```

Die Variablen `i`, `j` und `k` repräsentieren drei Zeiger:

- `i`: Position, an die das nächste 'R' (Rot) platziert werden soll.
- `j`: Der aktuelle Index, der überprüft wird.
- `k`: Position, an die das nächste 'B' (Blau) platziert werden soll.

Warum wird `j` in `if` und `else`, aber nicht in `elif` aktualisiert? In der `elif`-Bedingung bleibt das `j` unverändert, da das neue Element, dass nach dem Tausch an Position `j` steht noch überprüft werden muss. Ohne diesen Mechanismus könnte ein 'B', das von `k` nach `j` getauscht wurde, übersehen werden.

Polnische Flagge

Sortiert eine Liste aus den Zeichen 'R' (rot) und 'W' (weiß) so, dass alle 'W'-Elemente vor den 'R'-Elementen stehen.

```
def pol_flag(a):
    # Invariante: a[:j] enthaelt nur 'W' und a[j:i]
      enthaelt nur 'R'.
    n = len(a)
    i = j = 0

    for i in range(n):
        if a[i] == 'R':
            pass
        else:  # a[i] == 'W'
            a[i], a[j] = a[j], a[i]
            j += 1

    return a
```

Längstes gemeinsames Präfix

Nimmt eine Liste von Zeichenfolgen und gibt eine Zeichenfolge aus, die das gemeinsame Präfix aller dieser Zeichenfolgen in der Eingabeliste enthält.

```
def common_prefix_of_two(str1, str2):
    i = 0
    min_len = min(len(str1), len(str2))

    while i < min_len and str1[i] == str2[i]:
        i += 1

    return str1[:i]

def common_prefix(words):
    if len(words) == 1:
        return words[0]
    else:
        mid_point = len(words) // 2
        str1 = common_prefix(words[:mid_point])
        str2 = common_prefix(words[mid_point:])
        return common_prefix_of_two(str1, str2)
```

Listen

Sublist

Gibt True zurück, wenn `[1, 2, ..., n]` eine Teilsequenz von `L` ist.

```
def sub_int(L, n):
    m = len(L)
    i = 0

    while i <= m - n:
        if L[i] == 1:
            match = True
            for j in range(1, n + 1):
                if L[i + j - 1] != j:
                    match = False
                    break
            if match:
                return True

        i = i + 1
    return False
```

Distance

Gibt True zurück, wenn der Abstand zwischen zwei aufeinanderfolgenden Nicht-Null-Elementen in `q` mindestens `d` beträgt.

```
def verify_pops(q, d):
    space = d

    for element in q:
        if element == 0:
            space += 1
        elif space < d:
            return False
        else:
            space = 0

    return True
```

Summe zweier grosser Ganzzahlen

Die Funktion nimmt zwei Listen mit ganzen Zahlen num1 und num2, wobei jede ganze Zahl zwischen 0 und 9 liegt. Jede Liste zeigt eine einzelne Ganzzahl an. Diese Ganzzahlen werden addiert und die Summe wird auf die gleiche Weise zurückgegeben wie Ganzzahlen in einer Liste.

```
def add(num1, num2):
    if len(num1) < len(num2):
        num1 = [0] * (len(num2) - len(num1)) + num1
    if len(num2) < len(num1):
        num2 = [0] * (len(num1) - len(num2)) + num2

    result = [0] * (len(num1) + 1)
    carry = 0

    # Invariant:
    # carry * 10^i + result[-1-i:] displays the sum
      of the arrays
    #     displayed by num1[-1-i:] and num2[-1-i:]
    for i in range(len(num1)):
        result[-1-i] = num1[-1-i] + num2[-1-i] +
     carry
        carry = result[-1-i] // 10
        result[-1-i] %= 10

    result[0] = carry
    return result
```

Search and Sort

Verbesserter Insertion Sort

Verbesserte Einfügungssortierung mit binärer Suche.

```
def insertion_sort_binary(li):
    for i in range(1, len(li)):
        val = li[i]
        j = binary_search(li, val, i - 1)
        li = li[:j] + [val] + li[j:i] + li[i + 1:]

    return li


def binary_search(li, val, right):
    left = 0

    while left <= right:
        mid = (left + right) // 2

        if li[mid] == val:
            return mid
        elif li[mid] < val:
            left = mid + 1
        else:
            right = mid - 1

    return left
```

k-kleinstes Element

Findet das k-kleinste Element in einer Liste von ganzen Zahlen arr.

```python
def find_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 find_kth_smallest(left, k)
    elif k > len(arr) - len(right):
        return find_kth_smallest(right, k - (len(arr) - len(right)))
    else:
        return pivot
```

Datenstrukturen

Table Reservation System

Das Reservierungssystem wird als Suchbaum verwaltet, wobei jede Reservierung als ein Knoten im Baum gespeichert wird, basierend auf der Reservierungszeit als Schlüssel. Neue Reservierungen werden nur hinzugefügt, wenn sie sich nicht innerhalb von 30 Minuten einer bestehenden Reservierung befinden, um Konflikte zu vermeiden. Zusätzlich wird für jeden Knoten die Größe des Teilbaums aktualisiert, sodass mit der `rank()`-Methode schnell berechnet werden kann, wie viele Reservierungen vor oder genau zur angegebenen Zeit existieren.

```python
from typing import Optional

class Node:
    def __init__(self, reservation_time: int):
        self.key = reservation_time
        self.left = None
        self.right = None
        self.size = 1

class SearchTree:
    def __init__(self):
        self.root: Optional[Node] = None

    def add(self, reservation_time: int) -> bool:
        if self.root is None:
            self.root = Node(reservation_time)
            return True

        node = self.root
        while True:
            if reservation_time < node.key and node.left is None:
                node.left = Node(reservation_time)
                self.increase_sizes(reservation_time)
                return True

            if reservation_time > node.key and node.right is None:
                node.right = Node(reservation_time)
                self.increase_sizes(reservation_time)
                return True

            if reservation_time < node.key:
                node = node.left
            else:
                node = node.right

    def increase_sizes(self, key: int):
        node = self.root
        while node is not None and node.key != key:
            node.size += 1
            if key < node.key:
                node = node.left
            else:
                node = node.right

    def rank(self, time: int) -> int:
        rank = 0
        node = self.root

        while node is not None and node.key != time:
            if time > node.key:
                rank += 1
                if node.left is not None:
                    rank += node.left.size
                node = node.right
            else:
                node = node.left

        if node is not None:
            assert node.key == time,
            rank += 1
            if node.left is not None:
                rank += node.left.size

        return rank
```

Hash-Tabelle mit Chaining

Gibt die Hash-Tabelle der Größe M zurück, die sich aus dem Einfügen von Elementen aus l in dieser Reihenfolge ergibt. Collision Handling mit Chaining.

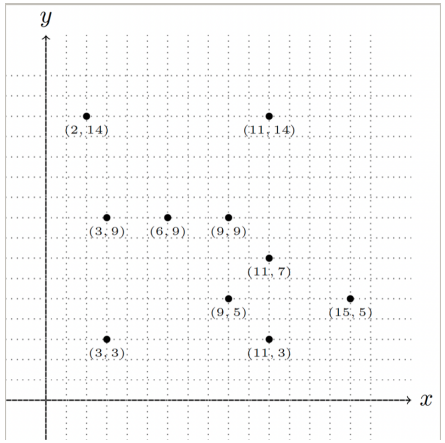
```python
def our_hash(word): # hash-Wert eines Strings
    return ord(word[0])


def create_hash_table(l, M):
    hash_table = [[] for _ in range(M)]

    for x in l:
        index = our_hash(x) % M
        if x not in hash_table[index]:
            hash_table[index].append(x)

    return hash_table
```

Hash-Tabelle mit Probing

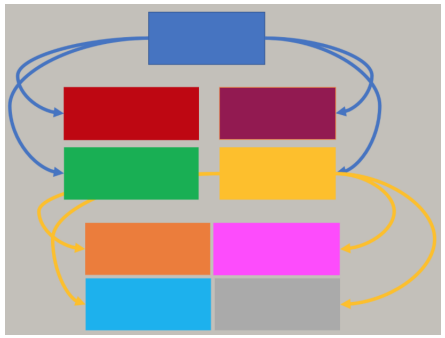
Gibt die Hash-Tabelle der Größe M zurück, die sich aus dem Einfügen von Elementen aus l in dieser Reihenfolge ergibt. Collision Handling mit Probing.

```python
def our_hash_function(word):
    s = 0
    for x in word:
        s += ord(x)
    return s


def create_hash_table(l, M):
    hash_table = [None for _ in range(M)]

    for x in l:
        index = our_hash_function(x) % M
        i = index

        while True:
            if hash_table[i] == x:
                break
            if hash_table[i] is None:
                hash_table[i] = x
                break
            i = (i + 1) % M
            if i == index:
                break

    return hash_table
```

Quadtree



Ein Quadtree, der aus den obigen Punkten erstellt wurde, hat die unten dargestellte Struktur.



Jedes farbige Rechteck bezeichnet ein Objekt vom Typ QuadTree. Geben Sie für jedes Rechteck die Attributwerte für die von diesem Rechteck bezeichneten Objekte an. Angenommen, die maximale Kapazität jedes Quadtrees beträgt hier 3.
Die Farben im Diagramm repräsentieren Knoten im Quadtree:

- Blau ist die Wurzel und umfasst das gesamte Gebiet.
- Rot, Grün, Gelb, Pflaume (Violett) sind die vier Hauptunterteilungen.
- Gelb wird weiter unterteilt, da es mehr als 3 Punkte enthält.

Jedes Quadrat im Quadtree hat eine untere linke Ecke $l = (x_{\min}, y_{\min})$ und eine obere rechte Ecke $u = (x_{\max}, y_{\max})$. Die Wurzel umfasst den gesamten Bereich $(0, 0)$ bis $(16, 16)$, und die Unterteilungen decken Teilbereiche ab.
Blau (Wurzel):

$$l = (0, 0),\ u = (16, 16)\ p = [\,]$$

Kinder: Grün, gelb, pflaume und rot
Rot:

$$l = (0, 8),\ u = (8, 16)\ p = [(2, 14), (3, 9), (6, 9)]$$

Kinder: -
Pflaume (Violett):

$$l = (8, 8),\ u = (16, 16)\ p = [(9, 9), (11, 14)]$$

Kinder: -
Grün:

$$l = (0, 0),\ u = (8, 8),\ p = [(3, 3)]$$

Kinder: -
Gelb:

$$l = (8, 0),\ u = (16, 8)\ p = [\,]$$

Kinder: Orange, Blau, Grau und Pint
Orange:

$$l = (8, 4),\ u = (12, 8)\ p = [(9, 5), (11, 7)]$$

Kinder: -
Blau (unten):

$$l = (12, 4),\ u = (16, 8)\ p = [(15, 5)]$$

Kinder: -
Grau:

$$l = (12, 0),\ u = (16, 4)\ p = [(11, 3)]$$

Kinder: -
Pink:

$$l = (8, 0),\ u = (12, 4)\ p = [(9, 3)]$$

Die Struktur des QuadTrees funktioniert nach folgendem Prinzip:

1. Die Punkte werden eingefügt. Jedes Quadrat kann maximal 3 Punkte enthalten.
2. Sobald ein Quadrat mehr als 3 Punkte enthält, wird es in vier Unterquadrate geteilt.
3. Jede Teilfläche speichert nur die Punkte, die in ihren Bereich fallen.
4. Falls ein Unterquadrat auch die Kapazitätsgrenze von 3 überschreitet, wird es weiter unterteilt.