ZF CompSci II D-MAVT

Julian Lotzer – jlotzer@student.ethz.ch
Daniel Steinhauser – dsteinhauser@student.ethz.ch

Version: 01.09.2023 Extended Version (by lehmannni @ co.)

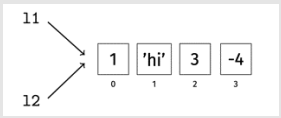
This summary is based on the D-MAVT Computer Science II lectures by Dr. Ralf Sasse and Dr. Carlos Cotrini. The ZF is constantly updated, as the lecture takes place for the first time in this form. No guarantee can be given for correctness or completeness. You will find the newest version on https://n.ethz.ch/~dsteinhauser/ or https://n.ethz.ch/~jlotzer/

1 General Python

1.1 Reference Semantics and Aliasing

Everything is a pointer:

```
l1 = [1, 3, 'hi', -4] #l1 -> [1] [3] ['hi'] [-4]
l2 = l1
```



If a copy is needed, use:

```
import copy
l2 = copy.copy(l2) #shallow copy (1D-Array)
l2 = copy.deepcopy(l2) #deep copy (Multi Dimensional Arrays)
```

1.2 Data Types

Python dynamically types variables, which means that the variable type can change during the program's execution

```
s = 5
print(type(s)) #output: <class 'int'>
s = False
print(type(s)) #output: <class 'bool'>
s = "Hello World"
print(type(s)) #output: <class 'str'>
```

• To convert the data type:

```
s = 5.9 #type(s) = <class 'float'>
x = int(s) #type(x) = <class 'int'>
y = str(s) #type(y) = <class 'str'>
```

1.2.1 Type Hints

• Are not enforced by python, but help make code more legible:

```
def add_integers(li: list[int]) -> int
    …
#the type annotation after the colon (:) indicates the
#expected type. The type annotation after the arrow
#indicates the expected return type
```

1.3 Input and Output

• Output (NOTE print(p.value, **end** = ' ') for no \n)

```
print("Hello World") #output: Hello World
print("Hello", "World") #output: Hello World
print("Hello", "World", sep = "--")
#output: Hello--World
print("Hello", "World", sep = "--", end = "!")
#output: Hello--World!
```

• Input

```
name = input("Enter name: ") #input returns a string
print("Hello", name)
```

• Input of an integer (str->int conversion):

```
number = int(input("Enter your number: "))
print("Number:", name)
```

1.4 Control Flows (if/else, while, for)

• if, elif (else if), else Block

```
x = int(input("Enter a number: "))
if x < 5 and x >= 0:
  print("too small") #if x between 0 and 5
elif x == 69 or x == 420:
  print("nice") #if x is equal to 69 or 420
elif x < 0:
  pass #do nothing if x negative
else:
  print("big number") #x is positive and the ifs/elifs
conditions don't hold
```

• while-loop

```
x = 0
while x <= 3:
  print(x)
  x += 1
```

• for-loop over value ranges (see Ranges Chapter 2.2)

```
for i in range(0,4,1): #range(start, stop, step)
  print(i, end = " ") #output: 0 1 2 3

for i in reversed(range(0,4,1)):
  print(i, end = " ") #output: 3 2 1 0
```

• for-loop over lists

```
l=[3,5,25]
for i in l:
  print(i, end = " ") #output: 3 5 25

for i in reversed(l):
  print(i, end = " ") #output: 25 5 3
```

1.5 Functions

Functions do not have to be declared in a specific order, in contrast to C++ (forward declarations). This means this is completely valid:

```
def foo():
  return bar()

def bar():
  return 0
```

1.5.1 Function Declaration

```
def function(arg1, arg2):
  …
  return value
```

1.5.2 Default arguments

Using default arguments, it is possible to have "optional" arguments:

```
def specialprint(data="hello world")
  print(data)
specialprint() #hello world
```

1.5.3 Global and local variables

Variables that are defined outside of a function are global and can also be used within the function if they are defined before the function call (although this should be avoided whenever possible!).

```
def scaled(x):
  return x*scale #ok, because scale is global

scale = 1.1 #has to be defined before scaled(10)!
print(scaled(10)) #Output: 11
```

• Making a local variable in a function globally accessible:

```
def double(x):
  global result #result is defined globally
  result = x*2

double(7) #function call, before print(result)!
print(result) #Output: 14
```

2 Python Containers

Python containers can be divided into ordered (sequences) and unordered (collections) containers.

Sequences include tuple (all types), list (all types), range (integers) and str (characters)
Collections are e.g.: set (non-associative) and dictionary (associative)

2.1 Operations on Containers

• Number of Elements:

```
len(c)
```

• Contains element x?

```
x in c
```

• Iterate over all elements:

```
for x in c:
 print(x)
```

2.2 Sequences (ordered containers)

• Tuple with 4 Elements:

```
t = ("a", 0, -6, 3.3) #t -> ["a"] [0] [-6] [3.3]
#Tuple with 1 Element: t = ('a',)
#empty tuple: t = ()
```

• List of 4 items:

```
l = [1, 3, "hi", -4]  #l -> [1] [3] ["hi"] [-4]
```

• Range with 4 elements:

```
r = range(0, 8, 2) #r -> [0] [2] [4] [6]
#Syntax:
#range(start, stop, step)
#range(start,stop) -> step = 1
#range(stop) -> start = 0, step = 1
```

• String with length 5:

```
s = "hello" #s ->  ['h'] ['e'] ['l'] ['l'] ['o']
#you can use both " or ' for strings
```

IMPORTANT: only list is mutable; tuple, range and string are immutable!

2.2.1 General sequence operations

• Subscript-Operator `l[i]`:

```
l = [1, 3, 'hi', -4]
print(l[2]) #output: hi
```

• Enumeration:

```
enumerate(iterable, start)
#iterable = iterable container (sequence)
#start (optional) = (optional) enumerate starts counting #at
this number, starts at 0 when omitting start->
#enumerate(iterable)
#the enumerate(iterable, start) function returns a #tuple:
(index, object).
```

• Enumeration example:

```
for index, value in enumerate(l):
 print(index, value)
#output:
# 0 1
# 1 3
# 2 Hi
# 3 -4
```

• Combine sequences s1 and s2 (zip):

```
z = zip(s1,s2)
#example:
#s1 -> ["Lea"] ["Tim"] ["Mortis"]
#s2 -> [22] [19] [69]
#z -> [("Lea",22)] [("Tim",19)] [("Mortis",69)]
```

• Output with a for loop:

```
for name, age in z:
 print(name,"->",age)
# output:
# Lea -> 22
# Tim -> 19
# Mortis -> 69
```

• Slicing (partial sequence) of a sequence s:

```
partseq = s[start:stop:step]
partseq = s[start:stop] #step = 1
partseq = s[:stop:step] #start = 0
partseq = s[start::step] #stop = len(s)
s[::step] # From 0 to len(s) or from len(s)-1 to (and
including!) 0 || if no start -> earlies, no end -> end
```

2.2.2 Common List Operations

• Access item:

```
l[i] = value
```

• Add item at the end:

```
l.append(value)
```

• Remove item at `location i`:

```
del l[i]
```

• Reverse list:

```
l.reverse()
```

• Create a list of k elements with value v:

```
l=[v]*k
```

• To convert a string s to a list of words:

```
s.split(seperator, maxsplit)
#seperator and maxsplit are optional
#s.split() -> split at all whitespaces
#s.split(", ") -> split at every ", "
#s.split(maxsplit = 10) -> split 10 times at the first 10
#whitespaces -> List will have 11 entries
```

2.2.3 List Comprehension

• Apply a `function f(x)` to all items in list `l`:

```
l2 = [f(x) for x in l] #z.g. 2*x for f(x)
```

• Apply a `function f(x)` to a range:

```
r2 = [f(x) for x in range(1,6)]
```

• Apply a `function f(x)` only to items in list `l` that satisfy `g(x)` (filter) ((we could do x for x without a function)):

```
l3 = [f(x) for x in l if g(x)]
```

• Example: Read a sequence of numbers:

```
l = [int(x) for x in input("Input: ").split()]
```

```
O = [[0 for i in range(n)]] * n # BAD, reference!
O = [[0 for i in range(n)] for i in range(n)] # good
```

Unzipping List of Tuples into Two Lists

```
test_list = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# using list comprehension to perform unzipping
res = [[i for i, j in test_list],
       [j for i, j in test_list]]
# print(str(res)): [[1, 2, 3], ['a', 'b', 'c']]
```

2.2.4 Common String Operations

• Access element:

```
s[i]
```

• Add two strings (concatenating):

```
s1 = "hello"
s2 = " world"
s3 = s1 + s2 #s3 = "hello world"
```

• Remove whitespace at beginning and end:

```
s = "   banana "
s=s.strip() #s="banana"
```

• Convert a string into a list of chars:

```
s = list(s)
```

• Example: check if s is a string with content:

```
type(s) == str and len(s.strip()) #False if empty
```

2.3 Collections (unordered containers)

• Set with 3 items:

```
s = {1, 29, 12}
```

• Dictionary with 3 items:

```
d = {"Lea":22, "Tim":19, "Mortis":69} #key:value
```

2.3.1 Set Operations

```
s = {1, 29, 12} #set create
```

• Add item:

```
s.add(69)
```

• Remove item:

```
s.remove(29)
```

• Search for an item:

```
12 in S #returns a bool
```

2.3.2 Dictionary Operations

```
d={"Lea":22, "Tim":19, "Mortis":69} #create dict
```

• Change item:

```
d["Lea"] = 23
```
• Add item:
```
d["Peter"] = Use 24 #an unused key
```
• Delete item:
```
del d["Mortis"] #delete item
```
• Search for a key:
```
"Tim" in d #returns a bool
```
• To access value at a key:
```
d["Tim"] #has value 19
```
• Make two lists into one dictionary:
```
cities = ["Zurich", "Basel", "Bern"] #list 1
zip code = [8000, 4000, 3000] #list 2
d2 = dict(zip(cities,code)) #dictionary D2
```

2.3.3 Iterating over a Dictionary

• Iterate over the keys of a dictionary:
```
for key in d.keys():
  print(key) #Lea Tim Mortis
```
• Iterate over entries of a dictionary:
```
for item in d.items():
  print(item) #("Lea",22) ("Tim",19) ("Mortis", 69)
```
• Iterate over entries, with keys and values separated:
```
for key, value in d.items():
  print(key+" "+value) #Lea 22 Tim 19 Mortis 69
```
• Iterate over the values of the dictionary:
```
for value in d.values():
  print(value) #22 19 69
```

2.3.4 Dictionary/Set Comprehension

• Transform a set into a dictionary by applying `f(x)` and `g(x)` on every element in the set to obtain key and value, respectively:
```
d3 = {f(x):g(x) for x in s} #s being a set
```
• Transform a set into a dictionary, only if the element satisfies `h(x)`:
```
d4 = {f(x):g(x) for x in s if h(x)}
```
• Dictionary comprehension with multiple variables:
```
d5 = {f(x):g(y) for x, y in h(z)}
#h must return a list of tuples, e.g.: zip, d.items. In #the case of d.items, we are applying f(x) on the keys #and g(y) on the values of the dictionary.
```
• Example: Multiply the value of every odd key in a dictionary by 2:
```
d6 = {k:2*v for k, v in d.items() if k % 2 == 1}
# Dict from zipped stuff:
d = {x ** 2 : y **3 for x,y in zip(range(5), [4,3,2,1]) if y > 1} print(d) # {0:64, 1:27, 4:8}
```
• **2D LIST OF DICT**
```
memo = [[{} for _ in range(n+1)] for _ in range(n+1)]
```

3 NumPy

Numpy is a Python package (equivalent to a C++ library) which supports operations with n-dimensional arrays and various computational methods. (fixed size and only one type. True multidimensionality instead of nested).

Import the Numpy package:
```
import numpy as np
```

Now you can refer to functions/classes from Numpy using: "np".

3.1 Numpy Arrays

Numpy arrays are like Python lists. Below is a summary of the key differences.

Lists	Numpy Arrays
Variable size	Fixed size
Different element types	Single element type
Mathematical operations on single elements only	Mathematical operations on whole arrays
Primarily 1D	Multi-dimensional

3.1.1 Declaring Numpy Arrays

• Using sequences (ordered containers):
```
l = [1, 2, 3, 4]
a = np.array(l)
#array([1,2,3,4])
b = np.array(range(2,10,3))
#array([2,5,8])
c = np.array([[1,2],[3,4]])
#array([[1,2],
#       [3,4]])
```
• With random numbers in [0,1)
```
R = np.random.random(10)
#a numpy array with 10 random values between 0 and 1
```
• With random numbers:
```
R = np.random.uniform(-1,1,5)
#a numpy array with 5 random values between -1 and 1
```
• With random integers:
```
R = np.random.randint(1,7,10)
#a numpy array with 10 random values between 1 and 6
```
• Using np.arange (stop is **not inclusive**):
```
R = np.arange(2,10,3) #array([2,5,8])
#the same output as np.array(range(2,10,3))
#np.arange(start, stop, step)
#np.arange(start,stop) -> step = 1
#n.arange(stop) -> start = 0, step = 1
```

• Using linspace (stop is **inclusive**):
```
R = np.linspace(start,stop,num)
#a numpy array with num equally spaced elements
#between start and stop. Stop is inclusive
#Step size = (stop-start)/(num-1)
```

3.1.2 Numpy Array Operations

• Return the number of elements:
```
a = np.arange(10)
a.size #=10
```
• Accessing elements (1D array)
```
a[5] #=5
```
• Accessing elements (2D array):
```
A = np.array([[1,2,3],[4,5,6]])
A[1,2] #=A[1][2]=6
A[:,2] #array([3,6])
A[1,:] #array([4,5,6])
```

3.1.3 Numpy Array Slicing

• Slicing is dependent on the dimensions of the matrix. For 1D arrays:
```
A = np.arange(10) #[0,1,2,3,4,5,6,7,8,9]
A[2:5:2] #array([2,4])
```
• For 2D arrays (matrices):
```
A = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
A[0:2,1:3] #array([[2,3],
                  [5,6]])
```

3.1.4 Numpy Array Statistics

```
a = np.linspace(-4,-2,3) #array([-4,-3,-2])
```
• Minimum of all elements:
```
a.min() #-4
```
• Maximum of all elements:
```
a.max() #-2
```
• Sum of all elements:
```
a.sum() #-9
```
• Average of all elements:
```
np.mean(a) #-3
```
• Standard deviation of all elements:
```
np.std(a) #0.81
```

3.1.5 Mathematical Operations on Numpy Arrays

• Generally mathematical operations are carried out element-wise:
```
A = np.array([[2,3,4],[6,7,6]])
B = np.array([[1,9,1],[2,3,9]])
A+1
#array([[3,4,5],
#       [7,8,7]])
A * 2
#array([[4,6,8],
#       [12,14,12]])
A ** 4
#array([[16,81,256],
#       [1296,2401,1296]])
np.sin(A)
#array([[0.909,0.141,-0.756],
#       [-0.279,0.657,-0.279]])
A + B
#array([[3,12,5],
#       [8,10,15]])
A * B
#array([[2,27,4],
#       [12.21,54]])
np.sum(A, axis = 0)
#= A.sum(axis = 0) -> array([8,10,10])
np.sum(A, axis = 1)
#= A.sum(axis = 1) -> array([9,19])
```

3.1.6 Matrix Operations on Numpy Arrays

• Given that the dimensionality of two matrices is correct, one is able to multiply them using @:
```
A = np.array([[2,3,4],[6,7,6]])
A @ np.array([[1, 4], [3, 4], [4,6]])
#array([[27, 44],
#       [51, 88]])
```
• Using the dot() function to multiply two matrices:
```
A.dot(np.array([[1,4],[3,4],[4,6]])
#array([[27, 44],
#       [51, 88]])
```
• Using the dot() function to find the scalar product of two vectors:
```
a = np.array([1,2,3])
b = np.array([3,4,6])
a.dot(b) #29
```

3.1.7 Filtering Numpy Arrays

• Filter a numpy array using the subscript operator:
```
a = np.arange(7) #array([0,1,2,3,4,5,6])
f = a % 2 == 0
a[f] #array([0,2,4,6])
```

4 Pandas

Pandas is a Python package which supports working with tabulated data. It describes itself as an open-source data analysis and manipulation tool.

Having pandas installed, one can use:
```
import pandas as pd
```

4.1 Using Pandas to read a CSV File

• To read a CSV file stored in the same directory as your code, use:
```
climate = pd.read_csv("climate.csv", sep=",", index_col=0, usecols=["time", …])
#"sep" -> what characters values in the csv #file are #separated by. "index_col" -> what the index column will #be. "usecols" -> what columns of the csv data will be #selected.
```

4.2 Pandas Dataframe

A dataframe can be thought of as a 2D list (list within a list) supporting access in more semantic, meaningful ways compared to using indices. Visualized, a dataframe may look something like the following:

	Unnamed: 0	time	jan	feb	mar	apr	may	jun
0	0	1864	-7.10	-4.52	0.04	2.11	7.43	9.48
1	1	1865	-3.47	-6.25	-5.91	7.03	10.09	10.98
2	2	1866	-1.31	-0.42	-1.00	4.11	4.95	12.02
3	3	1867	-3.87	0.56	-0.13	3.49	7.74	10.57
4	4	1868	-5.46	-1.53	-2.30	2.33	12.04	11.97
…	…	…	…	…	…	…	…	…
153	153	2017	-5.15	0.46	4.11	4.42	9.80	15.18
154	154	2018	0.48	-5.21	-0.21	7.81	10.43	13.81
155	155	2019	-4.37	0.73	2.27	4.47	6.08	15.25
156	156	2020	-0.28	1.62	1.53	7.62	9.53	11.82
157	157	2021	-3.56	NaN	NaN	NaN	NaN	NaN

climate

The leftmost column is known as the "index column".

4.2.1 Changing the Index Column

• Change the index column by using (creates a copy):
```
climate2 = climate.set_index("time")
```

time	Unnamed: 0	jan	feb	mar	apr	may	jun
1864	0	-7.10	-4.52	0.04	2.11	7.43	9.48
1865	1	-3.47	-6.25	-5.91	7.03	10.09	10.98
1866	2	-1.31	-0.42	-1.00	4.11	4.95	12.02
1867	3	-3.87	0.56	-0.13	3.49	7.74	10.57
1868	4	-5.46	-1.53	-2.30	2.33	12.04	11.97
…	…	…	…	…	…	…	…
2017	153	-5.15	0.46	4.11	4.42	9.80	15.18
2018	154	0.48	-5.21	-0.21	7.81	10.43	13.81
2019	155	-4.37	0.73	2.27	4.47	6.08	15.25
2020	156	-0.28	1.62	1.53	7.62	9.53	11.82
2021	157	-3.56	NaN	NaN	NaN	NaN	NaN

climate

4.2.2 Renaming Columns

• Using the "rename" function:
```
climate = climate.rename(columns={"time":"date", …}
#renames the time column as "date". Add any entries in #the form of: "old_index_name":"new_index_name"
```
• Directly set column names:
```
data.columns = ["Date", "January", "February", …]
#needs to be the same length as the number of columns
```

4.2.3 Accessing Dataframe Elements

• Access a single column (type: Series):
```
climate["feb"]
#gets the column "feb"
```
• Access multiple columns (type: Dataframe):
```
climate[["jan", "mar"]]
#gets the columns "jan" and "mar"
```
• Access a single row, using an index (type: Series):
```
climate.iloc[3]
#gets row 3
```
• Multiple rows (type: Dataframe):
```
climate[1:4]
#gets rows 1 to 3
```
• Access to a subtable using indices (type: Dataframe):
```
climate.iloc[4:7,1:2]
#gets rows 4,7 with data only from column 1
```
• Access to a subtable using index column values and column name (type: Dataframe):
```
climate2 = climate.set_index("time")
climate2.loc[1864:1868,"jan":"mar"]
#includes the rows labeled with 1866 until and including #1868, the columns from "jan" until and including "mar"
```
• Access to a single element:
```
climate["jan"][3]
#gets the element in column "jan" in row 3
```

4.2.4 Filtering Dataframes

• Filter rows:
```
climate[climate["jan"]>2]
#filters out the rows with values in the "jan" column #less than 2
```
• Example: All entries in "jan" with values more than 2:
```
climate["jan"][climate["jan"]>2]
```

4.2.5 Dealing with Invalid Data

• Convert all the values in a column to numeric:
```
data[column] = pd.to_numeric(data[column], errors="coerce")
#converts all the values to numeric values. #errors="coerce" -> converts values which cannot be #converted to NaN.
```
• Delete all rows containing NaN entries:
```
data.dropna(axis = 0, how="any")
#how="any" -> delete row if any value is NaN.
#how="all" -> delete row if all values are NaN
#axis = 1 -> delete column instead of row
```
• Fill all entries containing NaN with a value:
```
data.fillna(0) #fill any NaN entries with 0
```

4.2.6 Modifying Dataframes

• Add a column:
```
climate["new_col"] = climate["time"] + climate["jan"]
#"new_col" is a new column who's values are #those of the
"time" and "jan" column added
```
• Delete a column:
```
climate = climate.drop(columns=["time"])
#delete the "time" column
```
• Add a row:
```
d = {"mar":34, "jan":23}
climate.append(d, ignore_index=True)
#adds another row with the values 34 for "mar" and 23 for
#"jan". Other entries are NaN
```
• Delete a row:
```
climate = climate.drop(climate.index[0])
#deletes row 0
```
• Transpose the dataframe:
```
climate = climate.T
```

4.2.7 Analysing Data in Dataframes

• Sum of all the entries in each column (type: Series):
```
climate.sum()
```
• Maximum of all the entries in each column (type: Series):
```
climate.max()
```
• Create a dataframe summarizing the max and sum for each column:
```
climate.agg(["max","sum"])
#A dataframe containing the same columns as climate
#with row 0 containing the max of the column and row 1
#containing the sum of the column. The strings in the
#list should be names of valid pandas Series functions.
```
• Get statistical information for each column (type: Dataframe):
```
climate.describe()
#includes a variety of statistical measures
```
• Sort a dataframe according to entries in a specific column(s):
```
climate = climate.sort_values(["time", "jan"],
ascending=False)
#sorts the rows by "time" in descending order. If two
#entries for "time" are equal, then the rows are sorted #by
"jan"
```
• Split a dataframe into groups based on a specified column and perform a computation on each group:
```
data.groupby("column").sum()
#groups data based on the entries for "column" and
#calculates the sum for each group.
data.groupby("column").max()
#groups data based on the entries for "column" and
#calculates the max for each group.
```

5 Matplotlib

Matplotlib is a Python package allowing you to visualize a variety of things: from functions to animations. To import matplotlib, use:
```
import matplotlib.pyplot as plt
```
(not relevant for exam so far)
Now you can refer to classes and functions from the package using "plt".

5.1 Line Plots

• To graph two numpy arrays, one representing the x values and the other representing the corresponding y values:
```
import matplotlib.pyplot as plt
import numpy as np

X = np.linspace(0,2*np.pi,100)
Y = np.sin(X)

fig, ax = plt.subplots()
ax.plot(X,Y)
```

5.2 Scatter Plots

• To graph two numpy arrays, one representing the x values and the other representing the corresponding y values:
```
X = np.arange(1,9)
Y = np.array([1,2,3,1,2,3,1,2])
fig, ax = plt.subplots()
ax.scatter(X,Y)
```

5.3 Histogram Plots

• To plot a histogram, use the following template:
```
fig, ax = plt.subplots()
X = np.random.randint(0, 100, 500)
# low: 0, high: 100, size: 500
ax.hist(X, bins=10)
plt.show()
```

5.4 Graph Styling

```
fig, ax = plt.subplots()
```

• To add a title, use:
```
ax.set_title("title")
```
• To set the x-label:
```
ax.set_xlabel("x label name")
```
• To set the y-label:
```
ax.set_ylabel("y label name")
```
• To add a legend:
```
ax.legend()
#requires that you labeled your plots, i.e.: when calling
#ax.plot(X,Y, label="name of function").
```

6 Algorithms

Algorithms are set a of instructions to solve specific problems. The most common problems we look at in computer science involve sorting and searching for elements in data.

6.1 Measuring Performance

6.1.1 Big O Notation

To measure the performance of an algorithm, we use big O notation.
Let g be the relationship time vs input size for an algorithm.

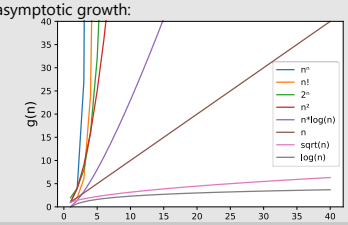
• If g does not grow faster than c*f:
$g = O(f)$ # upper limit / worst case
• If g grows about the same as c*f:
$g = \Theta(f)$ #g~c*f, where c is a constant
• If g does not grow slower than c*f:
$g = \Omega(f)$ # best case

Mathematical definitions:

$$O(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$
$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq \frac{1}{c} \cdot g(n) \leq f(n) \leq c \cdot g(n)\}$$
$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

6.1.2 Asymptotic Growth of Functions

• Functions in **increasing** asymptotic growth:
1. $\log(n)$
2. $\sqrt{n}$
3. n
4. $n \cdot \log(n)$
5. n^2
6. 2^n
7. $n!$
8. n^n



6.1.3 Code Runtime Analysis

To be able to analyze the time complexity of code, we must make certain assumptions:

• Comparisons have a 'time cost' of 1:
```
if x==1 #has a cost of 1
if x>1 #has a cost of 1
```
• Mathematical operations have a cost of 1:
```
6 + 4 #has a cost of 1
```
• Assignment has a cost of 1:
```
x = 69 #has a cost of 1
```
• In general, operations on fundamental types have a cost of 1.

• Example of runtime analysis (selection sort):
(almost always f() in questions O(1), not guaranteed!)
```
def sort(a):
  n = len(a)
  for i in range(n):
    mini = i
    for j in range(i+1,n):
      if a[j] < a[mini]:
        mini = j
    a[mini], a[i] = a[i], a[mini]
```

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

```
# Time complexity patterns

for(i = 0; i < n; i++)-> O(n)
for(i = 0; i < n; i = i + 2)-> n/2 O(n)
for(i = n; i > 1; i--)-> O(n)
for(i = 1; i < n; i = i * 2)-> O(log_2(n))
for(i = 1; i < n; i = i * 3)-> O(log_3(n))
for(i = n; i > 1; i = i / 2)-> O(log_2(n))

for(i = 1; p <= n; i++){-> k > sqrt(n)
p = p + 1
}

# remember 1+2+3+4+...+k -> k(k+1)/2
```

Examples:

$$\sqrt{\log(n)} < \log(\sqrt{n}) < n\log(n^2) < n\log(n^2) < \sum_{i=0}^{n} n - i < n^2\log(n) < n^3 - n^{-3}$$

$$\frac{1}{n} < \log\left(\sum_{i=1}^{n} i\right) < \sqrt{n}\log(n) < n\log(\sqrt{n}) < \sum_{i=1}^{n} i < n!$$

remember: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

```
# θ(sqrt(n))
def g(n):
m = 0;
while m*m < n:
  f()
m=m+1
```

```
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        count += 1
#n/2^c >= 1 | n >= 2^c | log_2(n) >= c
```

6.1.4 Useful Formulas

$\sum_{i=0}^{n-1} 1 = n$
$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$
$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1)(2n+1)}{6}$
$\sum_{i=0}^{\log(n)} 2^i = n(2^{(\log_2(n)+1)} - 1)$
$\sum_{i=0}^{n-1} i^a = \theta(n^{a+1})$
$\log(n^b) = b \cdot \log(n)$
$n \sum_{i=0}^{\log n} 2^i = n\left(2^{(\log_2 n+1)} - 1\right)$

6.1.5 Telescoping & Master Theorem

Master Theorem **Telescoping**

$$T(n) = \underbrace{a}_{\text{recursive call(s)}} \cdot T\left(\frac{n}{b}\right) + \underbrace{\theta(n^k \cdot log^p(n))}_{\text{additional runtime in each call of } T(n), \text{ (example: for-loop)}}$$

Tipp: determine $\theta()$ directly and do Koeffizientenvergleich with the Formula above

1. if $a > b^k$, then $T(n) = \theta(n^{log_b(a)})$
2. if $a = b^k$, and
   a) if $p > -1$, then $T(n) = \theta(n^{log_b(a)} \cdot log^{p+1}(n))$
   b) if $p = -1$, then $T(n) = \theta(n^{log_b(a)} \cdot log(log(n)))$
   c) if $p < -1$, then $T(n) = \theta(n^{log_b(a)})$
3. if $a < b^k$, and
   a) if $p >= 0$, then $T(n) = \theta(n^k \cdot log^p(n))$
   b) if $p < 0$, then $T(n) = \theta(n^k)$

n = size of input **a**= number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size. **f(n)** = $\theta(n^k \log^p n)$ cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions (e.g. a linear operation d = 1) O^d

$$T(n) = \begin{cases} d, & n = 1 \\ T\left(\frac{n}{2}\right) + c, & n > 1 \end{cases}$$
$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c$$
$$= T\left(\frac{n}{2^i}\right) + i \cdot c$$
$$= T\left(\frac{n}{n}\right) + \log_2 n \cdot c$$
$$= d + \log_2 n \cdot c = \Theta(\log(n))$$

6.2 Sorting Algorithms

6.2.1 Invariants

When it comes to algorithms that involve loops, there is usually a **condition** called the **invariant**. This invariant fulfils:
1. Initialization: the condition is met before the loop.
2. Continuation: the condition holds at each iteration of the loop.
3. Termination: the condition holds at the end of the loop.

An algorithm involving a loop and having an invariant is said to be correct if the invariant fulfils the above.

6.2.2 Divide and Conquer

Divide and conquer is type of algorithm involving the process of recursively splitting the problem into two or more equally sized subproblems of the same type.

Examples: **mergesort, quicksort.**

6.2.3 Selection Sort ✅

Selection sort iterates through the list, repeatedly finds the smallest element, and swaps it to its final position (current position in iter.). code:

```
def sort(a):
  n = len(a)
  for i in range(n):
    # find minimum
    mini = i
    for j in range(i+1, n):
      if a[j] < a[mini]:
        mini = j
    # swap minimum
    a[mini],a[i] = a[i],a[mini]
```

Case	Description	Runtime
Worst-case	A is reverse sorted.	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(n)$

Comparisons: Worst, Best & Average: $\theta(n^2)$
Swaps: Worst & Average: $\theta(n)$, Best: $\theta(1)$

6.2.4 Insertion Sort

Sort an array of values by taking the next value in the array and putting it in the right position. code:

```
def insertion_sort(items):
    for i in range(1, len(items)):
        while i > 0 and items[i] < items[i-1]:
            items[i], items[i-1] = items[i-1], items[i]
            i -= 1
    return items
```

Case	Description	Runtime
Worst-case	A is reverse sorted.	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(n)$

Comparisons: Worst & Average: $\theta(n^2)$, Best: $\theta(n)$
Swaps: Worst & Average: $\theta(n^2)$, Best: $\theta(1)$

6.2.5 Merge Sort

Sort an array by splitting it into smaller subarrays (divide and conquer) and rearranging them to form a sorted array. code:

```
# pre: list A of comparable elements,
# post: list A sorted between l (included) and r (not included)
def mergeSort(A, l, r):
  if r-l>1:
    m = (l + r) // 2
    mergeSort(A,l,m)
    mergeSort(A, m, r)
    merge(A,l,m,r)
```

```
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)
```

Uses additional $\Theta(n)$ storage for subarrays.

Case	Description	Runtime

All cases	-	$\Theta(n \cdot \log(n))$

6.2.6 Quick Sort

A divide and conquer algorithm which recursively splits the array into two parts: one which only contains elements bigger than the pivot and the other containing only elements smaller. code:

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

```
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 i
```

Case	Description	Runtime
Worst-case	Pivot is the min/max value.	$\Theta(n^2)$
Average-case	Pivot is chosen randomly.	$\Theta(n \cdot \log(n))$
Best-case	Pivot is always the median of the array.	$\Theta(n \cdot \log(n))$

Pivot is often the median of three elements:
`Pivot = Median3(A[l],A[(l+r)//2],A[r])`

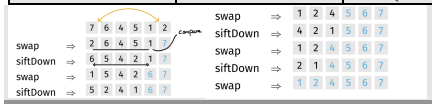
6.2.7 Heapsort

A sorting algorithm which converts an array into a heap (see 7.2) and creates a sorted array from the heap. Pseudocode:

```
def heapify(arr, n, i):  # SiftDown operation
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:       # MaxHeap: '<' '<' MinHeap: '<' '>'
        largest = l
    if r < n and arr[largest] < arr[r]: # MaxHeap: '<' '<' MinHeap: '<' '>'
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)
    # Build max heap
    for i in range(n//2, -1, -1): # Create heap
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        # Swap
        arr[i], arr[0] = arr[0], arr[i]
        # Heapify root element
        heapify(arr, i, 0)

arr = [1, 12, 9, 5, 6, 10,0,999,213,1,22]
heapSort(arr)
```

```
# after we got max heap, swap smallest with biggest, draw tree so you can sift down the element so the largest is on top again.
```

Case	Description	Runtime
All	-	$\Theta(n \cdot \log(n))$



6.3 Searching Algorithms

6.3.1 Linear Search

Search for the index of a specific element in an **unsorted** array. Code:

```
def LinearSearch(array, n):
    for i, x in enumerate(array):
        if x == n:
            return i
    return "not found"
```

Case	Description	Runtime
Worst-case	b is at the end of the array.	$\Theta(n)$
Average-case	-	$\Theta(n)$
Best-case	b is at the beginning.	$\Theta(1)$

6.3.2 Binary Search

Search for the index of a specific element in a **sorted** array. Code:

```
def binarySearch(array, x, low, high):
    if high >= low:
        mid = low + (high - low)//2
        if array[mid] == x:     # If found at mid, return it
            return mid
        elif array[mid] > x:    # Search left half
            return binarySearch(array, x, low, mid-1)
        else:                   # Search right half
            return binarySearch(array, x, mid + 1, high)
    else:
        return -1

array = [3, 4, 5, 6, 7, 8, 9]
x = 4
result = binarySearch(array, x, 0, len(array)-1)
```

Case	Description	Runtime
Worst-case	b is the max/min value.	$\Theta(\log(n))$
Average-case	-	$\Theta(\log(n))$
Best-case	b is exactly the median value.	$\Theta(1)$

7 Data Structures

Data structures are ways of organizing and storing data for efficient access and manipulation.

A note on BSTs: a binary tree is a tree with at most 2 children. Binary trees can be:

- Full: every node has 0 or 2 children.
- Complete: every level except for the lowest is filled. Lowest level is filled from left to right.
- Perfect: every level is completely filled.

7.1 FIFO/LIFO (Stack)

Some data structures can be categorized as FIFO (first in, first out) or LIFO (last in, first out).

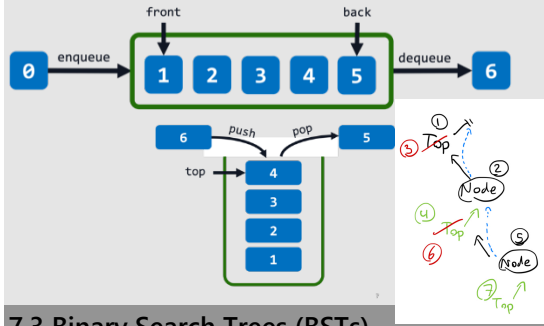
FIFO data structures offer fast access to the first element that was inserted.

LIFO data structures offer fast access to the last element that was inserted.

FIFO example: queue

7.2

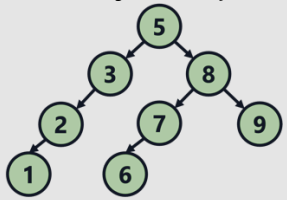
LIFO example: stack



7.3 Binary Search Trees (BSTs)

A binary search tree is a binary tree which fulfils the following:

1. Every node v stores a key
2. Keys in the left subtree are smaller than `v.key`
3. Keys in the right subtree are larger than `v.key`



A binary search tree

7.3.1 Height of a BST

The height of a BST is defined as the maximum depth one must recurse to reach a node with no children. Code:

```
def height(node: Optional[Node]) -> int:
    if node is None:
        return 0
    left_height = height(node.left)
    right_height = height(node.right)
    return max(left_height, right_height) + 1
```

7.3.2 Search for a Node

Return the node with a specific key.

following code is based upon the class implementation

```
def search(self, key: int) -> Optional[Node]:
  """Search for the node with the specified key in the tree and return
  it, or None if there is no such node.
  """
  node = self.root
  while node != None:
    if key == node.key:
      return node
    elif key < node.key:
      node = node.left
    else:
      node = node.right
  return None
```

Case	Description	Runtime
Worst-case	The tree is degenerated	$\Theta(n)$
Average-case	The tree is balanced	$\Theta(\log(n))$
Best-case	Key is the root of the tree	$\Theta(1)$

7.3.3 Insert a Node

Add a node with a specific key to the tree.

```
def add(self, key: int) -> bool:
  """Add a node with the specified key to the tree if it does not
  already exist in the tree.

  Return True iff the node was added, and False otherwise.
  """
  if self.root is None:
    self.root = Node(key)
    return True

  node = self.root
  while True:
    # Key already exists in the tree
    if key == node.key:
      return False

    # Found the position with an empty child
    if key < node.key and node.left is None:
      node.left = Node(key)
      return True

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

    # Otherwise, keep searching for the node position
    if key < node.key:
      node = node.left
    else:
      node = node.right
```

Case	Description	Runtime
Worst-case	The tree is degenerated	$\Theta(n)$
Average-case	The tree is balanced	$\Theta(\log(n))$
Best-case	The tree is empty	$\Theta(1)$

7.3.4 Remove a Node

1. If a node has no children, simply delete the node by setting the variable to `None`
2. If a node has one child, replace the node with its child.
3. If a node has two children, replace the node with its ***symmetric successor*** – the next biggest element. Code:

```
def remove(self, key: int) -> bool:
  """Remove the node with the specified key from the tree.  Return
  True if the node was removed, False if it was not in the tree.
  """
  # Handle the 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:
    # Found the 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
```

```
def findSuccessor(start_node: Node) -> Optional[Node]:
    # If the node has a right child, the successor is the leftmost node in the right subtree
    if start_node.right:
        node = start_node.right
        while node.left:
            node = node.left
        return node
    return None  # This is a simplified version.
    # In a full implementation, you'd traverse up the tree to find the successor.
```

Case	Description	Runtime
All	Runtime is dominated by finding the successor.	$\mathcal{O}(h)$ h is the height of the tree

7.3.5 Traversal

There are different ways to traverse over a BST:

1. Inorder traversal (**will always print elements in ascending order**):

```
def inorder_traverse(node: Optional[Node]):
    if node:
        inorder_traverse(node.left)
        print(node.key)
        inorder_traverse(node.right)
```

2. Preorder traversal:

```
def to_preorder_list(node: Optional[Node]) -> list:
  if node is None:
    return []
  return [node.key] + to_preorder_list(node.left) + to_preorder_list(node.right)
```

3. Postorder traversal:

```
def postorder_traverse(node: Optional[Node]):
    if node:
        postorder_traverse(node.left)
        postorder_traverse(node.right)
        print(node.key)
```

Case	Description	Runtime
All	Each edge is traversed twice and the number of edges is $\Theta(n)$	$\Theta(n)$

7.3.6 Traversal Rules

- Given an inorder traversal: there is no possible representation of the tree if the sequence is not in ascending order. **Representation is not unique.**

Example: 1 2 4 3 has no representation as a BST.

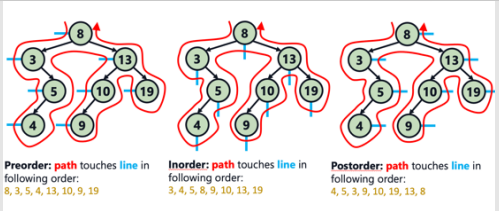
- Given a preorder traversal: there is no possible representation of the tree if there is not a way to (recursively for the new subsequences as well) place the first number in the sequence so that the numbers to the left are smaller and that the numbers to the right are greater. **Representation is unique.**

Example: 4 3 1 2 8 6 5 7
3 1 2 4 8 6 5 7
1 2 3, 6 5 7 8
1 2, 5 6 7 -> valid

- Given a postorder traversal: there is no possible representation of the tree if there is not a way to (recursively for the new subsequences as well) place the last number in the sequence so that the numbers to the left are smaller and the numbers to the right are greater. **Representation is unique.**

Example: 1 3 2 5 6 8 7 4
1 3 2 4 5 6 8 7
1 2 3, 5 6 7 8
5 6 -> valid

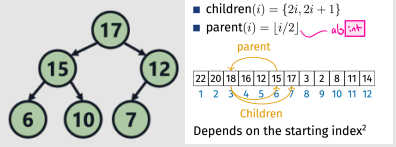
7.3.7 Tree Traversal - Trick



7.4 (Max) Heaps

A heap is a binary tree which fulfils the following:
1. It is complete (see 7).
2. If there are any gaps in the tree, they're on the last level to the right (definition of complete, see 7).
3. Key of parent is always bigger than the one of its children.

It is a good data structure to use when one wants efficient access to the max/min at all times.



A max heap / Heap as an Array

7.4.1 Implementation of a Heap as Array

Given an element with index i, with first index = 0:
- Children of i: `{2i+1, 2i+2}`
- Parent of i: `i-1//2`

With first index = 1:
- Children of i: `{2i, 2i+1}`
- Parent of i: `i//2`

7.4.2 Height of a Heap

- Given n elements:

$$H(n) = \lceil \log_2(n+1) \rceil$$

7.4.3 Insert an Element

Insert an element by placing it in the first free place on the lowest level of the heap. Iteratively swap the element with its parent until the heap conditions are fulfilled (Chapter 7.2).
The Code snippets of 7.4 will work by themselves (independent of 6.2.7)

```
def insert(heap, value):  # heap: list representing the heap
    heap.append(value)
    SiftUp(heap, len(heap)-1)
```

```
def SiftUp(a, m):   # a: list representing the heap,
                    # m: index of the element to be sifted up
    v = a[m]        # v: value to be sifted up
    c = m           # c: current index of the value
    p = c//2        # p: parent index of the current value
    while c > 0 and v > a[p]: #! For MinHeap: change '>' '>' to '>' '<'
        a[c] = a[p]
        c = p
        p = c//2
    a[c] = v
```

Case	Description	Runtime
All	In the worst case, inserting an element will involve $\log(n)$ swaps.	$\mathcal{O}(\log(n))$

7.4.4 Remove the Maximum (Max-Heap)

Replace the maximum with the rightmost element in the lowest level and iteratively swap the replacement element in direction of the greater child (smaller child for min-heaps)

```
def removeMax(heap):    # heap: list representing the heap.
                        #! (For MinHeap: rename to removeMin)
    if len(heap) == 0:
        return None
    max_val = heap[0]   #! For MinHeap: rename max_val to min_val
    heap[0] = heap[-1]
    heap.pop()
    SiftDown(heap, 0, len(heap)-1)
    return max_val      #! For MinHeap: return min_val
```

```
def SiftDown(a, i, m):  # a: list representing the heap, i: start index, m: end index
    while 2*i + 1 < m:
        j = 2*i + 1     # j: left child index
        if j + 1 < m and a[j] < a[j + 1]:  #! For MinHeap: change '<' '<' to '<' '>'
            j = j + 1   # j: right child index if right child exists and is greater than left child
        if a[i] >= a[j]:                   #! For MinHeap: change '>=' to '<='
            break
        a[i], a[j] = a[j], a[i]
        i = j
```

Case	Description	Runtime
All	In the worst case, removing an element will involve $\log(n)$ swaps.	$\mathcal{O}(\log(n))$

7.4.5 Heapify an Array

Property: the leaves of a heap fulfill the heap condition trivially -> only need to "heapify" the first n/2 elements.

```
def heapify(a):  # a: list to be transformed into a heap
    n = len(a)
    for i in range(n//2 - 1, -1, -1):
        SiftDown(a, i, n)
```

`def SiftDown(a, i, m): REQUIRED! see 7.4.4`

Case	Description	Runtime
All	-	$\Theta(n)$

7.4.6 Sorting a heap

If "a" is a heap, one can efficiently sort the array:

```
def SortHeap(a):  # a: list representing the heap
    n = len(a)-1
    while n > 0:
        a[0], a[n] = a[n], a[0]
        SiftDown(a, 0, n-1)
        n = n - 1
```

Case	Description	Runtime
All	SiftDown traverses at most $\log(n)$ nodes. Sorting the array requires n calls to SiftDown.	$\Theta(n \cdot \log(n))$

```
Testing
heap = []
insert(heap, 5)
insert(heap, 3)
insert(heap, 8)
insert(heap, 1)
print("Heap after insertions:", heap)
print("Removed max:", removeMax(heap))
print("Heap after removeMax:", heap)
```

7.5 Vectors/Lists

- Ordered data.
- Fast Access via index.
- Slow for updates.

Operation	Time Complexity
Index Access	$\mathcal{O}(1)$
Search (`in`)	$\mathcal{O}(n)$
Sorted Search	$\mathcal{O}(\log(n))$
Insertion	$\mathcal{O}(n)$
Removal	$\mathcal{O}(n)$

7.6 Linked Lists

- Ordered data.
- Fast updates at the front of the linked list.

Operation	Time Complexity
Access	$\mathcal{O}(n)$
Search (`in`)	$\mathcal{O}(n)$
Insertion (head)	$\mathcal{O}(1)$
Removal (head)	$\mathcal{O}(1)$
Insertion (after value)	$\mathcal{O}(n)$
Insertion (after value)	$\mathcal{O}(n)$

7.7 Hash Tables

Hash tables are a data structure which allow fast access to elements.
- Unsorted, unordered data.
- Fast search.

The idea behind the implementation is that one uses a "*hashing function*" to obtain an index/address from the element, and to store the element there.

Operation	Best Case Complexity	Worst Case Complexity
Search	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Removal	$\mathcal{O}(1)$	$\mathcal{O}(n)$

7.7.1 Collision Handling

With probing: next available index is chosen.
- Problem: entry at calculated index may not contain element.

With chaining: a linked list at every entry.

7.7.2 Properties of a Hash Function

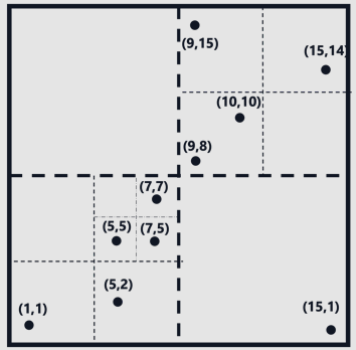
- Consistent (always same output for given input).
- As collision-free as possible.

7.7.3 Hash Table manual

There most likely will be one or two hash functions given, if after the first one we have a collision we go to the next one and add from where we got the collisions the amount of steps we've newly calculated

7.8 Quadtrees

A quadtree is a type of tree with 4 children. In this course, the application of a quadtree is mainly graphical.



A QuadTree with max_cap =2

Example of a quadtree implementation:

```
class QuadTree:
  def __init__(self, l, u, max_cap):
    self.l = l
    self.u = u
    self.m = max_cap
    self.points = []
    self.children = None
    self.count = 0
```

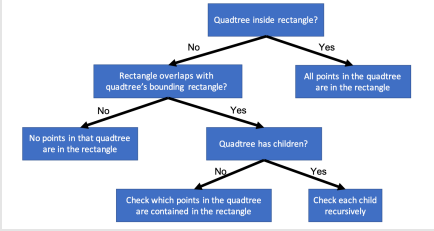
```python
def subdivide(self):
    lx, ly = self.l
    ux, uy = self.u
    mx, my = (lx + ux) / 2, (ly + uy) / 2
    self.children = [None, None, None, None]
    self.children[0] = QuadTree((lx, ly),
                                (mx, my),
                                self.m)
    self.children[1] = QuadTree((mx, ly),
                                (ux, my),
                                self.m)
    self.children[2] = QuadTree((mx, my),
                                (ux, uy),
                                self.m)
    self.children[3] = QuadTree((lx, my),
                                (mx, uy),
                                self.m)
```

```python
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.points = []
```

```python
def get_index(self, point):
    lx, ly = self.l
    ux, uy = self.u
    mx, my = (lx + ux) / 2, (ly + uy) / 2
    px, py = point
    if lx <= px < mx:
        if ly <= py < my:
            return 0
        elif my <= py <= uy:
            return 3
    elif mx <= px <= ux:
        if ly <= py < my:
            return 1
        elif my <= py <= uy:
            return 2
    return None
```

```python
# Initialize a QuadTree with boundary (0,0) to (10,10) and max capacity of 4
qt = QuadTree((0, 0), (10, 10), 4)
# Insert points
points = [(2, 3), (4, 5), (7, 8), (9, 1), (6, 6), (3, 7), (8, 9), (5, 2)]
for point in points:
    qt.insert(point)
```

7.8.1 Search for Number of Points within Rectangle



Time complexity: $\mathcal{O}(\log(n))$

8 Programming Concepts

8.1 Classes and Objects (OOP)

Classes:
- Bundling of data that belongs together contentwise.
- Definition of a new type.

Data:
- Stored in variables of the class (attributes).
- Default values can be declared in the class (see 1.5.2).

Object:
- Instance of a class

8.1.1 Example Implementation

```python
class Earthquake:
    #Constructor (member variables: location, magnitude)
    def __init__(self, location, magnitude):
        self.loc = location
        self.mag = magnitude
        self.__id = 5 #hidden (private) attribute

    #Print operator overload -> what is shown when
    #print(Earthquake) is called
    def __str__(self):
        return "earthquake with mag: " + str(self.mag)

    #Overloaded ==
    def __eq__(self,other):
        return self.mag == other.mag and self.loc == other.loc

    #method to access hidden/private attribute
    def get_id(self):
        return self.__id

    # A class method
    def bar(cls, x):
        print("This is a class method with argument:", x)

a = Earthquake("Indonesia", 5)
print(a) #earthquake with mag: 5
a.bar(5) #call class method bar
print(a.__id) #AttributeError
```

8.1.2 Magical Methods

Magical methods allow you to overload operators in python. Below is a table of magical methods you can define.

Comparisons:

Operation	Meaning	Magical Method
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__
==	Equal to	__eq__
!=	Not equal to	__ne__

Relational Operations:

Operation	Meaning	Magical Method
+, +=	Addition	__add__, __iadd__
-	Subtraction	__sub__
*	Multiplication	__mul__
/	Division	__truediv__
//	Integer division	__floordiv__
%	Modulo	__modulo__
**	Exponentiation	__pow__

Others:

Operation	Meaning	Magical Method
print()	overload print()	__str__
-	Negation	__neg__
Classname()	Constructor	__init__

```python
class Student:
    def __init__(self, name):
        self.name= name
    def __str__(self): #overload print()
        return self.name
```

8.1.3 Inheritance

Through inheritance a class can "inherit" all the attributes and the methods of the class it is inheriting from.

Example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("The animal makes a sound.")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
        print("The dog barks.")

#Create an instance of the Dog class
dog = Dog("Fido", "Golden Retriever")

#Access the attributes of the Dog instance
print(dog.name)   #Output: Fido
print(dog.breed)  #Output: Golden Retriever

#Call the speak method on the Dog instance
dog.speak()  #Output: The dog barks.
```

8.2 Compiled vs Interpreted

Compiled (C++):
- Program code is translated to assembly.
- Assembly is executed.
- Single translation, with optimizations.
- Usually, higher performance

Interpreted (Python):
- Program code executed together with translation.
- Translation is repeated each time.
- Quick and easy to make minor changes.

8.3 Static vs Dynamically Typed

C++ is statically typed:
- Each element has a type defined by the programmer.
- Types used fitting together correctly is checked at compilation, yielding *compile time errors* (happen during the program itself) if wrong.

Python is dynamically typed:
- Elements have no type in advance.
- At runtime the type is chosen.
- Type changeable at runtime.
- Depending on the type when executing, there may be *runtime errrors* (happen during the program).
- Errors are more difficult to debug, do not happen all the time.

8.4 Generic Programming

The goal of generic programming is to make code as widely usable as possible (no need for new functions for different types).

Can be done with templates in C++.

No need to do anything in Python thanks to dynamic typing.

8.5 Functional Programming

The central idea is to pass functions as parameters to functions. Example where we pass a function to **"map"**:

```python
#Define a list of numbers
numbers = [1, 2, 3, 4, 5]

#Define a function that squares a number
def square(x):
    return x ** 2

# Use map() to apply the square function to each
# element #in the numbers list
squared_numbers = list(map(square, numbers))

#Print the result
print(squared_numbers) #Output: [1,4,9,16,25]
```

8.5.1 Lambda Functions

Lambda functions are small functions without a specific name, useful to pass into a function as parameter.

```
lambda arguments : expression
```

- Lambda with one argument:

```python
n = [1,2,3,4,5]
sqrd_numbers = map(lambda x : x**2, n)
print(list(sqrd_numbers)) #[1,4,9,16,25]
```

- Lambda with multiple arguments:

```python
y = lambda x,y: x*y
y(5,3) #15
```

8.5.2 Examples of Functions that Accept Functions

- `map(func, it)` – applies a function on each element of a container.

```python
n = [1,2,3,4,5]
sqrd_numbers = map(lambda x : x**2, n)
print(list(sqrd_numbers)) #[1,4,9,16,25]
```

- `filter(func, it)` – removes any elements that don't fulfil a condition.

```python
n = [1,2,3,4,5]
even_numbers = filter(lambda x : x%2==0, n)
print(list(even_numbers)) #[2,4]
```

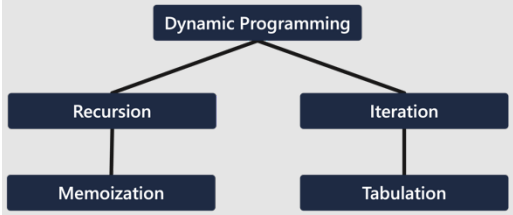
- `reduce(func,it)` – recursively reduce a container to a single value by applying a function to two elements.

```python
from functools import reduce
n = [1,2,3,4,5]
sum_numbers = reduce(lambda x,y: x + y, n) #15
```

9 Dynamic Programming (DP)

DP is a problem-solving strategy:
- It is generally a "bottom-up" strategy – we iteratively solve smaller problems to solve progressively bigger problems.
- It is faster than recursion because we avoid recalculating known solutions.
- We store answers to smaller problems in a table.

```
In dynamic programming, a "top-down" approach usually refers to memoization
On the other hand, a "bottom-up" approach usually refers to tabulation
```



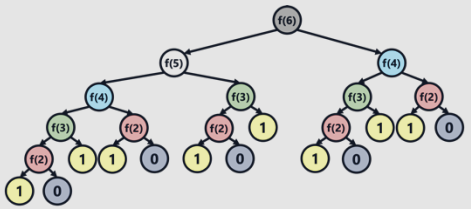
9.1 Where can we use DP?

Problems need to have:
- Optimal substructure – the answer of the problem depends on the answer of some smaller subproblems.

• Overlapping subproblems – in calculating the answer of a problem, we often recalculate the answer to the same subproblems.

Example: Fibonacci

```
fib(n) = fib(n-1) + fib(n-2) #Optimal substructure
```



Recursion tree for fib(6)

9.2 DP Implementation of Fibonacci

```
def fib_DP(n):
    #create table
    F = [None] * (n+1)
    #border cases
    F[0] = 1
    F[1] = 1
    #Bottom-Up for loop
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    #return last value
    return F[n]
```

9.3 Solving Strategy

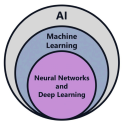
1. Find the first solution, using **brute force or the recursive** implementation. Draw a tree or **visualize** the process.
2. Analyze the solution – **look for** repeating **subproblems**. Then look for an optimal substructure to your solution – how does the answer of the problem depend on the answer of the subproblems? **(Find relationship of subp.)**
3. Think about how to store the answers of the subproblems. Should it be a list? A table?
4. Flip the recursive implementation around to implement a bottom-up, iterative solution.

EXAMPLES IN APPENDIX

10 Machine Learning (ML)

(Supervised) Machine learning is the use of "models" to create functions which map inputs to desired outputs. Generally, there are two areas:

1. Regression: given some input values, what is the output value? Example: given a house has 5 bedrooms, 1000 square meters, 3 bathrooms and is 50m from the nearest train station, what is its price?
2. Classification: given some input values, to what group does the input belong to? Example: given the temperature is 5 degrees and it is cloudy, will it rain?



An existing package for many ML algorithms is sklearn.

10.1 General Procedure

- Select a model, using a technique such as cross validation.
- Read the data, using pandas.
- Split data into test set and train set.

(X (input features) and y (target) need to be provided below)

```
Split the dataset into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
#0.3 of the data will be used in validation
```

- Train the model.

```
from sklearn import linear_model
model = linear_model.LinearRegression()
model.fit(X_train,y_train)
```

- Validate the model to rate how well it performed.

```
from sklearn.metrics import mean_squared_error
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

10.2 Validation Metrics

- Accuracy score $\frac{\#\ correctly\ classified}{total\ number}$. 1 is the best score, 0 is the worst.

```
from sklearn.metrics import accuracy_score
```

- R^2 score. 1 is the best score, the more negative the worse.

```
from sklearn.metrics import r2_score
```

- Mean squared error. 0 is the best score. The bigger the worse.

```
from sklearn.metrics import mean_squared_loss
```

For classification tasks **(discrete categories)**, use **accuracy_score.** For **regression tasks (continuous variables),** use metrics like **mean_squared_error or R-squared** for evaluation.

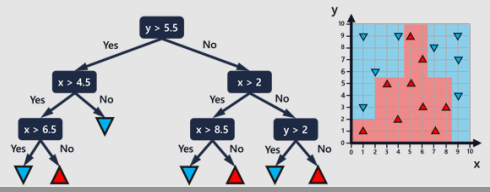
10.3 Classification

10.3.1 Decision Tree Classifier

Given: some data X, and labels y. Constructs a decision tree to divide up the data based on its features.

```
# Before Split the dataset into training and testing sets
from sklearn.tree import DecisionTreeClassifier
# Define the classifier
tree = DecisionTreeClassifier(max_depth=4)

# Train the model
tree.fit(X_train, y_train)

# Predict using the test set
y_pred = tree.predict(X_test)
```

Example:



10.4 Regression

10.4.1 Linear Regression

Given: some data X, and labels y. Constructs a linear model $w^T x + b_0$ so that the loss function (residual sum of squares) is minimized for the data provided.

```
from sklearn import linear_model
model = linear_model.LinearRegression()
#train the model
model.fit(X_train, y_train)
#calculate the prediction of the model
y_pred = model.predict(X_test)
```

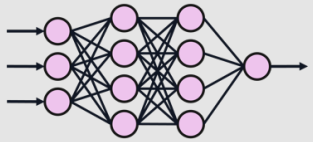
10.4.2 Logistic Regression

```
from sklearn.linear_model import LogisticRegression
# Define the classifier
log_reg = LogisticRegression(max_iter=1000)
# Train the model
log_reg.fit(X_train, y_train)
# Predict using the test set
y_pred = log_reg.predict(X_test)
```

Logistic Regression is a statistical method used for modeling the probabilities of binary outcomes

10.5 Neural Networks

A neural network is an ML algorithm which is incredibly powerful because it can be used for both regression (Predicts continuous values.) and classification (Predicts discrete values (labels or categories)). It is comprised of layers of neurons, where the output of each neuron is made non-linear through an activation function.



Example classification:

```
from sklearn.neural_network import MLPClassifier
#hidden_layer_sizes is an array indicating the #number
#of neurons in each of the hidden layers,
#activation is the activation function
nn = MLPClassifier(hidden_layer_sizes = [4,4], #!
activation="logistic", max_iter=1000) #! indentation
#train the regression model
nn.fit(X_train,y_train)
#calculate the prediction
y_pred = nn.predict(X_test)
```

Example regression:

```
from sklearn.neural_network import MLPRegressor
nn = MLPRegressor(hidden_layer_sizes = [4,4], activation="relu")
# you may set max_iter=1000 The default value in scikit-learn is 200
#train the regression model
nn.fit(X_train,y_train)
#calculate the prediction of the model
y_pred = nn.predict(X_test)
```

10.6 ML Theory Overview

1. Underfitting and Overfitting:
- Underfitting: Model too simple; performs poorly on training/test data.
- Overfitting: Model too complex; captures noise, good on training but bad on test data.

2. K-fold Cross Validation:
- Splits training data into 'K' parts. Model trains on K-1 parts and tests on 1. Repeated K times. Averages used for performance.

3. Grid Search:
- Tunes hyperparameters by testing all combinations within a range. Uses cross-validation to identify the best.

4. Convolutional Neural Networks (CNNs):
- For image recognition tasks:
  - Filters: Extract features by moving over the image.
  - Pooling: Downsamples to reduce dimensions.
  - Fully Connected Layers: Final layers for prediction output.

5. Non-Numerical Data Encoding:
- Ordinal Encodin: Maps categories to ordered integers. E.g., Low -> 1, Medium -> 2.
- Mean Encoding: Assigns each category a value based on the average target variable. Beware of overfitting.
- One-hot Encoding: Creates binary columns for each category value, marking presence (1) or absence (0).

6. Clustering:
- Categorizes data into groups based on similarity, without prior labels.
  - K-means: Divides data into 'K' groups by minimizing intra-cluster distances.
  - Hierarchical: Produces a tree of clusters. Can merge (agglomerative) or split (divisive) groups.

7. Loss Function:
- Assesses model prediction accuracy. Goal is minimization. Examples: Mean Squared Error (regression), Cross-Entropy (classification).

8. Dimension Reduction: - Techniques to lower feature count, preserving essential data characteristics.

10.6.1 Common sorting algorithms

- Bubble sort $O(n^2)$

```
# "bubbles" items on top by going through the array again
and again and taking the biggest element to the top position
it belongs
```

- Insertion sort $O(n^2)$ / if sorted O(n) **Chap. 6.2.4**

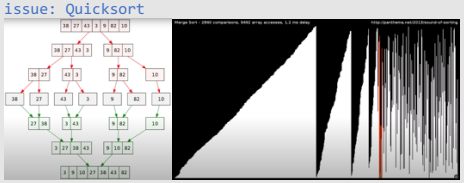
```
# Will move right to left taking the next element as far
left until it is the next biggest too the one on the left
```

- Selection sort $O(n^2)$ / always $O(n^2)$ **Chap. 6.2.3**

```
Finds smallest element in the array and exchanges it with
the element at the beginning (current start index)
```

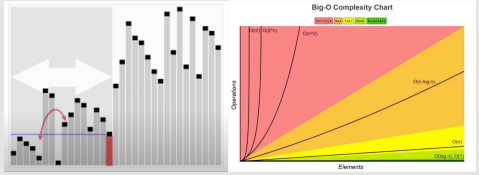
- Merge Sort (Divide and conquer) / O(n*log(n)) **Chap. 6.2.5**

```
# Divides (goes for midpoints splice) , sorts, reassembles
(merge function)
Requires extra space as it doesn't sort in place => if
issue: Quicksort
```



- Quicksort (divide and conquer with pivot) / O(n*log(n)) **Appendix**

```
# Quick Sort, another Divide and Conquer algorithm, uniquely
partitions the array around a chosen 'pivot' element,
placing smaller elements before and larger ones after. (
Space complexity: O(log(n)) )
```



- Heapsort O(n*log(n)) **Chap. 6.2.7 and 7.4 (min, max)**

```
Heap Sort manipulates data using a binary heap data
structure (construction phase), continually removing the
largest element from the heap (extraction phase) and
reconstructing it, resulting in a sorted array. Space
Complexity O(n).
n*log(n) also called quasilinear, of not too big dataset
```

SORTING ALGORITHMS TIME COMPLEXITY OVERVIEW

Algorithm	Time Complexity			Worst Case Space Complexity
	Best	Average	Worst	
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Quicksort	O(n log n)	O(n log n)	O(n²)	O(n)
Bucket Sort*	O(n + k)	O(n + k)	O(n²)	O(n)
Radix Sort**	O(n * k)	O(n * k)	O(n * k)	O(n+k)
Counting Sort***	O(n+k)	O(n +k)	o(n+k)	O(k)

*k represents the number of buckets
**k represents the number of digits in the largest number in the array
***k represents the difference between the smallest and the largest array value

11 Appendix

11.1.1 Code Snippet Overview

(Not directly in Appendix below) : mergeSort 6.2.5
1D DP: chap 9.2 Fibonacci
Sorting Algos Code locations are referenced in 10.6.1
For main Topics:
[6.3 SearchAlgo],[7.3 Binary Search Tree],[7.8 Quadtrees],[8.1.1 OOP],[8.1.3 Inheritance]

CODE (codes in black boxes not under correct chap. (space reasons))

PANDAS EXAMPLE

```
--- DATA ---
2021-01-07,AG,johnson_johnson,0,vaccine,694072,0,0,0,0,0,COVID19VaccDosesAdministered
,2022-02-22_07-01-47,detailed
2021-01-08,AG,johnson_johnson,0,vaccine,694072,0,0,0,0,0,.....
-----------
```

```python
import pandas as pd
def read_and_filter_data():
    """Return dataframe from CSV with specific columns filtered by 'detailed'
    granularity."""
    data = pd.read_csv(
        "COVID19AdministeredDoses_vaccine.csv",
        usecols=["date", "geoRegion", "vaccine", "entries", "pop",
                 "sumTotal", "granularity"],
                 comment="#")
    data = data[data["granularity"] == "detailed"]
    data = data.drop(columns = "granularity")
    return data

def question_one(data):
    """Return the total vaccine doses across all geoRegions."""
    return data["entries"].sum()

def question_two(data):
    """Return the maximum administered doses in any region."""
    return data["sumTotal"].max()

def question_three(data):
    """Return doses per capita for geoRegion "BE"."""
    be_data = data[data["geoRegion"] == "BE"]
    assert len(be_data["pop"].unique()) == 1
    be_population = be_data["pop"].iloc[0]
    return be_data["entries"].sum() / be_population

def question_four(data):
    """Return percentage of total doses administered by each vaccine."""
    vaccine_types = data["vaccine"].unique()
    total_doses = question_one(data)
    percentages = {vt: data[data["vaccine"] == vt]["entries"].sum() / total_doses
                   for vt in vaccine_types}
    return percentages
```

Bubble Sort

```python
def bubbleSort(array):
    n = len(array)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
                swapped = True
        if not swapped:
            break
```

```python
# DP Unbounded Knapsack (Repetition of items allowed)
def unboundedKnapsack(W, n, val, wt):
    dp = [0 for i in range(W + 1)]
    for i in range(W + 1):
        for j in range(n):
            if (wt[j] <= i):
                dp[i] = max(dp[i], dp[i - wt[j]] + val[j])
    return dp[W]
```

Quick Sort

```python
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
    array[i + 1], array[high] = array[high], array[i + 1]
    return i + 1


def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)
```

Dict comprehension

```python
def bridge_lengths(
  names: list, lengths: list, completion_years: list, min_length: int) -> dict:
  return {names[i]: completion_years[i] for i in range(len(names)) if lengths[i] >= min_length}
```

3D Memoization Example

```python
# Memoization using A matrix with a dict as third dimension
# could also be solved with a 3D list but stop at neg. index!
def countMemoized(A,cost,x,y,M = None):
  if M == None:               #! new DP memo part
    M = [[{} for j in range(0,len(A))] for i in range(0,len(A))]
  if cost in M[x][y]:         #! new DP memo part
    return M[x][y][cost]
  else:                       #! new DP memo part
    if x == 0 and y == 0:
      result = 1 if (A[x][y] == cost) else 0
    elif x == 0:
      result = countMemoized(A, cost-A[x][y],x,y-1,M)
    elif y == 0:
      result = countMemoized(A, cost-A[x][y],x-1,y,M)
    else:
      result = countMemoized(A, cost-A[x][y],x-1,y,M) + countMemoized(A, cost-A[x][y],x,y-1,M)
    M[x][y][cost] = result  #! new DP memo part
  return M[x][y][cost]

def computeCount(A,cost):
  n = len(A)
  count = countMemoized(A,cost,n-1,n-1) # hint: you may want to change this call
  return count
```

```python
def subsequence(nums: list) -> int: # (Sliding Window app.)
    """Determine the length of the longest subsequence in 'nums'
    where the difference between the largest and smallest values is 1.
Args: nums: Sorted list of integers.
Returns: Length of the specified subsequence.
"""
    l = r = 0
    length = 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
```

```python
#given a string s consisting of words and spaces
#return length of last word
def lengthOfLastWord(self, s):
    """
    :type s: str
    :rtype: int
    """
    return len(s.strip().split(" ")[-1])
```

Python Linked lists

```python
def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode(0)  # Create a dummy node to start the new list
        last = dummy  # This will always point to the last node in the new list
        p1 = list1
        p2 = list2
        while p1 is not None and p2 is not None:
            if p1.val <= p2.val:
                last.next = p1
                p1 = p1.next
            else:
                last.next = p2
                p2 = p2.next
            last = last.next
        # At this point, at least one of p1 and p2 is None, so we append the non-None list to the end
        if p1 is not None:
            last.next = p1
        else:
            last.next = p2
        return dummy.next  # The first node is the dummy node, so we return the next node
```

ML

ML General Blueprint

```python
import numpy as np
import pandas as pd

def main(): # CIRCLES MASTER SOLUTIONS
    # Task 1: Load the data into a Pandas data frame.
    df = pd.read_csv("data.csv") #! If it doesnt work see other snippets
    """ #! EXAMPLE OF OTHER df creation
    df = pd.read_csv("data.csv")
    X = df.drop(['diagnosis'], axis=1)
    y = df['diagnosis']
    """
    X = df[["Feature1", "Feature2"]].to_numpy()
    y = df['Label'].to_numpy()

    # Task 2: Split the dataset into training and testing sets
    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)

    # Task 3: Train at least two different models on the training set
    #! For more Models go to chapter 10. Dont forget to import the libraries
    # Train a neural network (Multi-Layer Perceptron) on the dataset
    from sklearn.neural_network import MLPClassifier

    mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=1000, random_state=42)
    mlp.fit(X_train, y_train)

    # Train a linear regression model on the dataset
    from sklearn.linear_model import LogisticRegression

    lr = LogisticRegression()
    lr.fit(X_train, y_train)

    # Train a decision tree model on the dataset
    from sklearn.tree import DecisionTreeClassifier

    dt = DecisionTreeClassifier()
    dt.fit(X_train, y_train)

    # Task 4: Evaluate the performance of all models on the test set
    from sklearn.metrics import accuracy_score

    y_pred_mlp = mlp.predict(X_test)
    y_pred_lr = lr.predict(X_test)
    y_pred_dt = dt.predict(X_test)
    accuracy_mlp = accuracy_score(y_test, y_pred_mlp)
    accuracy_lr = accuracy_score(y_test, y_pred_lr)
    accuracy_dt = accuracy_score(y_test, y_pred_dt)

    print("MLP Accuracy: {:.2f}".format(accuracy_mlp))
    print("Logistic Regression Accuracy: {:.2f}".format(accuracy_lr))
    print("Decision Tree Accuracy: {:.2f}".format(accuracy_dt))
#   Task 5: Choose a model and use it to make predictions for the points in X_final.

    df = pd.read_csv("X_final.csv")
    y_final = mlp.predict(df)
    return y_final
# Train MLP classifier SPIRALS
mlp = MLPClassifier(hidden_layer_sizes=(32,32,16,16,8), max_iter=5000, random_state=0)
mlp.fit(X_train, y_train)
```

Grind search:

```python
import pandas as pd
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.tree import DecisionTreeRegressor #1
from sklearn.pipeline import Pipeline #2
from sklearn.linear_model import LinearRegression #2
from sklearn.preprocessing import PolynomialFeatures #2
from sklearn.metrics import mean_squared_error, r2_score

def foo(model, param_grid):
    # Load the data
    df = pd.read_csv("data.csv")
    X = df.drop(['target'], axis=1) # make sure correct one is selected
    y = df['target']

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Grid search
    grid = GridSearchCV(model, param_grid=param_grid, cv=5)
    grid.fit(X_train, y_train)

    # Best model
    best_model = grid.best_estimator_

    # Evaluate the model
    y_pred = best_model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print("R2 on test data: {:.3f}".format(r2))

    # Make predictions on new data
    X_final = pd.read_csv("X_final.csv")
    y_final = best_model.predict(X_final)

    return y_final

# Choose a model and a parameter grid
# TODO: Uncomment one model and one parameter grid
# model = #1 DecisionTreeRegressor()
# param_grid = #1 {'max_depth': [1, 2, 3, 4, 5], 'criterion': ["squared_error", "friedman_mse"]}

# model = #2 Pipeline([("transf", PolynomialFeatures()), ("lr", LinearRegression())])
# param_grid = #2 {"transf__degree" : range(1, 10)}

# foo(model, param_grid)
```

```python
def Aufgabenplanner_v2(w1, t1, w2, t2):
  n = len(w1)
  s = [0]*(n+1)

  s[n] = 0
  for i in range(n-1, -1, -1):
    maxw1= max(w1[i]+s[i+t1[i]], s[i+1])
    maxw2= max(w2[i]+s[i+t2[i]], s[i+1])
    s[i] = max(maxw1, maxw2)
  return s[0]
```

DP

```python
# DP for 0-1 Knapsack problem
# W: Maximum capacity of the knapsack.
# wt: List of weights of each item.
# val: List of values of each item.
# n: Number of items.
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    # Build table K[][] in bottom up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] #! indent
                              + K[i-1][w-wt[i-1]], #! indent
                              K[i-1][w]) #! indent
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
```

```python
def reverse_string(text, interval):
  reverse = ""

  for i in range(0, len(text), interval):
    reverse += text[i:i+interval:1][::-1]

  return reverse
```

```python
#Return the number of combinations that make up that
#amount. If that amount of money cannot be made up by
#any combination of the coins, return 0. Coins is a
#list of denomanations
def coin_change2(self, amount, coins):
    """
    :type amount: int
    :type coins: List[int]
    :rtype: int
    """
    n = amount + 1
    dp = [0]*(n)
    #base case: there is one way to make 0, take no coins
    dp[0] = 1
    for coin in coins:
        for i in range(n):
            if i - coin >= 0:
                dp[i] += dp[i-coin]

    return dp[amount]
```

```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:
        northeast = s[i-1][j+1] if i > 0 else -1
        east = s[i][j+1]
        southeast = s[i+1][j+1] if i < m - 1 else -1
        if northeast >= east and northeast >= southeast:
          s[i][j] = a[i][j] + northeast
          L[i][j] = "North east"
        elif east >= northeast and east >= southeast:
          s[i][j] = a[i][j] + east
          L[i][j] = "East"
        else:
          s[i][j] = a[i][j] + southeast
          L[i][j] = "South east"


  path = []
  i, j = 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
```

```python
# return the maximum number of pieces to cut n with pieces of lengths from p
# if no possibility to cut without rest, -1 is returned
def Rod_Cutting_DP(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

  return solution[n]
```

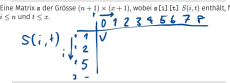
```python
def longestCommonSubsequence(self, text1: str, text2: str) -> int:
    dp = [[0 for j in range(len(text2) + 1)] for i in range(len(text1) + 1)]
    for i in range(len(text1) - 1, -1, -1):
        for j in range(len(text2) - 1, -1, -1):
            if text1[i] == text2[j]:
                dp[i][j] = 1 + dp[i + 1][j + 1]
            else:
                dp[i][j] = max(dp[i][j + 1], dp[i + 1][j])
    return dp[0][0]
```

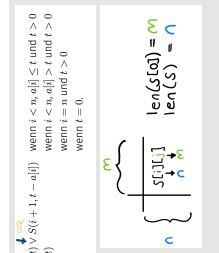
```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for a in range(1, amount + 1):
            for c in coins:
                if a - c >= 0:
                    dp[a] = min(dp[a], 1 + dp[a - c])
        return dp[amount] if dp[amount] != amount + 1 else -1
```

Binomial Coefficient

```python
def compute_binomial_coefficient(n, k):
    coefficients = [[0] * (k + 1) for _ in range(n+1)]

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

    return coefficients[n][k]
```

```python
def subsetsum(a, x):
  n = len(a)
  s = [[None] * (x + 1) for _ in range(n + 1)]
  L = [[None] * (x + 1) for _ in range(n + 1)]

  for i in range(n, -1, -1):
    for t in range(x + 1):
      if t == 0:
        s[i][t] = True
      elif i == n:
        s[i][t] = False
      elif a[i] > t:
        s[i][t] = s[i+1][t]
        L[i][t] = "no"
      else:
        s[i][t] = s[i+1][t] or s[i+1][t - a[i]]
        if s[i+1][t - a[i]]:
          L[i][t] = "yes"
        else:
          L[i][t] = "no"

  subset = []
  i, t = 0, x
  while i < n and t > 0:
    if L[i][t] == "yes":
      subset.append(a[i])
      t -= a[i]
      i += 1
    else:
      i += 1

  return s[0][x], subset
```





Binary Insertion Sort

```python
def binary_search(li, target):
  l = 0
  r = len(li) - 1
  while l <= r:
    m = (l+r)//2
    if li[m] < target:
      l = m + 1
    else:
      r = m - 1
  return l

def insertion_sort_binary(li):
  for i in range(1, len(li)):
    if li[i] < li[i-1]:
      j = binary_search(li[:i], li[i])
      li.insert(j, li.pop(i))
  return li
```

Farthest Point

```python
def get_mean(S):
  x_total = 0.0
  y_total = 0.0
  for i in range (0, len(S)):
    x_total += S[i][0]
    y_total += S[i][1]
  x_mean = x_total / len(S)
  y_mean = y_total / len(S)
  return np.array([x_mean, y_mean])

def furthest_away_from_mean(S):
  distances = {}
  mean_point = get_mean(S)
  for i in range (0, len(S)):
    distance = ((mean_point[0] - S[i][0])**2
                + (mean_point[1] - S[i][1])**2)
    distances[distance] = (S[i][0], S[i][1])
  max_distance = max(distances.keys())
  x_max = distances[max_distance][0]
  y_max = distances[max_distance][1]
  return np.array([x_max, y_max])
```

11.1.2 String manipulation terms

A subsequence is a sequence derived from another by deleting some elements but preserving order, not necessarily containing adjacent elements.
A substring is a subsequence with adjacent elements.
A palindrome is a sequence or subsequence, including substrings, that reads the same forwards and backwards.

11.1.3 Additional Information

An **AVL tree** is a type of self-balancing binary search tree in computer science. It maintains balance by ensuring the heights of the left and right subtrees of any node differ by at most one. This leads to efficient insertions, deletions, and look-ups, all with a time complexity of O(log n), making AVL trees useful in databases and file systems.
Greedy Algorithm: A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. // Theory BAUG: Tree traversal O(n) // Inserting into AVL tree O(log(n))

11.1.4 APPENDIX 2. Note to the reader

This is a revised version of the original summary, kindly provided by Julian Lotzer and Daniel Steinhauser. The new code snippets were created using VSC in the high-contrast light theme and (most of them) tested. In case some code snippets shouldn't work, please contact me:
lehmannni@ethz.ch
(No responsibility is taken by the author for the correctness of the additional code snippets)

Tip for future versions: Add more ML theory. In the exam, there were questions regarding filters (chapter about CNN's) and principal component analysis. See exam collection: Info II exam 2023 D-MAVT.

It's recommended to print this cheat sheet using a printer on high-quality settings, like an HP OfficeJet Pro, due to the small font size.