Informatik 2 Cheatsheet

Julian Lotzer – jlotzer@student.ethz.ch
Daniel Steinhauser – dsteinhauser@student.ethz.ch
Modified by: Christian Leser - cleser@ethz.ch

Version: November 2, 2023
Template by Micha Bosshart

1 General Python

1.1 Reference Semantics and Aliasing

Everything is a pointer: l1 and l2 point to the same adress

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

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 # <class 'bool'>
s = "Hello World" # <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

help make code more legible

```
def add_integers(li: list[int]) -> int:
  ...
# after colon: expected input type
# after arrow: expected return type
```

1.3 Input and Output

Output

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

Input

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

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

if, else

```
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
else:
  print("big number") #x is positive and the ifs/elifs conditions don't hold
```

while

```
x = 0
while x <= 3:
  x += 1
```

for with range

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

for with lists

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

1.5 Functions

Functions do not have to be declared in a specific order

1.5.1 Function Declaration

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

1.5.2 Default arguments

When calling a function with default arguments, it is not necessary to call the function with arguments.

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

1.5.3 Global and local variables

Avoid this kind of code!

global variables can be used within a function if declared before the funciton call. Also, local variables can be made global:

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

2 Algorithms

2.1 Invariants

The invariant is a condition in an algorithm involving a loop.

Phase	Description	Example (list 'a' is sorted until index 'i')
1. Initialization	the condition is met before the loop	i = 0, a[:i] = a[0]
2. Continuation	the condition holds at each iteration	i = j, a[:j]
3. Termination	the condition holds at the end of the loop	i = n, a[:n] = a[n]

2.2 Divide and Conquer Algorithms

1. Divide Problem into easier to solve subproblems
2. Solve subproblems
3. Reassemble solutions from subproblems

Examples: Sorting Algorithms for lists

- Mergesort
- Quick Sort

3 Python Containers

3.1 Operations on Containers

Number of Elements:

```
len(c)
```

Contains element x?

```
x in c
```

Iterate over all elements:

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

3.2 Sequences (ordered containers)

3.2.1 Sequence Operations (Überarbeiten)

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).

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)
```

3.2.2 List (mutable)

- ordered data
- Fast Access via index
- Slow for updates

Initialise a list with []

```
l = [1, 3, "hi", -4]
M = [[-1 for i in range(n)] for j in range(m)]
# 2D list or m x n-Matrix filled with -1
```

Common List Operations

```
l[i] = value #Access and change item
l.pop(i) #Delete element at index i
l.remove(v) #Delete element with value v
l.insert(i, v) #Insert element with value v at i
l.append(value) #Add item at the end
del l[i] #Remove item at location i
l.reverse()
l=[v]*k #Create a list of k elements with value v
```

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):

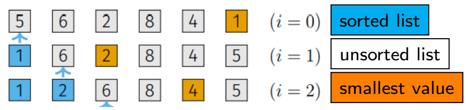
```
l3 = [f(x) for x in l if g(x)]
```

Example: Read a sequence of numbers:

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

3.2.3 Sorting Algorithms for Lists

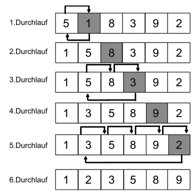
Selection Sort



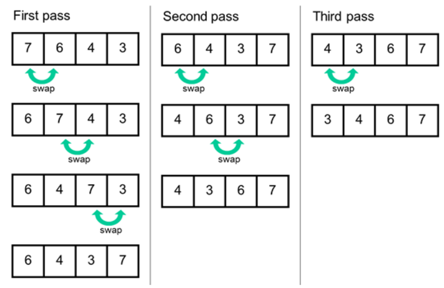

```
#Input: Array a = (a[0],...,a[n])
#Output: Sorted Array a
def sort(a):
    n = len(a)
    for i in range(n):
        mini = i
        for j in range(i+1, n):
            if a[j] < a[mini]:
                mini = j
        a[mini],a[i] = a[i],a[mini]
    return a
```

Insertion Sort

```
def insertionSort(arr):
    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```



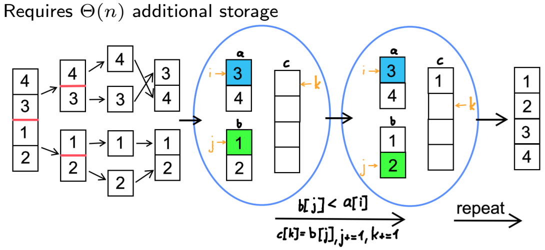
Bubble Sort




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

Mergesort (Divide and Conquer)

Requires $\Theta(n)$ additional storage



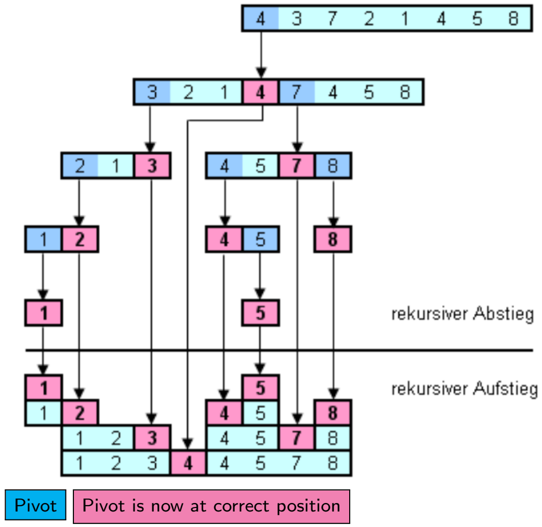
greater value | smaller value

```
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)
```

Quick Sort (Divide and Conquer)

Requires no additional storage

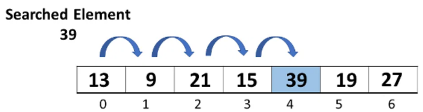



```
def partition(a, l, r):
    p = a[r]
    j = l
    for i in range(l, r):
        if a[i] < p:
            a[i], a[j] = a[j], a[i]
            j += 1
    a[j], a[r] = a[r], a[j]
    return j

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

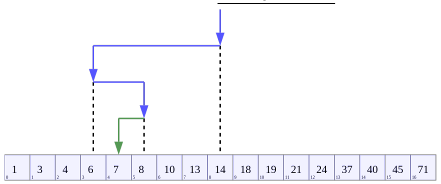
3.2.4 Search Algorithms for Lists

Linear Search




```
#Input: array a, key b
#Output: index k with a[k] = b or "not found"
def LinearSearch(a, b):
    for i, x in enumerate(a):
        if x == b:
            return i
    return "not found"
```

Binary Search



```
#Input: sorted array a, key b
#Output : index k with a [ k ] = b or " not found "
def bin_search(a, l, r, b):
    if r < l:
        return None
    else:
        m = (l + r) // 2
        if a[m] == b:
            return m
        elif b < a[m]:
            return bin_search(a, l, m-1, b)
        else: # a[m] > b
            return bin_search(a, m+1, r, b)
```

3.2.5 Tuple (immutable)

Initialise a tuple with ()

```
t = ("a", 0, -6, 3.3)
```

3.2.6 Range (immutable)

Initialise a range

```
#range(start, stop, step)
#befault: start = 0, step = 1, end not included
r = range(0, 8, 2) #r ->  0   2   4   6,
```

3.2.7 String (immutable)

Initialise a string with ""

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

Common String Operations

```
s1 = "hello"
s2 = " world"
s1[1] #Access element at position 2 -> e
s3 = s1 + s2 ##Add strings (concatenating) ->s3 = "
    hello world"
s2=s2.strip() #Remove whitespace at beginning and
    end -> "world"
s = list(s1) #Convert string into list of chars
```

Example: check if s is a string with content:

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

Example: convert a string to a list of words:

```
s = "Hello World"
w = s.split() #-> w = ['Hello', 'World']
s.split(seperator, maxsplit)
#seperator and maxsplit are optional
#s.split() -> split at all whitespaces
#separator = ", " -> split at every ", "
#maxsplit = 10 -> split only at first 10 separators
```

3.3 Collections (unordered containers)

3.3.1 Set (non-associative)

Initialise Set

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

Common Set Operations

```
s.add(69) #Add item
s.remove(29) #Remove item
12 in S #Search for item, returns bool
```

3.3.2 Dictionary (associative)

See 3.2.1 Sequence operations for 'enumerate' and 'zip'

Initialise a Dictionary with {}

A dictionary consists of **tuples (key, value)** as items. For that reason, one can think of it as a list of tuples (Which it is not in reality)

```
d = {"Lea":22, "Tim":19, "Mortis":69} #key:value
#Merge two lists into one dictionary
cities = ["Zurich", "Basel", "Bern"] #list 1
code = [8000, 4000, 3000] #list 2
d2 = dict(zip(cities,code)) #dictionary D2
```

Common Dictionary Operations

```
d["Lea"] = 23 #Change item
d["Peter"] = 24 #Add item
#no key can exist more than one time
del d["Mortis"] #delete item
"Tim" in d #Search for key, returns bool
d["Tim"] #access value at a key, output: 19
```

Iterate over a Dictionary

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

Dictionary Comprehension

Transform a set into a dictionary by applying $f(x)$ and $g(x)$ on every element in the set:

```
d3 = {f(x):g(x) for x in s} #s being a set
d4 = {f(x):g(y) for x, y in z.items()} #z being a
    dict
```

Transform a set into a dictionary, only if the element satisfies h(x):

```
d5 = {f(x):g(x) for x in s if h(x)}
```

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}
```

4 Python Classes

- entity with a name that contains data and functionality
- bundling of data that belongs together contentwise
- definition of a new type
- every object of this type stores data and offers functionality

```
class Class_name_1:
    def __init__(self, att1, att2): #constructor
        self.attribute1 = att1
        self.attribute2 = att2

#the constructor is called to create object1
object1 = Class_name_1(1, 1)

class Class_name_2: #type definition
    attribute3 = None
    attribute4 = Class_name_1(0, 0)

    def method1(self, parameter1):
        ...

    def __eq__(self, other): #overloads ==
        return (self.attribute3 == other.attribute3
     and self.attribute4 == other.attribute4)

object2 = Class_name2() #generate object
object3 = Class_name2()
object2.attribute3 = 64 #write attribute
print(object_name.attribute1) #read attribute
object2 == object3 #True, uses overloaded ==
object2.method1(...) #call function
```

4.1 Private attribute

```
class ClassName:
    __private = 0 #private attribute

    def set_privateat(self, a):
        self.__privateat = a

    def get_private(self):
        return self.__private
```

4.2 Inheritance

- Child inherits all attributes and methods from Parent
- Child can define additional attributes and methods

```
class Parent:
    def __init__(self, x1):
        self.x = x1
    def double_x(self):
        self.x *= 2

class Child(Parent):
    def __init__(self, x1, y1):
        Parent.__init__(self, x1)
        self.y = y1
    def double_y(self):
        self.y *= 2

c = Child(2, 2)
c.double_x()
print(c.x, c.y) #Output: 4 2
```

4.3 Magical Methods

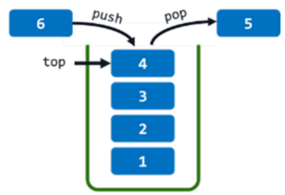
Operation	Meaning	Magical Method
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__
==	Equal to	__eq__
!=	Not equal to	__ne__
+, +=	Addition	__add__, __iadd__
-	Subtraction	__sub__
*	Multiplication	__mul__
/	Division	__truediv__
//	Integer division	__floordiv__
%	Modulo	__mod__
**	Exponentiation	__pow__
print()	overload print()	__str__
-	Negation	__neg__

5 Data Structures

5.1 FIFO / LIFO

FIFO - first in, first out, fast access to first inserted element

LIFO - last in, first out, fast access to last inserted element



5.2 Search Tree

- sorted data
- all operations fast
- well organized

5.2.1 Tree terminology

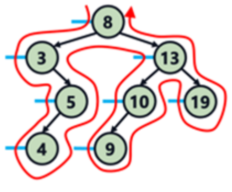
- Root: Highest node has children and no parent
- Order: Maximum number of child nodes (BST order = 2)
- Height: Maximum path length root to leaf
- Degenerated: Every Node has only one child, leading to a linked list structure
- Complete: Every level except for the lowest is filled, lowest level is filled from left to right
- Perfect: Every level is completely filled
- Full: the amount of children of every node equals either the order or 0
- ALV-Tree: balanced binary search tree $(-1 \le \text{height(n.left)} - \text{height(n.right)} \le 1)$

5.2.2 Binary Tree Traversal

Preorder

```
def preorder(node):
    print(node.key)
    preorder(node.left)
    preorder(node.right)
```

If a list 'a' was the inorder traversal of a tree: the tree can always be recreated
⇒ **Representation unique**

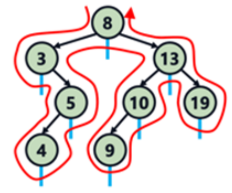


Preorder: path touches line in following order:
8, 3, 5, 4, 13, 10, 9, 19

Inorder

```
def inorder(node):
    inorder(node.left)
    print(node.key)
    inorder(node.right)
```

Entries are printed in ascending order
If a list 'a' was the inorder traversal of a tree: the tree can be recreated only if the list is in ascending order
⇒ **Representation not unique**

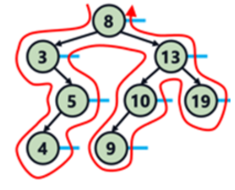


Inorder: path touches line in following order:
3, 4, 5, 8, 9, 10, 13, 19

Postorder

```
def postorder(node):
    postorder(node.left)
    postorder(node.right
    )
    print(node.key)
```

If a list 'a' was the inorder traversal of a tree: the tree can always be recreated
⇒ **Representation unique**

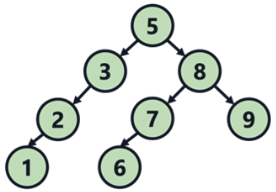


Postorder: path touches line in following order:
4, 5, 3, 9, 10, 19, 13, 8

5.2.3 Binary Search Tree (BST)

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



Implementation

```
class SearchNode:
    def __init__(self, k, l=None, r=None):
        self.key = k
        self.left, self.right = l, r

class Tree:
    def __init__(self):
        self.root = None
    def find(self,key):
        return findNode(self.root, key)
    def add(self,key):
        self.root = addNode(self.root, key)
```

Height

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

Search for Node

```
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
```

Insert Node

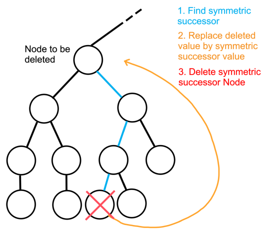
```
def addNode(root, key):
    if root == None:
        root = Node(key)
    n = root
    while n.key != key:
        if key < n.key:
            if n.left == None:
                n.left = Node(key)
            n = n.left
        else:
            if n.right == None:
                n.right = Node(key)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

Remove Node

Cases:

1. node has no children: set variable to None
2. node has one child: replace node with child
3. node has two children: replace node with symmetric successor – the next biggest element

Case 3:



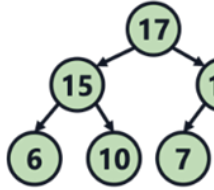

```
def findSuccessor(start_node):
    parent = None
    node = start_node.right
    while node.left is not None:
        parent = node
        node = node.left
    return node

def deleteNode(root, key):
    if root is None:
        return None
    # Find the node to be deleted
    if key < root.val:
        root.left = deleteNode(root.left, key)
    elif key > root.val:
        root.right = deleteNode(root.right, key)
    else:
        # Case 1: Node has no children
        if root.left is None and root.right is \
            None:
            root = None
        # Case 2: Node has one child
        elif root.left is None:
            root = root.right
        elif root.right is None:
            root = root.left
        # Case 3: Node has two children
        else:
            successor = findSuccessor(root)
            root.val = successor.val
            root.right = deleteNode(root.right,\
                successor.val)
    return root
```

5.2.4 (Max) Heaps

Heaps are only visualised as trees while they are stored as arrays
Useful for quick access to max (/ min) value

- complete binary tree (see 5.2.1 Tree terminology)
- Key of parent is always greater (smaller) than the one of its children



17	15	12	6	10	7

Array corresponding to the max heap

Implementation

$\text{Heap}[i = 1] = \text{Root}$:

- Children of i: 2i+1, 2i+2
- Parent of i: i-1//2

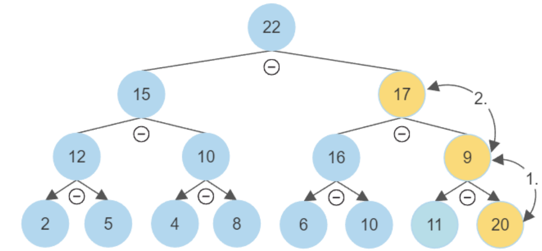
Height

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

```
# a is a heap with n elements
def height(a):
    return math.log(len(a) + 1, 2)
```

Sift Up

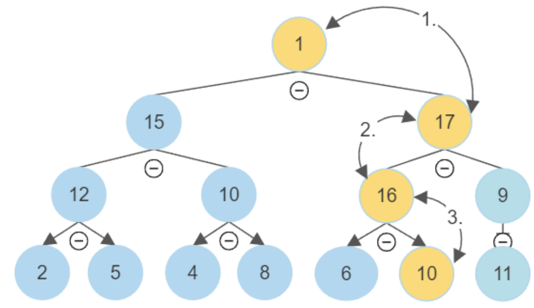
Reestablishes Heap structure, Used in 'Insert'



```
def SiftUp(a, m):
    v = a[m]
    c = m
    p = c//2
    while c > 0 and v > a[p]:
        a[c] = a[p]
        c = p
        p = c//2
    a[c] = v
```

Sift Down

Reestablishes Heap structure, Used in 'Remove' and 'Heapify'
If the parent value is smaller: exchange the parent value with the greater children value



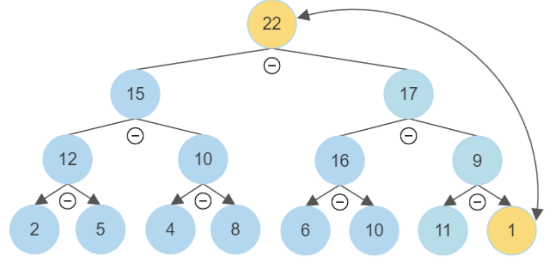
```
#i is the index of the element that needs to be
    sifted down. a is the array. m is the end of
    the array
def SiftDown(a, i, m):
    while 2*i + 1 < m:
        j = 2*i + 1
        if j + 1 < m and a[j] < a[j + 1]:
            j = j + 1
        if a[i] >= a[j]:
            break
        a[i], a[j] = a[j], a[i]
        i = j
```

Insert

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

Remove Max Value

Change the first and last entry in the array, then delete the last entry. Now, the max value does not exist any more.



Reestablish the Heap condition by applying Sift Down.

```
def removeMax(heap):
    if len(heap) == 0:
        return None
    max_val = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    SiftDown(heap, 0, len(heap)-1)
    return max_val
```

Heap creation / Heapify

Repeatedly apply Sift Down until the array is heapified.
Leaves fulfill the heap condition trivially → only "heapify" the first n/2 elements.

```
def heapify(a):
    n = len(a)
    for i in range(n//2 - 1, -1, -1):
        SiftDown(a, i, n)
```

Sorting a heap

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

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

5.3 Linked List

- ordered data
- fast updates in first elements

Implementation

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

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

Traversal

```
def print_elements(l):
    current = l.head
    while current != None:
        print(current.value)
        current = current.Next
```

Search

```
def search(l, v):
    current = l.head
    while current != None:
        if current.value == v:
            return current
        current = current.Next
    return None
```

Insert

```python
def insert_after_node(node, value):
    new_node = Node(value, node.Next)
    node.Next = new_node

def insert_after_value(l, value, new_value):
    node = search(l, value)
    if node != None:
        insert_after_node(node, new_value)
```

Remove

```python
def remove_after_node(node):
    to_remove = node.Next
    if to_remove == None:
        return
    else:
        node.Next = to_remove.Next

def remove_after_value(l, value):
    node = search(l, value)
    if node != None:
        remove_after_node(node)
```

Convert List to Linked List

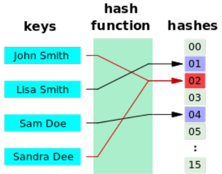
```python
"""Create and return linked list from list a."""
def create_from_list(a):
    l = Linked_List(Node(a[0]))
    last = l.head
    for v in a[1:]:
        last.Next = Node(v)
        last = last.Next
    return l
```

5.4 Hash Tables

- fast access to elements
- Unsorted, unordered data
- (O) only in expectation

Given an element 'x' to be stored, a Hash Table 'M' of length 'l' and a hash function 'f(x)':

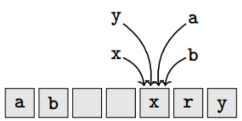
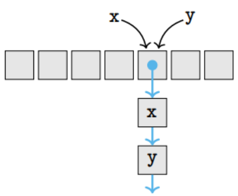
```python
M[f(x) % l] = x
```



example of collision

5.4.1 Collision handling

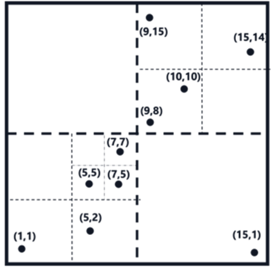
Entry at calculated index may already contain an element

Probing: next available index is chosen	Chaining: a linked list at every entry
	

5.5 Quadtree

- Faster searching for points within a rectangle
- Efficient for querying multiple rectangles

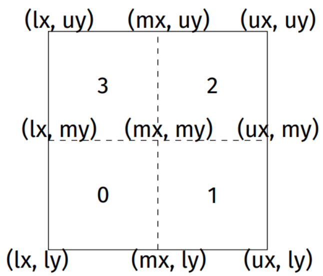
Data containing two values are inserted into squares. The squares have a maximum number of points they may contain. If a new point would exceed that limit, the square is subdivided into four smaller squares.



Implementation

```python
class QuadTree:
    def __init__(self, l, u, max_cap):
        self.l = l #lower left corner coordinate
        self.u = u #upper right corner coordinate
        self.m = max_cap
        self.points = []
        self.children = None
        self.count = 0 #points within quadtree
```

Insert

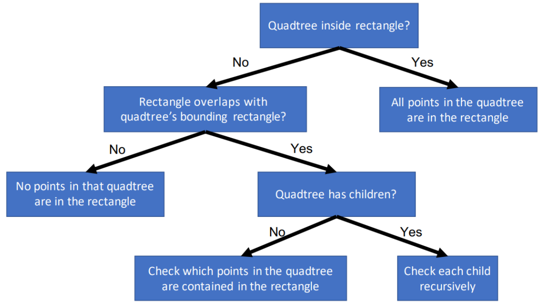



```python
def subdivide(self):
    lx, ly = self.l
    ux, uy = self.u
    mx, my = (lx + ux) / 2, (ly + uy) / 2
    self.children = [None, None, None, None]
    self.children[0] = QuadTree((lx, ly),(mx, my), self.m)
    self.children[1] = QuadTree((mx, ly),(ux, my), self.m)
    self.children[2] = QuadTree((mx,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

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


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 = []
```

Search




```python
def rect_intersect(self, lr, ur):
    def intervals_overlap(a_start, a_end, b_start, b_end):
        a_end_in_b = b_start <= a_end <= b_end
        b_end_in_a = a_start <= b_end <= a_end
        return a_end_in_b or b_end_in_a
    self_lx, self_ly = self.l
    self_ux, self_uy = self.u
    rect_lx, rect_ly = lr
    rect_ux, rect_uy = ur
    return (intervals_overlap(self_lx, self_ux,
                              rect_lx, rect_ux) and
            intervals_overlap(self_ly, self_uy,
                              rect_ly, rect_uy))

def covered_by_rect(self, lr, ur):
    self_lx, self_ly = self.l
    self_ux, self_uy = self.u
    rect_lx, rect_ly = lr
    rect_ux, rect_uy = ur
    return (rect_lx <= self_lx and
            rect_ly <= self_ly and
            self_ux <= rect_ux and
            self_uy <= rect_uy)


def count_in_rect(self, lr, ur):
    if self.rect_intersect(lr, ur):
        if self.children is None:
            pts = [p for p in self.points if lr[0] <= p[0] <= ur[0] and lr[1] <= p[1] <= ur[1]]
            return len(pts)
        else:
            return sum(c.count_in_rect(lr, ur) for c in self.children)
    elif self.covered_by_rect(lr, ur):
        return self.size()
    else:
        return 0
```

6 Runtime analysis

6.1 Runtime analysis

6.1.1 upper, lower and tight bound

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

- Upper bound (If g does not grow faster than c*f): $g = \mathcal{O}(f)$
- Tight Bound (If g grows about the same as c*f): $g = \Theta(f)$
- Lower Bound (If g does not grow slower than c*f): $g = \Omega(f)$

6.1.2 Asymptotic behaviour of funcitons

For all functions: $\lim\limits_{n\to\infty}$

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

$$n = \mathcal{O}(n^2) \to f(n) = \mathcal{O}(g(n)) \to \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq C, C \in \mathbb{R}$$

Useful conversions:

$$B^a \cdot B^b = B^{a+b} \qquad \log_B(a \cdot b) = \log_B(a) + \log_B(b)$$

$$\frac{B^a}{B^b} = B^{a-b} \qquad \log_B\left(\frac{a}{b}\right) = \log_B(a) - \log_B(b)$$

$$(B^a)^b = B^{a \cdot b} \qquad \log_B(a^r) = r \cdot \log_B(a)$$

6.1.3 Code Runtime Analysis

Assumptions of 'time cost':

```python
#Comparisons cost 1
if x==1
if x>1
#Mathematical operations cost 1
6 + 4
a * b
#Assignment cost 1
x = 5
```

Example of runtime analysis on selection sort:

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

Useful Sums

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

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

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

$$\sum_{i=0}^{n}\sum_{j=0}^{i} j = \frac{1}{12}n(n+1)(2n+4) \qquad = \Theta(n^3)$$

Telescoping (recursive code)

Example: Binary search

1. Find cases
2. Find general expression for $T(n)$
3. Find condition for base case $T(1)$

$$1. T(n) = \begin{cases} d & n = 1 \\ T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + c$$

$$2. \Rightarrow T(n) = T\left(\frac{n}{2^i}\right) + i \cdot c$$

$$T\left(\frac{n}{n}\right) = T(1) = d$$

$$3. \Rightarrow 2^i = n \qquad \Rightarrow i = \log(n)$$

$$\Rightarrow T(n) = d + \log(n) \cdot c = \Theta(\log(n))$$

6.2 Runtime of specific algorithms

6.2.1 List

Search element: see 6.2.3 Search Algorithms on Lists

Operation	Runtime	
Index Access	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(n)$	
Removal	$\mathcal{O}(n)$	
Remove last element	$\mathcal{O}(1)$	

6.2.2 Sorting Algorithms on Lists

Selection Sort

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

Insertion Sort

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

Bubble Sort

Case	Description	Runtime
Worst-case		
Average-case		
Best-case		

Mergesort

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

Quick Sort

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

Heap Sort

heapify(a) and HeapSort(a)

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

6.2.3 Search Algorithms on Lists

Linear Search

Case	Description	Runtime
Worst-case	b = a[n] is at end of array	$\Theta(n)$
Average-case	-	$\Theta(n)$
Best-case	b = a[0] at begining of array	$\Theta(1)$

Binary Search

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

6.2.4 Binary Search Tree

Search Node

Case	Description	Runtime
Worst-case	Tree is degenerated	$\Theta(n)$
Average-case	Tree is balanced	$\Theta(\log(n))$
Best-case	key is at root	$\Theta(1)$

Insert Node

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

Remove Node

Runtime is determined by the Runtime of 'Find symmetric successor'. 'h' is the height of the Tree

Case	Description	Runtime
All cases	-	$\mathcal{O}(h)$

Traversal

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

6.2.5 Heap

Insert Element

Case	Description	Runtime
Worst case	log(n) swaps	$\mathcal{O}(\log(n))$

Remove Element

Case	Description	Runtime
Worst case	log(n) swaps	$\mathcal{O}(\log(n))$

Heapify

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

Sort a heap

Used in 'Heapsort'. SiftDown traverses log(n) nodes and is called n times

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

6.2.6 Linked List

Operation	Runtime
Access	$\mathcal{O}(n)$
Search	$\mathcal{O}(n)$
Insertion	$\mathcal{O}(n)$
Insertion at head	$\mathcal{O}(1)$
Removal	$\mathcal{O}(n)$
Removal at head	$\mathcal{O}(1)$

6.2.7 Hash Table

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

7 Programming Concepts

7.1 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.

7.2 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.

7.3 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.

7.4 Functional programming

Pass functions as parameters to functions. Example:

```
numbers = [1, 2, 3, 4, 5]
def square(x):
    return x ** 2
squared_numbers = list(map(square, numbers))
#map function takes square function as a parameter
```

7.4.1 Lambda Expressions

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

```
lambda arguments : expression
#Lambda with one argument:
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:
y = lambda x,y: x*y
y(5,3) #15
```

7.4.2 Common Functions

Map

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

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

Filter

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

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

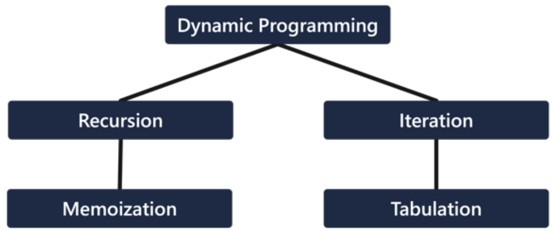
Reduce

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

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

8 Dynamic Programming (DP)

- "bottom-up" strategy
- iteratively solve smaller problems to solve progressively bigger problems
- Overlapping subproblems
- answers to smaller problems are stored in a table
- Example: Fibonacci



Convert code from recursive to dynamic programming:

- Look for repeating subproblems
- look for an optimal substructure to the solution – how does the answer of the problem depend on the answer of the subproblems?
- Should the answers of the subproblems be stored in a list? A table?
- Flip the recursive implementation around to implement a bottom-up, iterative solution.

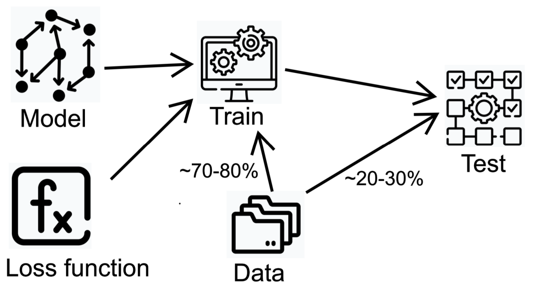
Example: dynamic programming approach to Fibonacci

```
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]
fib_n = F[n]
```

9 Machine Learning (ML)

create functions which map inputs to desired outputs by analysing underlying patterns

- Regression: find output value $\in \mathbb{R}$ based on input. Example: given a house has 5 bedrooms, 1000 square meters, what is its price?
- Classification: assign element to a group. Example: Based on color and texture, is a mushroom edible?




```
#read data
import pandas as pd
data = pd.read_csv('data.csv')

#split data into input and result set
y = data["diagnosis"]
X = data.drop(columns=["diagnosis"])

#split data into test and train set
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, random_state= 42) #30% of data used for validation

#choose and train model
from sklearn import linear_model
model = linear_model.LinearRegression()
model.fit(X_train,y_train)

#validate and test model
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)
score = accuracy_score(y_test, y_pred)

#CodeExpert grading based on y.final
X_final = pd.read_csv("X_final.csv")
y_final = model.predict(X_final)
return y_final
```

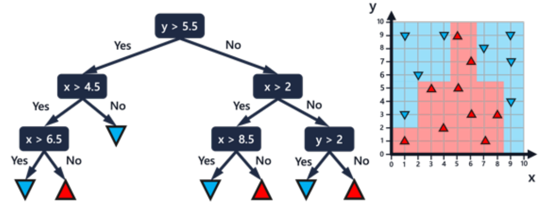
Loss function:

- $\frac{\#\text{ wrongly classified examples}}{\#\text{ total examples}}$
- Mean Square Error (MSE)
- L2 (Gauss Loss)

9.1 Models

9.1.1 Decision Tree Classifier

Constructs a decision tree to classify data based on features

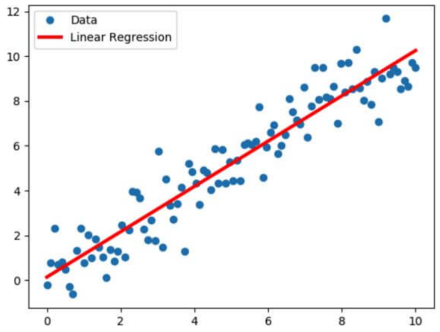



```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=4) #max
    height of tree
```

9.1.2 Linear

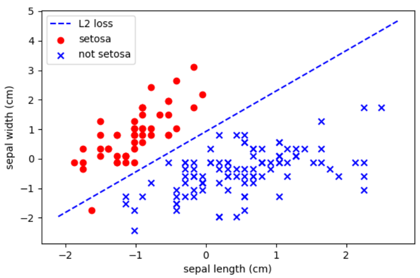
Regression

Constructs a linear model $y = \beta_0 + \sum_{i=0}^{n} \beta_i x_i$ so that the loss function is minimized



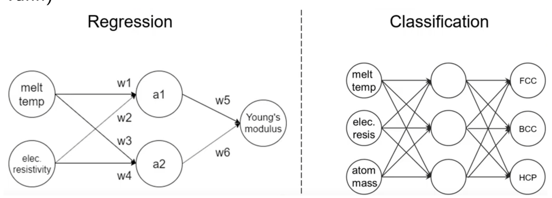

```
from sklearn import linear_model
model = linear_model.LinearRegression()
```

Classification



9.1.3 Neural Network

Constructs layers of neurons, where the output of each neuron is made non-linear through an activation function (Sigmoid, Relu, Tanh)



Regression

Usually no activation function in output layer

```
from sklearn.neural_network import MLPRegressor
#hidden_layer_sizes: array indicating number of
    neurons per layer
#activation: activation function
model = MLPRegressor(hidden_layer_sizes = [4,4],
    activation="relu")
```

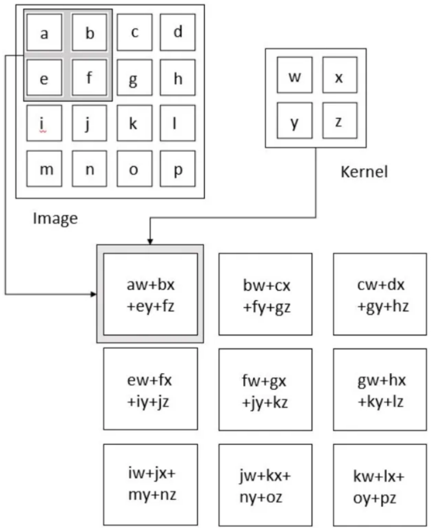
Classification

Usually Sigmoid in output layer

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(hidden_layer_sizes = [4,4],
    activation="logistic")
```

Convolutional Neural Network

Softmax can be used in output layer
Image processing, process to reduce data
Example using Joint Matrix (Faltungsmatrix):



One can also choose the max matrix. For the first field in the example this would yield the output: max(a, b, e, f)

9.2 Test and Validate

Accuracy score

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

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

R2 score

1 is the best score, the more negative the worse.

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

Mean Squared Error (MSE)

0 is the best score. The bigger the worse.

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

9.3 Over-/Underfitting and Cross Validation

- Underfitting: model is too simplistic to capture the underlying patterns in the data
- Overfitting: model memorizes random fluctuations in the data rather than capturing the underlying patterns

Result: No liable predictions from the model
Solution: Cross-Validation

1. subdivide data set into smaller parts
2. use n-1 parts for training and 1 remaining part for testing, trying out different settings

9.4 Encoding

Ordinal Encoding

Assign number to a string in a dataset:
red → one, green → 2, blue → 3

+ easy, efficient
− In our example, the 'distance' between "red" and "blue" might seem bigger than between "green" and "blue"

Mean Encoding

red	True
red	False
red	True
green	False
green	False

$$\text{mean}_{\text{red}} = \frac{\#(\text{red and True})}{\#\text{red}} = \frac{2}{3}$$

$$\text{mean}_{\text{green}} = 0$$

+ meaning behind the encoding (probability of occurence)
− ML Algorithm can access this probability → Overfitting

One-Hot Encoding

Color	On?
red	True
red	False
red	True
green	False
green	False

→

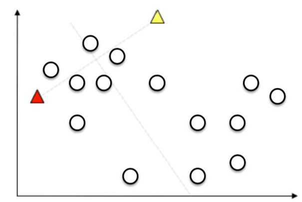
red	green	On?
1	0	True
1	0	False
1	0	True
0	1	False
0	1	False

+ Same distance between all properties
− Tables can become very big, leading to slower training

9.5 Clustering

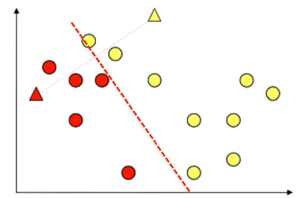
Given: $X = \{x_1, x_2, ..., x_n\} \in \mathbb{R}^d$

Initialize K centroids (randomly): $z_i \in \mathbb{R}^d \to Z = \{z_1, z_2, ..., z_K\}$



Assign each point to centroid it is closest to:

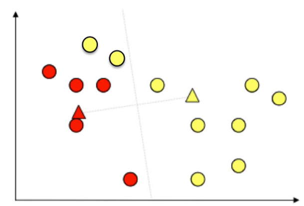
$c(x) = \text{argmin}||x - Z_i||$



Place centroid in the mean of its assigned points:

$$S_j = x_i \in z_j$$

$$z_j = \frac{1}{len(S_j)} \sum_{S_j} x$$



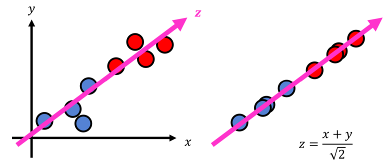
Repeat step 2 and 3 until the centroids have converged, that means no point is assigned to another centroid anymore
Overfitting: Each point has its own centroid
Loss function:

$$L = \left(\sum ||x_i - z_{x_i}||^2\right) + \lambda e^K$$

squared distance of point to its centroid + penalty for using to many centroids (prevents overfitting)

Principal Component Analysis (PCA)

Reduce dimension while keeping the maximum information: goal is to have the values as far apart from each other as possible after a linear projection on a line.



10 Numpy

Numpy is a Python package (equivalent to a C++ library) which supports operations with n-dimensional arrays and various computational methods.
Import the Numpy package:

```
import numpy as np
```

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

10.1 Python Lists vs. Numpy Arrays

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

Python 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

10.2 Declaring Numpy Arrays

Using sequences

```
a = np.array([1, 2, 3, 4]) #array([1,2,3,4])
b = np.array(range(2,10,3)) #array([2,5,8])
c = np.array([[1,2],[3,4]]) #comparable to matrix
```

Using random numbers

```
R = np.random.random(10) #10 random numbers in
    [0,1)
R = np.random.uniform(-1,1,5) #5 random numbers
    between -1 and 1
R = np.random.randint(1,7,10) #10 random integers
    between 1 and 6
```

Using arange command

creates an array from the start to the stop value with a given step value, stop is **not** inclusive

```
R = np.arange(start, stop, step)
R = np.arange(2,10,3) #array([2,5,8])
#the same output as np.array(range(2,10,3))
```

Using linspace command

creates a numpy array with num equally spaced elements between start and stop, stop is inclusive:

```
R = np.linspace(start,stop,num)
#Step size = (stop-start)/(num-1)
R = np.linspace(3, 2, 3) #array([3, 2.5, 2])
```

10.3 Numpy Array operations

Common numpy array operations

```
a = np.arange(10)
a.size() #=10, return number of elements
a[5] #=5, access element (1D array)
A = np.array([[1,2,3],[4,5,6]])
A[1,2] #=6, access element (2D array)
A[1][2] #=6, access element (2D array)
```

To access lines of a 2D array, see slicing.

Slicing

stop is not included

- 1D array: A[start : stop : step]

```
A = np.arange(10) #array([0,1,2,3,4,5,6,7,8,9])
A[2:5:2] #array([2,4])
```

- 2D array: $A[\underbrace{\text{start : stop}}_{\text{row}}, \underbrace{\text{start : stop}}_{\text{column}}]$

default values: start = 0, stop = len(A) if only one number is given, only the row / column corresponding to that index is taken

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
A[1,:] #array([4,5,6]) #row at index 1
A[:,2] #array([3,6,9]) #column at index 2
A[0:2,1:3] #array([[2,3],[5,6]]) # rows 0 and 1,
    columns 1 and 2
```

Statistics

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

Mathematical Operations

Most 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]])
C = np.array([[1, 4], [3, 4], [4,6]])
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) #sum of the columns
#= A.sum(axis = 0) -> array([8,10,10])
np.sum(A, axis = 1) #sum of the rows
#= A.sum(axis = 1) -> array([9,19])

A @ C #matrix multiplication
A.dot(C) #matrix multiplication
#array([ [27, 44] , [51, 88] ])

a = np.array([1,2,3])
b = np.array([3,4,6])
a.dot(b) #29, scalar product
```

Filtering

```
a = np.arange(7) #array([0,1,2,3,4,5,6])
f = a % 2 == 0 #f = [True, False, True, False, True
    , False, True]
a[f] #array([0,2,4,6])
```

11 Pandas

Pandas is a Python package which supports working with tabulated data.
To import pandas, use:

```
import pandas as pd
```

Now you can refer to classes and functions from the package using "pd".

11.1 read CSV file into dataframe

```
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.
```

11.2 Pandas Dataframe operations

A dataframe can be thought of as a list within a list supporting access in more meaningful ways compared to using indices.

Example dataframe

	Unnamed: 0	time	jan	feb
0	0	1864	-7.10	-4.52
1	1	1865	-3.47	-6.25
2	2	1866	-1.31	-0.42
3	3	1867	-3.87	0.56
4	4	1868	-5.46	-1.53
...	...	...	...	...
157	157	2021	-3.56	NaN

table of the climate, entries accessible via index. The left-most column is known as the "index column".

Change Index Column

time	Unnamed: 0	jan	feb
1864	0	-7.10	-4.52
1865	1	-3.47	-6.25
1866	2	-1.31	-0.42
1867	3	-3.87	0.56
1868	4	-5.46	-1.53
...	...	...	...
2021	157	-3.56	NaN

```
climate2 = climate.
    set_index("time")
    #creates copy
```

table of the climate, entries accessible via time

Rename Columns

```
climate = climate.rename(columns={"old_index_name":
    "new_index_name", ...})
#renames the "old_index_name" column to "
    new_index_name"
```

rename all columns (len(data.columns) = number of colums must be true)

```
data.columns = ["Date", "January", "February", ...]
```

Access Dataframe Elements

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

Filter 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]
```

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
```

Modify Dataframes

Add a column:

```
climate["new_col"] = climate["time"] + climate["jan
    "]
#"new_col" is a new column whose 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
```

Analyse Data

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.
```

12 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
```

Now you can refer to classes and functions from the package using "plt".

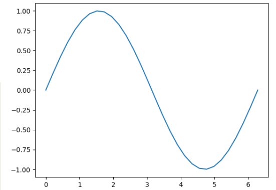
12.1 Line and Scatter Plots

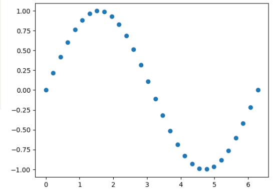
To graph two numpy arrays, one representing the x values and the other representing the corresponding y values:

```
import matplotlib.pyplot
    as plt
import numpy as np

X = np.linspace(0,2*np.
    pi,30)
Y = np.sin(X)

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

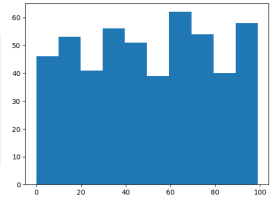




12.2 Histogram Plots

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()
```



12.3 Graph Styling

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