PVK Skript: Informatik II

Jérôme Schneider
jerome.schneider@inf.ethz.ch

Juni 2023

Vorwort

```
def main():
  print("hello, world")
```

An den Leser

Es freut mich sehr, dass du dieses Skript entdeckt hast. Hier findest du einen Überblick über den Stoff, der in der Vorlesung Informatik II vorgestellt wurde. Kombiniert mit Übung und eigener Repetition ist der Zweck dieses Skripts, dich auf die Prüfung vorzubereiten.

Dieses Skript ist *nicht* offizielles Material und wurde von mir als Begleitung zum PVK geschrieben. Ich gebe keinerlei Garantie über Vollständigkeit oder Richtigkeit der Inhalte dieses Skripts. Insbesondere wurde nicht jede Einzelheit, die in der Vorlesung besprochen wurde, in dieses Skript aufgenommen.

Solltest du einen Fehler oder eine Unklarheit in diesem Skript vorfinden, oder einen Verbesserungsvorschlag haben, so bitte ich um eine Mail an die Adresse, welche sich auf der Titelseite findet. Vielen Dank!

Ich wünsche dir eine gute Prüfungsvorbereitung und viel Glück an der Prüfung selbst!

Jérôme Schneider

Inhaltsverzeichnis

Teil I

Python Basics

Kapitel 1

The Language

Python erlaubt euch, sehr schnell Code zu schreiben, ohnedass ihr euch gross Gedanken über Typen und Pointer machen müsst. Sogar das Semikolon am Ende jeder Zeile fällt weg. Vergleicht etwa die folgenden Code-Snippets:

```cpp
// C++
int myNum = 15;
cout << myNum;
```

```python
# Python
myNum = 5
print(myNum)
```

Viele Sachen, die in `C++` dazu geführt hätten, dass euer Programm gar nicht erst kompiliert, wird in Python entweder gar kein Problem sein (wenn auch zum Teil mit *interessanten* Auswirkungen), oder aber ihr werdet zur Laufzeit einen Fehler sehen. Dadurch wird das gute alte "`print`-debugging" um einiges nützlicher!

In Python werdet ihr das Meiste, was ihr in `C++` verwendet habt, wiederfinden, etwa `if-else`, `for`, `while`, etc.

1.1 Containers: Lists and Dicts, Sets and Tuples

Wohl eines der zentralen Themen dieser Vorlesung, was Python angeht, sind sogenannte Container. Darunter verstehen wir eine Sammlung von Datenstrukturen, die "mehrere Elemente enthalten". Wir können diese weiterhin unterteilen in Sequenzen (welche geordnet sind) und Kollektionen (welche nicht geordnet sind). Das einfachste Beispiel von einer Sequenz ist die `list`, das einfachste Beispiel einer Kollektion ist das `set`.

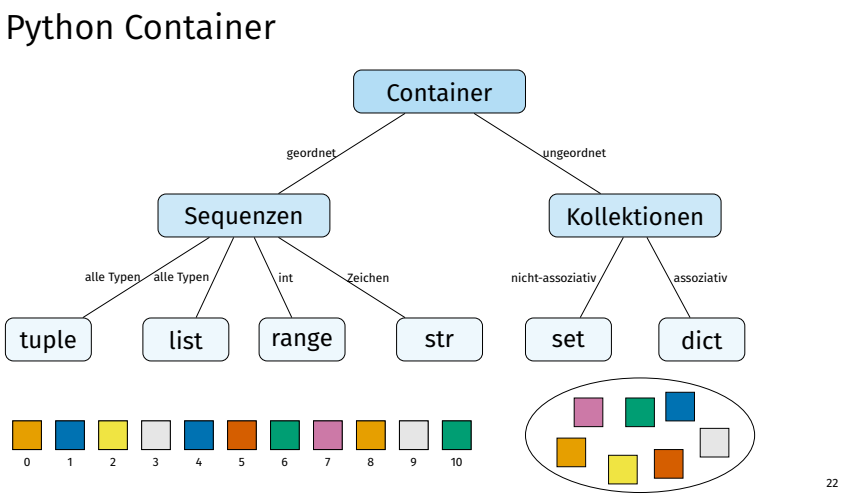


Abbildung 1.1: Eine Übersicht über Container in Python (lecture 1)

Python bietet euch einige Funktionen und Operationen an, welche ihr auf Container nutzen könnt, egal, ob es jetzt ein `set`, eine `list` oder ein anderer Container ist. Hier ein Beispiel der Operationen:

```
l = [2,3,5,7,11,13]
s = {2,3,5}

# Using len
print(len(l))
# 6
print(len(s))
# 3

# Using in
if 6 in l:
  print("6 is in the list!")
else:
  print("6 isn't in the list!")
# 6 isn't in the list!

# Iterating over container elements:
for x in l:
  print(x+1)
# 3 4 6 8 12 14
```

Die letzten Zeilen Code zeigen euch weiterhin, dass in Pyton for-loops in der Regel über Elemente iterieren, anstatt wie in C++ über Indizes:

```
// C++
int array[3] = {10,20,30}
for(int i = 0; i < sizeof(array); i++)
{
  cout << array[i]
}
```

```
# Python
array = [10,20,30]
for x in array:
  print(x)
```

Betrachten wir nun zuerste die einzelnen Sequenzen im Detail:

tuple sind, nun, Tupel. Wie die meisten Container können die einzelnen Elemente unterschiedliche Typen haben:

```
t = ('a', 0, 42.314)
```

String ist uns allen wohlbekannt. In Python sind Strings container, die als Elemente die einzelnen Zeichen haben:

```
s = "hello world"
for letter in s:
  print(letter + "1")
# h1 e1 l1 l1 o1  1 w1 o1 r1 l1 d1
```

List ist das Python-Analogon zu Arrays in C++. Analog zu C++ können wir auf einzelne Elemente via [] (subscript) Operator zugreifen (Achtung: 0-indexed!). Zusätzlich gibt es einige nette Kniffe:

```
l = [1, 15, 'hi', -3.14]

# Element am Ende hinzufuegen
l.append(17)
print(l)
# [1, 15, 'hi', -3.14, 17]

# Element loeschen
del l[1]
print(l)
# [1, 'hi', -3.14, 17]
```

```
# Liste umkehren
l.reverse()
print(l)
# [17, -3.14, 'hi', 1]

# Liste mit Anzahl von gleichen Werten erstellen
l = [6] * 2
print(l)
# [6, 6]
```

Ranges Stehen für einen Wertebereich. Wir vertiefen dies im nächsten Unterkapitel zusammen mit Slicing.

Und hier einige Operationen, welche auf allen *Sequenzen* funktioneren, also auf lists, tuples, ranges und strings:

```
l = [2,3,5,7,11,13]
# Enumeration
for idx, val in enumerate(l):
  print(idx,val)
  # 0 2  1 3  2 5  3 7  4 11  5 13

# Zugreifen via []
print(l[2])
# 5

# 'von hinten indexen':
print(l[-1])
# 13

sl = [5, 6, 7]
z = zip(l, sl)
print(list(z))
# [(2,5), (3,6), (5,7)]
```

Und nun zu den Kollektionen, namentlich Set und Dict:

Set ist eine (mathematische) Menge. Heisst, es gibt keine Duplikate und keine bestimmte Reihenfolge von Elementen:

```
s = {3,5,5,5}
print(s)
# {3, 5}
```

```
s.add(7)
s.remove(3)
print(s)
# {5,7}
```

Dict (oder zu Deutsch 'Wörterbuch') weist jeweils einem Wert (genannt key) einen anderen Wert (genannt value) zu. Nachschlagen funktioniert wiederum via subskript-Operator:

```
d = { "Enrolled" : True, "Money" : -5, "Motivation" : 0}
print(d["Motivation"])
# 0

# Adding to a dict
d["Attendance"] = "Always"
# Modifying
d["Money"] = -10

# Iterating
for k in d.keys():
  print(k)
# 'Enrolled' 'Money' 'Motivation' 'Attendance'
for v in d.values():
  print(v)
  # True -10 0 Always
for k, v in d.items():
  print(k,v)
  # Enrolled True  Money -10  Motivation 0  Attendance Always
```

Wenn ihr aus irgendeinem Grund einen bestimmten Datentypen haben wollt - Container oder sonstwas - (zum Beispiel beschwert sich Python darüber, dass ihr zwei unterschiedliche Typen zusammenadiert o.Ä.), könnt ihr einfach den namen des Datentypes als Funktion verwenden, um den Typen anzupassen:

```
num = "5"
# let's make that an int
int_num = int(num)

l = [1,2,3,4]
# let's make this a tuple
tup_l = tuple(l)
```

1.2 Ranges und Slicing

Ranges und Slices sind eng verwandt, was Syntax angeht. In beiden Fällen wollen wir einen Wertebereich definieren, Slicing ist dabei die direkte Anwendung von diesem Wertebereich als indizieren einer Liste. Ranges werden häufiger für loops gebraucht.

```
# Ranges:
# From 0 up to and including stop-1, in stepsize 1:
range(stop)
# i.e.
range(5)
# 0, 1, 2, 3, 4

# From start up to and including stop-1, in stepsize 1:
range(start, stop)
# i.e.
range(3,5)
# 3, 4

# With modified stepsize:
range(start, stop, stepsize)
#i.e.
range(1,5,2)
# 1, 3

# Negative stepsize supported!
range(5,1,-1)
# 5,4,3,2

# Need to convert to pretty print
print(range(5,1,-1))
# range(5,1,-1)
print(list(range(5,1,-1))
# [5,4,3,2]

# Typical usecase
for i in range(10)
  ...
# Executes the thing 10 times
```

Slices funktionieren ganz ähnlich. Im Gegensatz zu Ranges funktioneren Slices aber nur auf Sequenzen, und wir können beliebige Argumente auslassen. Spielt ein wenig damit herum, um ein Gefühl für das Verhalten von Ranges zu bekommen:

```
s = list(range(5))
# Leads to s = [0,1,2,3,4]

s[start:stop:step]
s[start:stop] # step = 1
s[:stop:step] # start = 0 (for step > 0, otherwise start = len(s)-1 )
s[start::step] # stop = len(s) (for step > 0)
s[::step] # From 0 to len(s) or from len(s)-1 to (and including!) 0
# You can remember it like this:
# If you don't give a start, it'll start at the earliest possible.
# If you don't give a stop, it'll iterate through to the end.
# If you don't give a stepsize, it'll be 1
```

1.3 List comprehension

List comprehension und die eng verwandte Dict comprehension erlauben es euch, einen beliebigen Container zu filtern als auch eine Funktion auf alle Elemente dieses Containers anzuwenden, und das Resultat als Liste bzw. als Dict zu speichern.

```
# General syntax:
# f is an arbitrary function
# c is a container
# b is a function returning a boolean
[f(elem) for elem in c if b(elem)]

# Example without filter
f = [x**2 for x in range(5)]
print(f)
# [0, 1, 4, 9, 16]

# Filter without function
c = [9,11,12,13,14,15,17]
f = [el for el in c if el > 10 and el <15]
print(f)
# [11, 12, 13, 14]
```

Mit einer sehr ähnlichen Syntax könnt ihr auch Sets und Dicts erstellen. Falls ihr als Quelle für eure Dicts mehrere Listen wollt, ist `zip` euer Freund.

```
# f,g are arbitrary functions
# Set
{f(elem) for elem in c if b(elem)}
```

```python
# Dict:
{f(elem):g(elem) for elem in c if b(elem)}

# Simple dict mapping number to its square
sqr = {x:x**2 for x in range(5)}
print(sqr)
#{0:0, 1:1, 2:4, 3:9, 4:16}

# 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}
```

1.4 Classes and magic methods

Klassen dienen vor allem dazu, zusammengehörige Daten zusammenzufassen und Operationen darauf bequem zugänglich zu machen. Instanzen von Klassen werden Objekte genannt.

```python
class ETH_Lecture:
  credits = 0
  name = ''
  description = ''

  def advertise(self, times):
    for _ in range(times):
      print(self.name +
      'is a cool lecture and gives you' +
      str(self.credits) +
      'credits!')

cs2 = ETH_Lecture()
cs2.credits = 4
cs2.advertise(1)
#   is a cool lecture and gives you4credits!
print(cs2)
# <__main__.ETH_Lecture object at 0x7efdf9e97f50>
print(sorted([cs2,cs2,cs2]))
# TypeError: '<' not supported between instances of 'ETH_Lecture' and 'ETH_Lecture'
```

Wie das code-Beispiel demonstriert, gibt es noch einige Dinge, die wir verbessern können. Dazu nutzen wir sogenannte "magic methods" 🪄✨. Diese beginnen und enden mit einem doppelten Unterstrich. Am interessantesten sind vermut-

lich `__init__` und `__str__`, aber auch Vergleichsoperatoren wite `__lt__` oder Operatoren wie `__add__` können hilfreich sein.

```
class ETH_Lecture:
  credits = 0
  name = ''
  description = ''

  def __init__(self, credits, name, descr):
    self.credits = credits
    self.name = name
    self.description = descr

  def __str__(self):
  # Linebroken for clarity
    return "Course " + \
    self.name + \
    ", giving " + \
    str(self.credits) + \
    " credits."

  def __lt__(self, other):
    return self.credits < other.credits

cs2 = ETH_Lecture(4, "CS2", "A course teaching CS. Part2.")
print(cs2)
# Course CS2, giving 5 credits.
print(cs2 < cs2)
# False
```

Operation	**Meaning**	**Magic 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__
-	Subtraction	__sub__
*	Multiplication	__mul__
**	Exponentiation	__pow__
/	Division	__div__
%	Modulo	__modulo__
print()	printing	__str__
Classname()	Konstruktor	__init__

1.4.1 Inheritance

Oft haben Klassen Ähnlichkeiten oder verwenden gleichen Code. In Python können wir mit Inheritance (oder zu Deutsch, Vererbung) gut "ist-ein" Beziehungen abdecken. Im Beispiel unten erstellen wir eine Klasse `Animal`, und eine Klasse `Dog`. Da jeder Hund ein Tier ist, wollen wir, dass die `Dog` Klasse von der `Animal` Klasse erbt. Wir erreichen dass, indem wir den Namen der Klasse, von der wir erben wollen, in Klammern nach dem eigenen Klassennamen schreiben:

```
class Animal:
  def info(self):
    print("This is an animal")

  def speak(self):
    print("This animal makes some sound.")

class Dog(Animal):
  def speak(self):
    print("Bark!")

  def doDogThings(self):
    print("The dog licks you")
```

Theoretisch könnten wir diese Vererbungskette weiter und weiter spinnen, also etwa eine Klasse `Labrador` erstellen, welche von `Dog` erbt, und so weiter und so fort.

Eine Klasse erbt von ihrer Elter-Klasse alle Attribute und Methoden. Wir können diese aber überschreiben (wie oben die `speak`-Methode), oder neue Attribute und Methoden (`doDogThings`) hinzufügen. Die so erstellten Objekte verhalten sich so, wie wir dies erwarten würden. Siehe das folgende code-Beispiel:

```
a = Animal()
d = Dog()

a.speak()
# This animal makes some sound
d.speak()
# Bark!
d.info()
# This is an animal
d.doDogThings()
# The dog licks you
a.doDogThings()
# AttributeError: 'Animal' object has no attribute 'doDogStuff'
```

Kapitel 2

Python Libraries

In der Vorlesung habt ihr drei verschiedene libraries/packages kennengelernt: `numpy`, `pandas` und `matplotlib`. In diesem Skript beschreiben wir nur die ersten zwei, da `matplotlib` kaum Prüfungsrelevanz besitzt.

Der Einfachheit halber nehmen wir an, dass in diesem Kapitel der folgende Code immer zuerst ausgeführt wurde:

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

2.1 Numpy

"`numpy` ist das fundamentale Paket für das wissenschaftliche Rechnen mit Python. Die Bibliothek bietet mehrdimensionale Arrays und viele Operationen darauf an."

Für uns von Relevanz ist genau das, was diese kurze Beschreibung erwähnt: (mehrdimensionale) Arrays und Operationen darauf.

Im Gegensatz zu Python's Listen sind numpy's Arrays von *fixer Grösse*, sie erlauben nur *einen Typ* pro Array, und können *mehrdimensional* sein (anstatt wie in Python einfach verschachtelt).

Wir können Arrays entweder aus Listen erstellen, oder mit einigen Funktionen von `numpy` kreieren:

```
# Listen zu Arrays
l = [1,2,3,4]
a = np.array(l)
b = np.array(range(2,10,3))
```

```
c = np.array([[1,2], [3,4]])

# random arrays
rand = np.random.random(10)
# 10 random numbers in [0,1)
rand = np.random.uniform(-1, 4, 5)
# 5 random numbers in [-1,4)
rand = np.random.randint(1,7,10)
# zehn random integers between 1 and 6 inclusive

# works just like np.array(range(start,stop,step))
np.arange(start, stop, step)
np.arange(start, stop) # step = 1
np.arange(stop) # start = 0, step = 1

# aequidistante Intervalle, i.e.
# num punkte auf dem Intervall start, stop
np.linspace(start, stop, num)
np.linspace(start, stop) # num = 50
```

Operationen und Zugriffe auf diesen Arrays funktionieren ähnlich wie mit Listen, wenn auch das zwei- und höherdimensionale etwas einfacher wird:

```
a = np.arange(2,6) # = [2,3,4,5]
len(a) # = 4
a.size # = 4

A = np.array([[1,2,3], [4,5,6]])
len(A) # = 2, Anzahl Zeilen
len(A[0]) # = 3, Anzahl Spalten
A.size # = 6, Anzahl Elemente

# Zugriff
a[1] # = 3
a[-1] # = 5

A[1,2] # = A[1][2] = 6

# Slicing
A = np.array([[1,2,3,4], [5,6,7,8], [9,3,2,1]])
A[1] # = A[1, :] = [5,6,7,8] (Zeile)
A[:, 2] # = [3,7,1] (Spalte)
A[1:2, 1:3] # = [[6,7]]
A[-2:-1, ::2] # = A[1:2, 0:4:2] = [[5,7]]
```

Operationen auf Arrays sind einfach und äusserst nützlich:

```
a = np.linspace(-4, -2, 3) # = [-4, -3, -2]
a.min() # = -4
a.sum() # = -9
a.max() # = -2

np.mean(a) # = b.sum() / b.size = -3
np.std(a) # = standard deviation = 0.816
np.cumsum(a) # Teilsummen: [-4, -7, -9]

# Elementweise Operationen
a + 1 # = [-3, -2, -1]
a * 2 # = [-8, -6, -4]
a ** 2 # = [16, 9 , 4]
np.sin(a) # elementweise sinus

# Mehrere Arrays
b = np.arange(3) # = [0,1,2]
a + b # = [-4, -2, 0]
a * b # = [-0, -3, -4]
```

Ihr könnt `numpy` Arrays auch filtern. Dazu könnt ihr euer Array als Input für eine Funktion sehen, welche True oder False zurückgibt. Wenn ihr dann an "dieser Stelle" das Array aufruft, erhaltet ihr das gefilterte Array. Schauen wir uns ein Beispiel an:

```
a = np.arrange(5) # = [0,1,2,3,4]
fil = a % 2 == 0 # f = [True, False, True, False, True]
x = [True, False, True, False, True]
a[fil] # = a[x] = [0,2,4]
```

2.2 Pandas

"pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language."

Wir werden uns vor allem darauf fokussieren, CSV (Comma Separated Value) files einzulesen, und dann auf gewisse Zeilen/Spalten zuzugreifen beziehungsweise diese aus der Tabelle zu löschen.

Pandas bietet noch viele weitere Funktionen, aber wir beschränken uns hier aufs Wesentliche.

```
df = pd.read_csv("data.csv", sep="\t")
```

```python
# sep by default is ',' so no need to specify for most files

# Create new dataframes that do not have the target column...
X = df.drop(['target'], axis=1)
# ... and one that has only the target column
y = df['target']

# In general, access rows by index, columns by name
row = df.iloc[3]
rows = df[1:4]
col = df["somecol"]
cols = df[["somecol", "othercol"]]

# Filter, analog zu Numpy:
df["Money"] > 0 # = [True, False, ...]
df[df["Money"] > 0] # = Alle Zeilen, in denen Money > 0 ist
df.sum() # Summiert jede Spalte auf
df.max() # Maximum jeder Spalte
```

Teil II

Algorithms

Kapitel 3

Mathematische Grundlagen, Big-O, Snippets

Algorithmen sind ganz allgemein, eine (endliche) Beschreibung des Vorgehens zum Lösen eines Problems. Diese Beschreibung soll, für unsere Zwecke, für einen Computer verständlich sein - "Sortiere die Liste" ist also für uns zu ungenau.

Bevor wir uns mit dem Design von Algorithmen beschäftigen, repetieren wir, wie wir die (Laufzeit-)Effizienz von Algorithmen klassifizieren. Dazu ist wichtig:

- Wir betrachten immer den *worst-case*. Dieser variiert von Algorithmus zu Algorithmus, selbst für das gleiche Problem!
- Wir klassifizieren Effizienz in Abhängigkeit von der Eingabegrösse n
- Wir vernachlässigen konstante additive und multiplikative Faktoren. Also ist $3 \cdot n + 4$ "gleich gut" wie $\frac{1}{42}n - 5$

Um diese Laufzeiten anzugeben, verwenden wir die Landau oder auch Big-O genannte Notation. Sei im Folgenden g die Funktion, welche die Laufzeit unseres Algorithmus angibt:

- Wächst g *nicht schneller* als f, so ist $g \in \mathcal{O}(f)$, oder der Einfachheit halber $g = \mathcal{O}(f)$
- Wächst g *nicht langsamer* als f, so ist $g \in \Omega(f)$ oder $g = \Omega(f)$
- Wächst g genau gleich schnell wie f (also nicht langsamer und nicht schneller), so ist $g \in \Theta(f)$.

Für die Interessiserten hier noch die mathematischen Definitionen. Diese sind nicht essentiell, es reicht, wenn ihr wisst, welcher Buchstabe für welche Art von

Schranke steht.

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

Eine sehr beliebt Prüfungsaufgabe ist das Sortieren von asymptotischen Laufzeiten. Damit ihr diese schnell und richtig lösen könnt, folgen die wichtigsten Laufzeiten im Verhältnis und einige wichtige Formeln:

Sammlung von Funktionen

$$\begin{aligned}\mathcal{O}(1) \leq \mathcal{O}(\log n) \leq \mathcal{O}(\sqrt{n}) \leq \mathcal{O}(n) \\ \leq \mathcal{O}(n \log(n)) \leq \mathcal{O}(n^2) \leq \mathcal{O}(2^n) \leq \mathcal{O}(3^n) \leq \mathcal{O}(n!) \leq \mathcal{O}(n^n)\end{aligned}$$

Natürlich gibt es noch mehr Funktionen (n^3 etc.), aber diese finden ihren Platz relativ einfach.

Hier sind zwei Formeln, die ihr bei Asymptotik-Aufgaben immer im Kopf haben solltet:

$$\sum_{i=0}^{n} i^a = \Theta(n^{a+1})$$
$$\log(n^b) = b \cdot \log(n)$$

Als Beispiel für die erste Formel: $\sum_{i=0}^{n} 1 = \Theta(n)$ da in diesem Falle $a = 0$ ist, oder $\sum_{i=0}^{n} i = \Theta(n^2)$. Für die zweite Formel zum Beispiel: $\log(n^n) = n \log n$

3.1 Code snippets to runtime: Sums and Telescoping

Neben "sortiere-nach-Laufzeit" ist auch "Laufzeit-dieses-Snippets". Eine sehr beliebte Prüfungsaufgabe.

Die Aufgabe ist dabei immer die selbe, auch wenn die Namen etwas ändern können:

"Geben Sie jeweils die asymptotische Anzahl von Aufrufen der Funktion `f()` in Abhängigkeit von $n \in \mathbb{N}$ mit Θ-Notation möglichst knapp an."

Für diese Art von Aufgabe gibt es drei "Tricks", die dabei helfen können, diese Aufgaben zu lösen:

3.1.1 For-loops als Summen

Die Idee dieses Tricks ist es, dass wir (verschachtelte) for-loops einfach als Summen schreiben, und diese dann berechnen. Hier ist die erste Formel der vorherigen Seite sehr hilfreich. Zum Beispiel:

```
def g(n):
  for m in range(0, n):
    for k in range(0, n):
      f()
```

Ist äquivalent[1] zu

$$\sum_{m=0}^{n}\sum_{k=0}^{n} 1 = \sum_{m=0}^{n} n = \Theta(n^2)$$

Nützlich wird dieser Trick, wenn die innere Summe/for-loop etwas komplizierter wird:

```
def g(n):
for m in range(0,n):
  for k in range(0,m):
    f();
```

$$\sum_{m=0}^{n}\sum_{k=0}^{m} 1 = \sum_{m=0}^{n} m = \Theta(n^2)$$

Oder

```
def g(n):
  for i in range(0,n):
    for j in range(0,i*i):
      f()
```

$$\sum_{i=0}^{n}\sum_{j=0}^{i^2} 1 = \sum_{i=0}^{n} i^2 = \Theta(n^3)$$

3.1.2 While-Counters

Ein Tipp für die typischen `while`-Aufgaben ist es, sich eine Counter-Variable vorzustellen. Manchmal sind diese sogar im Snippet schon drin, etwa im Folgenden:

[1] mit Ausnahme, dass wir als Grenze n statt dem korrekten n-1 wählen. Der Einfachheit halber :)

```
def g(n):
  i = 1
  while i*i <= n:
    i = i + 1
    f()
```

Bemerkt, wie die Variable `i` genau die Anzahl Aufrufe von `f`(abzüglich 1) speichert. Wir können also uns einfach die Loop-Condition anschauen: Der Loop endet, wenn `i * i <= n` - wir können dies einfach nach `i` umformen, um herauszufinden, wie oft `f` aufgerufen wurde: $i^2 = n \implies i = \sqrt{n}$.

Was, wenn dieser Trick nicht direkt anwendbar ist?

```
def g(n):
  i = 1
  while i <= n:
    f()
    i = i * 4
```

Nun denken wir uns eine counter-variable dazu, das Snippet sieht also so aus:

```
def g(n):
  i = 1
  c = 0
  while i <= n:
    f()
    c = c + 1
    i = i * 4
```

Nun können wir zwar nicht direkt `c` und `n` vergleichen, wohl aber `i` und `c`: Wir erkennen, dass $i = 4^c$. Nun setzen wir dies in die Loop-Condition ein: $i = n \implies 4^c = n \implies c = \log_4 n = \Theta(\log n)$

3.1.3 Teleskopieren

Die Idee des Teleskopieren ist es, sich die Laufzeit eines rekursiven Snippets "Schritt-für-Schritt" aufzuschreiben, bis man eine Struktur darin erkennt, die man dann nutzen kann, um eine geschlossene Form aufzuschreiben (und halt dann umformen etc.) Schauen wir uns ein Beispiel an:

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

Sei also nun die Anzahl Aufrufe für `g(n)` gegeben als $T(n)$. Dann wissen wir

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

Da wir einen Aufruf von `f` haben und einen rekursiven Aufruf, und zwar mit `n-3`. Nun machen wir das Gleiche mit `T(n-3)`

$$\begin{aligned} T(n) &= 1 + T(n-3) \\ T(n) &= 1 + 1 + T(n-6) \\ &\dots \end{aligned}$$

Jetz können wir schon ein Muster erkennen:

$$\begin{aligned} T(n) &= 1 + T(n-3) \\ T(n) &= 1 + 1 + T(n-6) \\ &\dots \\ T(n) &= 1 + 1 + \dots + T(n - 3 \cdot k) \end{aligned}$$

Dies endet, wenn $k = n/3$, und wir sehen weiterhin, dass es genau k viele Einsen in diesem Term hat. Also muss die Anzahl Aufrufe $k = n/3 = \Theta(n)$ sein.

Ein weiteres Beispiel:

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

Und nun das Teleskopieren:

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ T(n) &= 1 + 1 + T(n/4) \\ &\dots \\ T(n) &= 1 + 1 + \dots + T(n/2^k) \end{aligned}$$

Auch hier: Diese Rekursion ended wenn $k = \log n$, wir haben wieder k Einsen, also ist die Anzahl Aufrufe $k = \log n = \Theta(\log n)$.

Ein letztes Beispiel, wo wir nicht einfach die Einsen zählen können:

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

Teleskopieren:

$$\begin{aligned} T(n) &= n + T(n/2) \\ T(n) &= n + n/2 + T(n/4) \\ &\cdots \\ T(n) &= n + n/2 + \cdots + T(n/2^k) \end{aligned}$$

Wir sehen hier, dass die Anzahl Aufrufe nicht in jeder Iteration gleich ist. Der Vorgang ist aber genau der Gleiche. Beenden wir nun diesen Versuch auf dem gleichem Wege: Die Rekursion endet, sobald $k = \log n$ ist. Betrachten wir nun die Summe, denn nur zu wissen, wie viele Terme da sind, hilft uns nicht besonders:

$$n + n/2 + \cdots + T(n/2^k) = n \cdot \sum_{i=0}^{k} \frac{1}{2^i}$$

Es ist bekannt, dass $sum_{i=0}^{\infty} \frac{1}{2^i} = 2$ ist. Da unser Term nicht ganz bis nach unendlich geht, können wir dies als obere Schranke sehen, also erhalten wir $n \cdot 2 = \Theta(n)$.

3.1.4 A note of caution

Manchmal ist ein Snippet keiner der obigen Kategorien zuzuordnen. Dann hilft es oft, sich seiner Intuition zu bedienen: Was “macht” das Code-Snippet? Gibt es ein Quadrieren, ein Halbieren? Oft ist die erste Abschätzung bei solch komischen Snippets schon nahe am Resultat. Im schlimmsten Fall ratet ihr halt, in der Regel könnt ihr einen 50:50 Versuch abgeben. Keine schlechten Chancen, würde ich sagen.

Kapitel 4

Sorting and Searching

4.1 Searching

Es gibt zwei grundsätzliche Wege, in einem Array zu suchen:

- Ist das Array *nicht sortiert*, so nutzen wir lineare Suche
- Ist das Array *sortiert*, so nutzen wir binäre Suche

Gibt es "genug" Suchanfragen, kann es sich lohnen, das Array zu sortieren, um alle zukünftigen Suchen schneller zu machen.

Lineare Suche ist relativ leicht: Wir vergleichen jedes Element im Array mit dem gesuchten Array. Finden wir eine Übereinstimmung, so geben wir den Index zurück.

Algorithm 1: Linear-Search(A,b)

```
1 for i ← 0, 1, ..., n − 1 do
2     if A[i]=b then
3         return i
4 return Not in array!
```

Die Laufzeit von linearer Suche ist klar $\mathcal{O}(n)$, da wir im schlimmsten Fall alle Elemente anschauen müssen.

Binäre Suche ist etwas komplizierter. Die Idee ist, jeweils die Problemgrösse zu halbieren. Das klappt wie folgt: Wir überprüfen das Element in der Mitte des Arrays. Ist das gesuchte Element grösser, so wiederholen wir diesen Schritt in der rechten Teilhälfte. Ist das gesuchte Element kleiner, in der linken Teilhälfte.

Algorithm 2: Iterative Binary-Search(A,b)

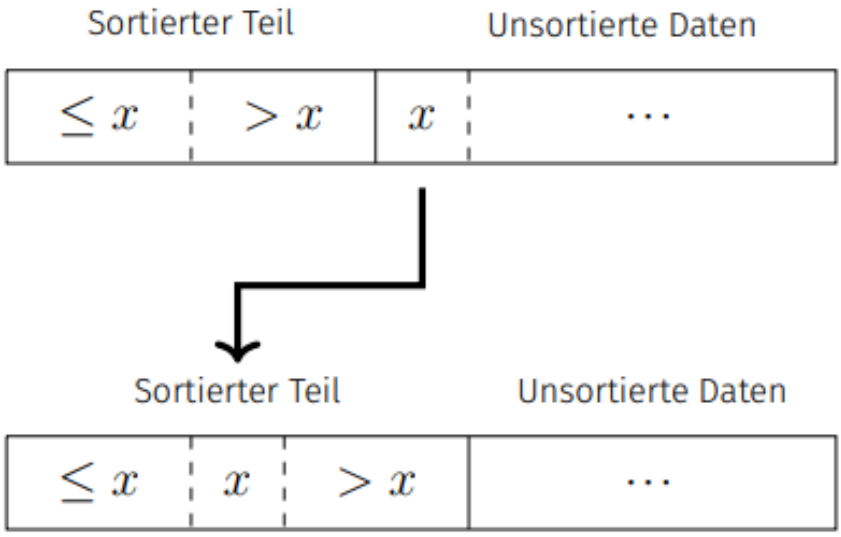
```
1  left ← 0
2  right ← n − 1
3  while left ≤ right do
4      mid ← ⌊(left + right)/2⌋
5      if A[mid] = b then
6          return mid
7      else if A[mid] > b then
8          right ← mid − 1
9      else
10         left ← mid + 1
11 return Not in array!
```

Durch dieses sukzessive Halbieren erreichen wir eine gute Laufzeit von $\mathcal{O}(\log n)$.

4.2 Sorting

4.2.1 Insertion Sort

Die Idee von Insertion Sort / Sortieren durch Einfügen ist es, einen sortieren Teil am Anfang des Arrays zu haben, und dann nach und nach Elemente an der richtigen Stelle in diesen sortierten Teil einzufügen.



Algorithm 3: Insertionsort(A)

```
1 for i ← 1, ..., n do
2     Index k ← Use binary search to find spot for A[i] in A[0,...,i-1]
3     b ← A[i]                                          ▷ Temporary variable
4     for j ← i, i − 1, ..., k do                       ▷ Move elements to the right
5         A[j] ← A[j − 1]
6     A[k] ← b
```

Die Laufzeit dieses Algorithmus ist $\mathcal{O}(n^2)$, da wir im schlechtesten Fall das Element immer an erster Stelle einfügen müssen, und damit immer die ganze Liste "nach hinten" schieben müssen. Das dauert für das zweite Element 1 Schritt, für das dritte 2, ..., und für das letzte Element $n-1$ Schritte, was in der Summe also $\mathcal{O}(n^2)$ macht.

4.2.2 Selection Sort

Die Idee von Selection Sort ist ähnlich, aber anstatt dass wir den richtigen Platz des nächsten Elements bestimmen, suchen wir stattdessen das Minimum aus dem unsortierten Teil:

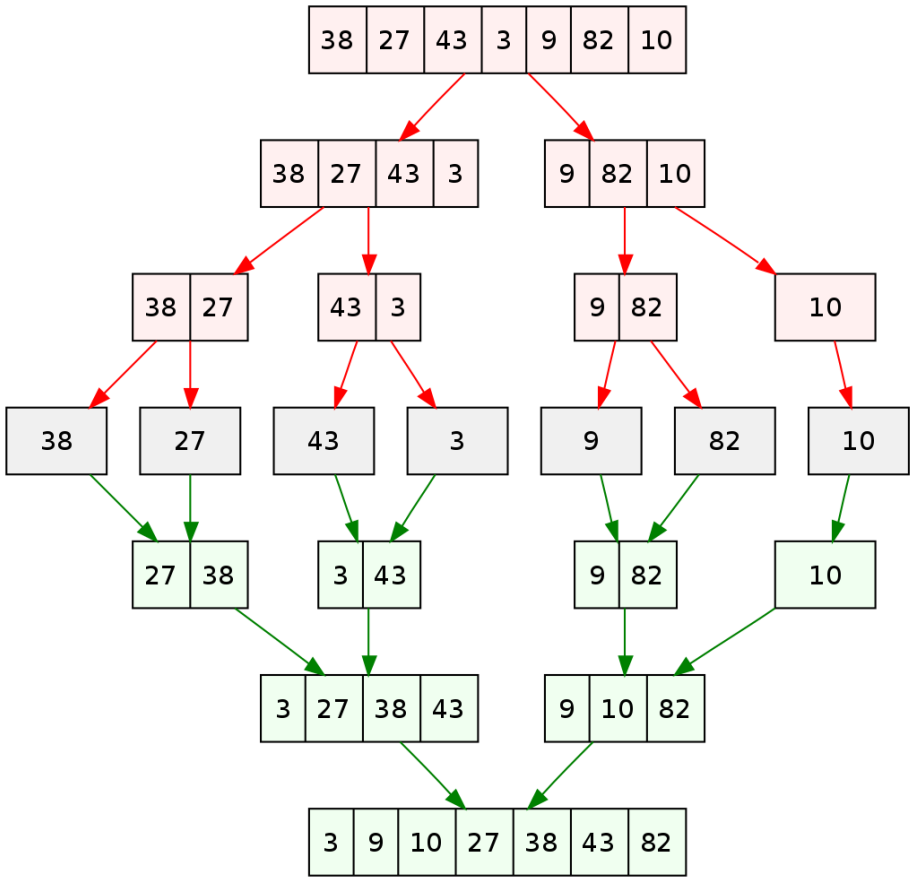
Algorithm 4: SelectionSort(A)

```
1 for i ← 0, ..., n − 1 do
2     j ← Find Index of minimum in A[i, ..., n]
3     Swap A[i] and A[j]
```

Das Minimum zu finden dauer im schlimmsten Fall für das erste Element n Schritte, für das zweite $n-1$ Schritte,..., für das letzte Element 1 Schritt. Wir erreichen also wiederum eine Laufzeit von $\mathcal{O}(n^2)$.

4.2.3 Merge Sort

Merge Sort / Sortieren durch Verschmelzen ist ein (im Vergleich zu den bisherigen Algorithmen) ziemlich komplizierter Algorithmus. Wir verwenden das Prinzip von Divide-and-Conquer, um das Problem in kleiner Teile aufzuteilen, diese beide zu lösen und die Lösung durch Verschmelzen zu finden:



In Pythoncode sieht das Ganze ungefähr so aus:

```
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)
```

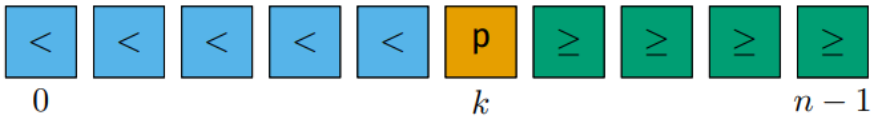
```
def merge(a1, a2):
  b, i, j = [], 0, 0
    while i < len(a1) and j < len(a2):
      if a1[i] < a2[j]:
        b.append(a1[i])
      i += 1
    else :
      b.append(a2[j])
      j += 1
  b += a1[i:]
  b += a2[j:]
  return b
```

Die Laufzeit von Merge Sort lässt sich wieder durch Teleskopieren bestimmen, und zwar mit $T(n) = 2T(n/2) + an + b$, wobei a ein Faktor ist, der die Dauer von MERGE beschreibt (MERGE ist klar in linearer Zeit), und b eine Konstante für "bookeeping" Operationen. Rechnen wir dies aus erhalten wir $\mathcal{O}(n \log n)$ als Laufzeit - eine deutliche Verbesserung also!

4.2.4 Quick Sort

Obwohl Merge-Sort schnell ist, benötigen wir zusätzlichen Speicherplatz für das Verschmelzen. Quicksort versucht dieses Problem zu beheben.

Wir wählen ein zufälliges Element aus dem Array aus, das ist unser sogenanntes *Pivot*. Dann teilen wir den Rest des Arrays in einen Teil mit allen Elementen kleiner und einen Teil mit allen Elementen grösser als das Pivot. Dann rekurrieren wir auf beiden Teilen.



In Python-Code:

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

```
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]
  # j is now the index of the Pivot
  return j
```

Die Laufzeit von Quicksort ist im *schlechtesten Fall* $\mathcal{O}(n^2)$: Wählen wir immer das Minimum als Pivot, so machen wir eigentlich nur Selection Sort.

Da die Wahl des Pivot zufällig erfolgt, so können wir Aussagen *in Erwartung* treffen - so etwas wie der "durchschnittliche" Fall. Quicksort ist *in Erwartung* sogar $\mathcal{O}(n \log n)$ - nicht aber im Worst Case!

4.2.5 Heap Sort

Heaps sind Teil des nächsten Kapitels - an dieser Stelle sei aber erwähnt, dass wir mit Heaps ganz leicht sortieren können: Wir fügen einfach jedes Element des Arrays in einen Min-Heap ein, dann entfernen wir immer das Minimum, und tada! - eine sortierte Liste. Heap-Sort dauert aufgrund der Laufzeiten der Heap-Operationen $\mathcal{O}(n \log n)$ - also gleich gut wie Merge-Sort!

Algorithm 5: HeapSort(A)

```
1 H ← CreateMinHeap()
2 R ← []
3 for i ← 0, ..., n − 1 do
4 |   H.insert(A[i])
5 for i ← 0, ..., n − 1 do
6 |   v ← H.removeMin()
7 |   R.append(v)
8 return R
```

Kapitel 5

Datastructures

5.1 Binäre Suchbäume

Im Rahmen dieser Vorlesung interessieren wir uns für eine besondere Art von Baum: Binären Suchbäumen.

In einem binären Baum speichert jeder Knoten einen Wert, und hat bis zu zwei Kinder, ein linkes und/oder ein rechtes.

Ein binärer *Suchbaum* (binary search tree (BST)) ist ein binärer Baum, in dem zusätzlich gilt:

- Jeder Knoten `v` speichert einen Wert (hier `v.key` genannt)
- Alle Schlüssel im Teilbaum `v.left` sind kleiner als `v.key`
- Alle Schlüssel im Teilbaum `v.right` sind grösser als `v.key`

Warum interessieren uns Bäume? Die Laufzeit von fast allen Operationen auf Bäumen ist abhängig von ihrere Höhe - und in gut balancierten Bäumen ist diese $\mathcal{O}(\log n)$. Ist unser Baum hingegen schlecht balanciert ist diese $\mathcal{O}(n)$, und wir sind gleich gut wie eine gewöhnliche Liste.

Betrachten wir nun die elementaren Operationen auf Suchbäumen:

5.1.1 Suchen

Suchen in einem BST ist analog zu binary search: Wir vergleichen unseren gesuchten Wert mit dem Wert der im momentanen Knoten gespeichert ist, und gehen dann nach rechts bzw. links je nachdem ob unser Element kleiner bzw. grösser war. Wir wiederholen dies solange bis wir entweder einen Knoten gefunden haben, der den gesuchten Wert speichert, oder bis wir das Ende des Baums erreicht haben.

In Python:

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

5.1.2 Einfügen

Einfügen ist sehr einfach: Wir suchen nach dem Element, falls es schon existiert machen wir nichts, und sonst fügen wir es an der richtigen Stelle ein. Deswegen sieht der Pythoncode auch sehr ähnlich aus:

```
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
```

5.1.3 Entfernen

Entfernen ist die umständlichste Operation, es gibt drei Varianten:

- Der zu entferndende Knoten hat kein Kind: Wir löschen den Knoten direkt
- Der zu entfernende Knoten hat ein Kind: Wir ersetzen den Knoten durch sein Kind
- Der zu entfernende Knoten hat zwei Kinder: Wir ersetzen den Knoten durch seinen *symmetrischen Nachfolger*

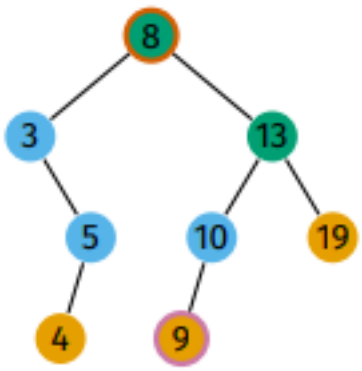
Die ersten beiden Fälle sind selbsterklärend, darum gehen wir hier auf den dritten Fall ein. Die Idee ist, dass wir den Knoten durch das “nächstgrösste” Element ersetzen wollen, um die Suchbaumeigenschaft zu erhalten. Das nächstgrösste Element ist das kleinste Element im rechten Teilbaum. Es zu finden ist leicht:

Vom zu entfernenden Knoten aus gehen wir einen Schritt nach rechts, und dann solange nach links, bis wir ans Ende des Baums gelangen.

Das Ganze sieht dann ungefähr so aus:

```
  def symmetric_desc(start_node:):
"""Find and return the symmetric descendant of the provided node. Also
rearranges the tree so that the symmetric descendant takes the place of
the start_node, while the right children of the symmetric descendant are
attached to its prior parent."""
if start_node.left is None:
  return start_node.right

if start_node.right is None:
  return start_node.left

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

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

return node
```

Bildlich: Wir entfernen die 8, der symmetrische Nachfolger ist die 9:



5.1.4 Traversals

Es gibt drei Traversalarten, um Bäume kompakt als Liste darzustellen:

- **Preorder**: ROOT, dann rekursive LEFTCHILD, RIGHTCHILD
- **Postorder**: Rekursiv LEFTCHILD, dann RIGHTCHILD und schlussendlich ROOT
- **Inorder**: Rekursiv LEFTCHILD, dann ROOT und schlussendlich RIGHTCHILD

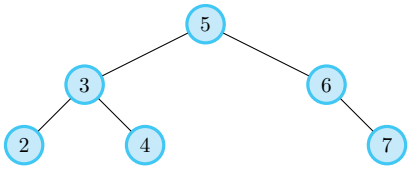
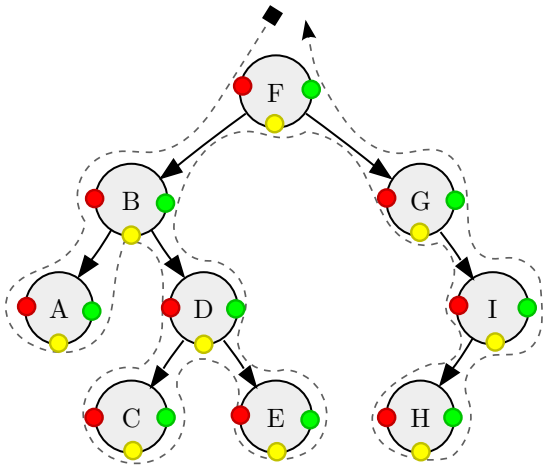


Abbildung 5.1: Traversierungsarten - Ein BST

- PREORDER: 5,3,2,4,6,7
- INORDER: 2,3,4,5,6,7
- POSTORDER: 2,4,3,7,6,5

Anstatt des mühseligen, rekursiven Aufzählens macht es graphisch Sinn, sich folgenden Trick zu Nutze zu machen: Man beginnt, in dem man den Umriss des Baums markiert. Dann markiert man entweder die linke Seite (preorder), die untere Seite (inorder) oder die rechte Seite (postorder) jedes Knotens. Nun folgt man einfach dem Umriss und notiert sich, in welcher Reihenfolge man den Markierungen begegnet.



Als Beispiel: Möchte ich die Postorder des oben abgebildeten Baumes bestimmen, markiere ich alle Knoten auf der rechten Seite (also nur die grüne Markie-

rung). Dann folge ich dem Umriss des Baums (schwarz-gestrichelt) und schreibe die Knoten in der Reihenfolge auf, in der ich die Markierung treffe: A, C, E, D, etc.

Die Traversierungen in Python sind sehr einfach:

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

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

5.2 Heaps

Heaps sind ebenfalls Bäume, aber nicht *Such*bäume. Wir verwenden Heaps, um einfach auf das Maximum bzw. das Minimum zugreifen zu können. Für dieses Kapitel sprechen wir immer von einem Max-Heap, für einen Min-Heap könnt ihr einfach "grösser" durch "kleiner" ersetzen, und Maximum durch Minimum.

Ein Heap ist ein binärer Baum, wobei er zusätzlich *vollständig* sein muss - Die einzigen Lücken sind also "unten rechts" im Baum. Weiterhin muss (anstatt bei BSTs die Suchbaumeigenschaft) jeder Knoten einen grösseren Wert als seine Kinder speichern - das Maximum befindet sich also an der Wurzel, daher der Name Max-Heap. Da Heaps vollständige Bäume sind, haben sie die tolle Eigenschaft, immer eine Höhe von $\mathcal{O}(\log n)$ zu haben.

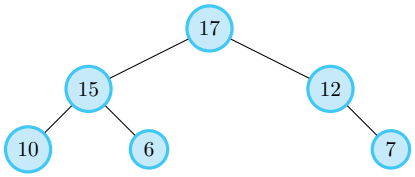
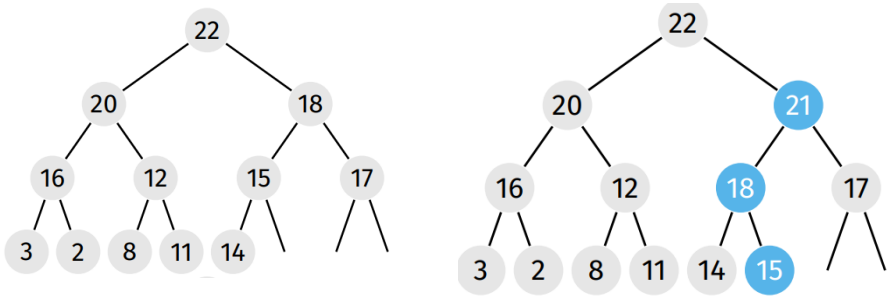


Abbildung 5.2: Ein Max-Heap

Wie sehen die Operationen auf einem Heap aus?

5.2.1 Einfügen

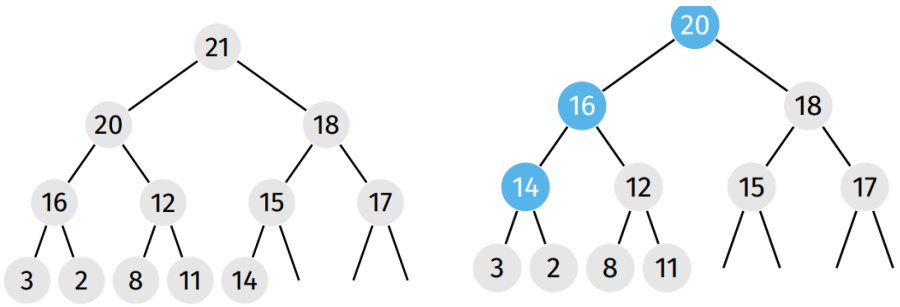
Um ein Element einzufügen, fügen wir es zuerst am ersten freien Platz ein. Da dies möglicherweise die Heap-Bedingung verletzt, müssen wir nun weiterhin dieses Element solange mit seinem Elter tauschen, bis die Heap-Bedingung erfüllt ist. In den Abbildungen unten fügen wir die 21 ein, alle Elemente, die mit der 21 getauscht wurden sind blau markiert:



Das Einfügen hat eine Laufzeit von $\mathcal{O}(\log n)$, da wir im schlimmsten Fall bis zur Wurzel hochtauschen müssen.

5.2.2 Entfernen des Maximums

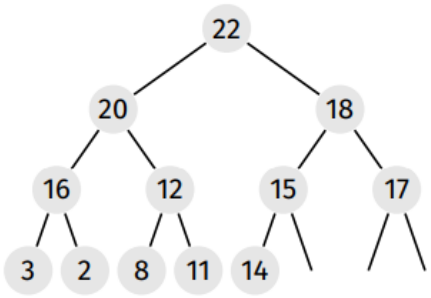
Das Entfernen des Maximums gestaltet sich wie folgt: Wir entfernen das Maximum und ersetzen es durch das "letzte" Element, ganz unten rechts. Da dies wiederum die Heap-Bedingung verletzen kann, müssen wir nun nach unten tauschen - und zwar immer mit dem grösseren Kind. Wir tun dies solange, bis beide Kinder kleiner sind. Unten ein Beispiel, wir entfernen das Maximum, ersetzen es durch die 14 und tauschen dies nach unten durch:



Wiederum ist die Laufzeit $\mathcal{O}(\log n)$, da wir im schlimmsten Fall bis nach ganz unten durchtauschen.

5.2.3 Heap als Array

Anstatt wie bei Bäumen Traversals zu verwenden, ist es relativ leicht ein Heap als Array darzustellen: Wir schreiben einfach von oben nach unten, links nach rechts alle Elemente auf. Dies bedeutet, dass für ein Element an Stelle i im Array gilt, dass sein Elter an der Stelle $\lfloor (i-1)/2 \rfloor$ und seine Kinder an den Stellen $2i+1$ und $2i+2$ befinden.




```
# Heap als Array, mit Indizes darunter zur Orientierung
[22, 20, 18, 16, 12, 15, 17, 3, 2, 8, 11, 14]
[ 0,  1,  2,  3,  4,  5,  6, 7, 8, 9, 10, 11]
```

5.3 Hashing

Können wir eine *noch schnellere* Datenstruktur bauen? Wenn wir nur Suchen, Einfügen und Entfernen unterstützen wollen - ja!

Wenn wir auf eine Liste zugreifen (also `l[i]`) ist dies sehr schnell. Aber, wenn wir ein Element suchen wollen, wissen wir nicht, an welcher Stelle es ist.

Dieses Problem lösen wir mit *Hashing* - Wir verwenden eine Funktion, die uns angibt, welcher Index ein Element haben sollte.

Um ein Element x einzufügen, fügen wir es einfach an der Stelle `h(`x`) % len(A)` ein. Wollen wir überprüfen, ob y in der Hash-Tabelle ist, checken wir die Stelle `h(`y`) % len(A)`.

Das gross Problem - wie gehen wir mit Kollisionen um? Es gibt zwei Optionen:

Entweder, wir nehmen einfach den nächsten freien Index - das ist einfach, aber bei der Überprüfung, ob ein Element in der Liste ist müssen wir nun potentiell mehrere Indizes abfragen.

Alternativ speichern wir an jedem Index eine List von allen Elementen, die an diese Stelle gehören. Im schlechtesten Fall haben wir dann einfach eine sehr lange Liste an einem Index.

Für "gute" Hashfunktionen ist die Laufzeit selbst mit Kollisionen *in Erwartung*

$\mathcal{O}(1)$. Die worst-case-Laufzeit bleibt aber $\mathcal{O}(n)$, wenn sich alle Elemente einen Slot teilen müssten.

Kapitel 6

Memoization und Dynamische Programmierung

Mit Memoization und Dynamischer Programmierung können wir Probleme, die naiv gelöst exponentielle Laufzeit haben, in deutlich besserer Laufzeit (oft linear) lösen. Die Hauptidee ist es, das mehrfache Berechnen von Resultaten zu verhindern, in dem man diese in einer Tabelle abspeichert.

6.1 Memoization

Betrachten wir als Beispiel die Fibonacci Zahlen. Die Fibonacci Zahlen sind definiert als

$$F_n = \begin{cases} n & \text{wenn } n \leq 1 \\ F_{n-1} + F_{n-2} & \text{wenn } n \geq 2 \end{cases}$$

Die naive Implementation in Python sähe etwa so aus:

```
def fibo(n):
  if n <= 1:
    return n
  else:
    return fibo(n-1) + fibo(n-2)
```

Das Umwandeln in eine memoisierte Lösung ist einfach: Wir fügen zuerst Checks ein, ob die Lösung bereits im memo-table vorhanden ist, und nach der Berechnung speichern wir das Resultat in diesem table bevor wir returnen.

```python
def fibo_mem(n, memo):
    # 1st, check if already in table
    if memo[n] != -1:
      return memo[n]

    # 2nd, check if basecase
    if n == 1 or n == 2:
      return 1

    # Calculate result recursively, update table, return
    res = fibo_mem(n-1, memo) + fibo_mem(n-2, memo)
    memo[n] = res
    return res

def fibo(n):
    return fibo_mem(n, [-1] * (n+1))
```

Das obige Template lässt sich auf fast alle Probleme anwenden, die wir einfach rekursiv definieren können.

Wichtig: Je nach Problem solltet ihr den anfänglichen Wert des memo-table ändern. In diesem Fall wurde -1 gewählt, aber ihr könnt je nach Problem auch `False` oder `None` wählen. Vergesst nicht, dies sowohl im Aufruf als auch im Check zu ändern!

6.2 Dynamic Programming

Die Idee hier ist, anstatt "auf Verlangen" das Resultat zu berechnen und zu speichern, stattdessen in einem for-Loop alles zu berechnen, und am Ende einfach diese Tabelle auszulesen. Im Falle Fibonacci:

```python
def fib_DP(n):
  # Tabelle erstellen
  F = [None] * (n+1)
  # Randfälle
  F[0] = 0
  F[1] = 1

  # Bottom-Up for-Schleife
  for i in range(2, n+1):
    F[i] = F[i-1] + F[i-2]

  # letzten Wert zurückgeben
  return F[n]
```

Neben dem "direkten Ausrechnen" sind auch Aufgaben wie die folgende typisch, wobei ein Eintrag nicht direkt die Lösung symbolisiert, sondern etwas á la "`T[i]` ist die Lösung des Problems, wenn wir an der Stelle `i` starten würden". Besonders beliebt bei 2D-Problemen!

Problemstellung:

- Gegeben eine Liste `a`.
- Sie starten bei Index `0`
- Sie dürfen an der Stelle `i` zwischen 1 und `a[i]` weit springen
- Welches ist die minimale Anzahl von Sprüngen, um an das Ende der Liste (oder darüber hinaus) zu gelangen?

Wenn wir nun definieren, dass in unserer Tabelle `T[i]` die Lösung ist, wenn wir an der Stelle `i` starten würden, können wir folgende Rekurrenz notieren:

$$T(i) = \begin{cases} 0 & \text{falls } i = n \\ 1 + \min(S(i+1), S(i+2), \ldots, S(i+\texttt{a[i]}) & \text{falls } 0 \leq i < n \end{cases}$$

Wir sehen nun, dass $S(i)$ nicht von den "vorherigen", sonder den "nachfolgenden" Resultaten abhängt. Wir müssen also von "hinten nach vorne" berechnen:

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

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

  return s[0]
```

Allgemeine Vorgehensweise bei DP-Problemen:

- Überlegt euch, was ihr für Teilprobleme braucht, um euer Problem zu lösen
- Schreibt diese Abhängigkeit als Rekursion auf
- Implementiert nun entweder Memo nach Template, oder DP indem ihr die Tabelle in einem for-loop ausfüllt
- Übung macht den Meister, löst einige alte Prüfungsaufgaben um DP zu trainieren!

Teil III

Machine Learning

Kapitel 7

Machine Learning - in Code

Dieses Kapitel soll euch einen kurzen, Step-by-Step guide zu ML geben. In dem anderen Kapitel dieses Teils erfahrt ihr mehr zur Theorie von ML.

In der Regel folgen wir immer einem typischen Ablauf:

- Prepare Dataset
- Select Model (-Family) and CV
- (Select loss function)
- Train Model
- Evaluate Model
- Predict on new data

Für uns sind manche von diesen Schritten weniger relevant, oder wir müssen nur Teile davon ausführen bzw. in Code umwandeln.

7.1 Datensatz vorbereiten

Wir brauchen uns glücklicherweise nicht mit dem Sammeln oder Erstellen von Daten zu plagen, sondern müssen diese nur einlesen:

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

# 'price' durch den richtigen Spaltennamen ersetzen
# -> Schaut ins csv
X = df.drop(['price'], axis=1)
y = df['price']
```

```
# Split into Train and Test data:
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2)
```

7.2 Modell auswählen

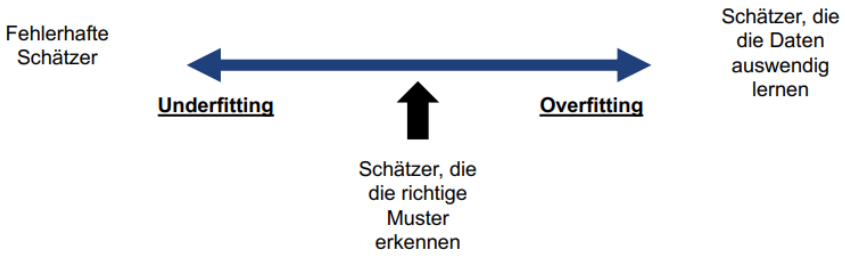
Das Modell auszuwählen heisst für uns vor allem, das richtige von sklearn zu importieren. Einige Optionen:

- Decision Trees
- Linear Regression
- MLPClassifier
- MLPRegressor

Eine ähnliche Auswahl in Code - in einer echten Aufgabe würdet ihr natürlich nur ein Modell, und nicht zwei erstellen:

```
# Below are some options for models
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier

model = DecisionTreeClassifier()
model2 = MLPClassifier(hidden_layer_sizes=(7, 7, 3), max_iter=1000)
```

Interlude: CV



Bei der Wahl des Modells habt ihr euch vielleicht gefragt, warum wir nicht einfach immer das "stärkste" Modell wählen - also einen möglichst grossen Entscheidungsbaum, ein Polynom mit möglichst grossem Grad, ein möglichst grosses Neuronales Netz etc.

Das Problem ist *Overfitting*: Sind eure Daten auch nur minimal fehlerhaft (z.B. durch menschliche Fehler bei der Klassifikation, oder Messungenauigkeiten), so wird ein "zu starkes" Modell diese Fehler mitlernen. Dies hat zur Folge, dass das Modell neue Daten nur sehr schlecht vorhersagen kann.

Natürlich wollen wir aber ein "stark genuges" Modell wählen - denn falls wir zum Beispiel ein Polynom von Grad 5 mit einem Polynom von Grad 1 vorhersagen wollen, werden wir ebenfalls äusserst schlecht abschneiden. Das wäre *underfitting*.

Um gute Hyperparameter (i.e. Tiefe des Baums, etc.) zu finden, benutzen wir Cross-Validation: Wir teilen unseren Datensatz auf in einen Trainings und einen Testdatensatz. Dann trainieren wir verschiedene Modelle auf dem Trainingsdatensatz und überprüfen ihre Performance auf dem Testdatensatz. Haben wir dadurch ein Modell gefunden, welches weder over- noch underfitted, können wir dieses auf dem gesamten Datensatz trainiern und verwenden.

Um noch bessere Abschätzungen für gute Modellparameter zu machen, verwenden wir *K-Fold Cross-Validation*. Wir splitten unseren Datensatz in eine gewisse Anzahl von gleich grossen Teilen (oft z.B. 5), und spielen das oben beschriebene Szenario durch, wobei jeder Teil einmal als Testdatensatz herhält. Dann können wir die Fehlerquote über insgesamt 5 verschiedene Test- und Trainingsdatensätze überprüfen. In der Abbildung unten würden wir uns also für das Modell mit Tiefe 20 entscheiden.

	Tiefe=2	Tiefe=20	Tiefe=200
% mistakes in	20	5	50
% mistakes in	15	3	80
% mistakes in	13	4	70

ETH Zürich

7.3 Verlustfunktion

Diese interessiert uns nicht weiter - sie dient nur dazu, das Modell zu trainieren. Für unsere Zwecke reicht, was auch immer die Modelle standardmässig verwenden.

7.4 Training

Dieser Teil ist theoretisch am kompliziertesten, praktisch am einfachsten:

```
model.fit(X_train, y_train)
```

7.5 Validierung

Wir überprüfen die Qualität des trainierten Modells, indem wir es Vorhersagen treffen lassen, und diese mit der Wahrheit abgleichen. Printed das Resultat eurer Wahl:

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

y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("R2 Score: " + r2)
```

7.6 Modell einsetzen

Nun gilt es, die Daten ohne zugehöriges `y` einzulesen und den Wert vorherzusagen:

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

Kapitel 8

Machine Learning - in Theory

In diesem Kapitel erfahrt ihr die theoretischen Grundlagen zu dem Speedrun von ML, der in dieser Vorlesung präsentiert wurde.

8.1 Problemtypen

Ihr habt euch mit vier verschiedenen Problemtypen beschäftigt, wobei ihr vorallem mit den ersten beiden vertraut sein solltet:

Classification In diesem Problem geht es darum, einen Datenpunkt (wobei ein Datenpunkt natürlich viele features aufweisen kann) einer Klasse zuzuordnen. Oft gibt es nur zwei Klassen (z.B. "Giftig oder Ungiftig"), aber es kann auch mehrere Klassen geben - z.B. in Bilderkennung, wo es vllt. 100 oder mehr Klassen geben kann

Regression Bei Regression geht es darum, einem Datenpunkt eine Zahl zuzuordnen.

Clustering Clustering ist eine etwas anderes Problem - denn anstatt dass ihr ein Modell mit bestehenden Daten trainiert und dann anwendet, sollen jetzt sogenannte "Cluster" - Ansammlungen von Datenpunkten - von eurem Modell erkannt werden.

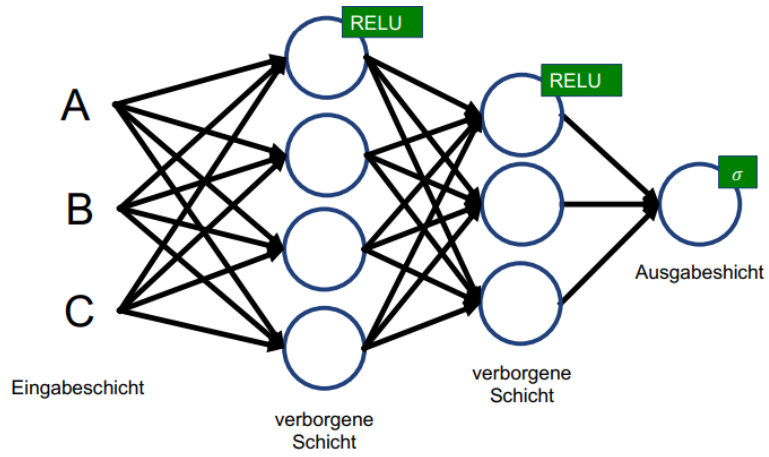
Dimensionality Reduction Hierbei ist euer Ziel, Daten mit möglichst geringem Verlust zu komprimieren, in dem ihr herausfindet, was "wirklich wichtig" ist, um euren Datensatz rekonstruieren zu können.

8.2 Modelltypen

Entscheidungsbaum Ein Entscheidungsbaum/Decision Tree ist ein relativ einfaches Modell welches vor allem zur Klassifikation eingesetzt wird. Ein Entscheidungsbaum hat einfache "Ja/Nein" Entscheidungen, welche zu einem Resultat führen. Je grösser die Tiefe eines Baum sein kann - d.h. je mehr solche Entscheidungen gefällt werden können - desto komplexere Daten kann der Baum klassifizieren. Beim Training lernt das Modell, welche Entscheidungen am Besten sind.

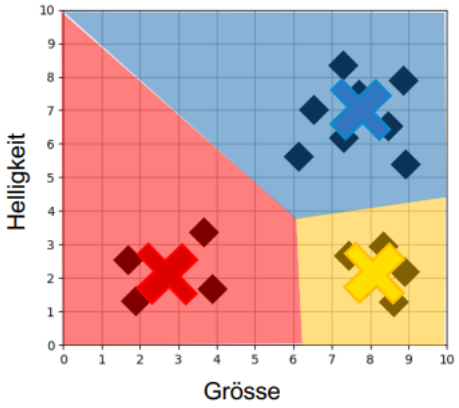
Lineare Regression Lineare Regression löst Regressionsprobleme, in dem ein Modell lernt, welche lineare Gleichung am Besten die Daten abbildet. Beim Training lernt das Modell, welche Koeffizienten dieser linearen Gleichung am ehesten den Trainingsdaten entsprechen

Neuronale Netzwerke Ein neuronales Netz/neural network (NN) ist eine Ansammlung von verbunden *Neuronen*, welche mithilfe unterschiedlicher sogenannten Aktivierungsfunktionen den richtigen Output lernen. Das Trainieren von NNs ist deutlich resourcenintensiver und undurchsichtiger als das Trainieren von anderen Modellen. Der Vorteil ist, das NNs selbstständig relativ komplizierte Funktionen erlernen können, da jedes Neuron eine eigene Funktion lernen kann, und diese dann kombiniert werden. Für Anwendungen wie Bilderkennung werden sogenannte Convolutional Neural Networks verwendet, welche zusätzlich noch sogenannte Faltungsmatrizen auf ihren Input anwenden, um etwa Kanten zu erkennen.



8.3 K-Means

K-Means ist ein Algorithmus um sogenannte Cluster zu bestimmen. Dies funktioniert, in dem wir K "Zentroide" oder Mittelpunkte bestimmen, und jeden Punkte seinem nächstgelegenen Zentroid zuweisen.



Die Anzahl an Clustern müssen wir selbst bestimmen, da wir in diesem Fall nicht einmal Cross-Validatin gebrauchen können :(

Das Trainieren via K-Means ist relativ leicht:

- Zufällige Initialisation der Zentroide
- Wiederhole bis konvergiert:
  - Weise jeden Punkt seinem nächsten Zentroid zu
  - Platziere jeden Zentroid am Durchschnitt all seiner zugewiesenen Datenpunkte

K-Means konvergiert zwar *immer* zu einer Lösung, allerdings ist diese Lösung abhängig von der Initialisierung der Zentroiden.

Teil IV

Addendum

Kapitel 9

Tipps zur Prüfungsvorbereitung

Wie bereite ich mich am Besten für die Prüfung vor? Wenn es eine universelle Antwort darauf gäbe, würde der Prüfungsschnitt von allen Prüfungen wohl um einiges besser ausfallen. Dennoch möchte ich euch im Folgenden einige persönliche Tipps zum Lernen auf diese eine Prüfung geben.

Verschafft euch einen Überblick Anstatt blind mit dem Lernen anzufangen, solltet ihr als erstes rausfinden, was euch schon gut liegt und was ihr nochmal genauer repetieren müsst. Schaut euch am Besten zuerst an, welche Themen in welcher Woche dran waren. Gibt es welche, bei denen noch Unklarheiten vorhanden sind? Gibt es welche, bei denen ihr euch relativ sicher seid, alles verstanden zu haben? Überfliegt ein paar alte Prüfungen: Gibt es Aufgaben (über euren Stoff), bei welchen ihr total keinen Plan habt?

Theorie - Verständnis Die meisten der theoretischen Fragen bauen auf euerem Verständnis auf. Wisst ihr, wie die Sortieralgorithmen funktionieren? Wie Operationen in einem binären Suchbaum passieren? Was Over- und Underfitting ist? Anstatt einfach auswendig zu lernen, versucht, eure Verständnislücken zu füllen. Versucht, ein Thema einem Mitstudierenden zu erklären: Könnt ihr alle Aspekte abdecken? Müsst ihr irgendwo ins "irgendwie so" ausweichen? Falls ihr jemandem ein Thema erklären könnt, seid ihr schon auf einem sehr guten Weg.

Coding - Übung Nichts bereitet euch besser auf den Programmierteil vor, als viel zu programmieren. Löst (in-class) Aufgaben, die ihr noch nicht gemacht habt, oder macht eine alte Aufgabe noch einmal - vielleicht mit einem anderen Ansatz? Das wichtigste ist, dass ihr *selbst etwas programmiert*, und euch nicht nur (eure eigenen oder die offiziellen) Lösungen

anschaut und hofft, dass ihr das an der Prüfung dann auch könnt.

Die Aufgaben die ihr während dem Semester gelöst habt sollten auf dem gleichen Level wie die Prüfungsaufgaben sein. Heisst, könnt ihr die Aufgaben lösen, seid ihr gut vorbereitet. Wir erwarten keine unglaublich smarten Ansätze die nur ein minmaler Bruchteil von euch haben wird, sondern vor allem, dass ihr Probleme auf dem Level der Aufgaben lösen könnt.

Stellt Fragen Falls ihr irgendwas nicht versteht - Fragt! Eure Mitstudierenden, eure Freunde, eure Freunde gegen Bezahlung (aka TAs) - Für eure Mitstudierenden ist es eine gute Übung, und meistens hilft ein anderer Blickwinkel beim Verständnis sehr.

Kapitel 10

Tipps für den Prüfungstag

Es folgt eine Liste von allgemeinen Prüfungstipps. Viele davon werdet ihr auf den ersten Blick als offensichtlich abtun, aber es kann hilfreich sein, sich diese Sachen vor der Prüfung nochmal in Erinnerung zu rufen.

Natürlich sind dies nur meine persönlichen Ratschläge, ihr wisst selbst am Besten, wie ihr euch in Prüfungssituationen verhaltet und worauf ihr achten solltet. Falls nicht, lade ich euch ein, nach den Prüfungen zu reflektieren, was denn eure persönlichen Ratschläge an einen Mitstudenten wären.

Macht euch am Tag der Prüfung nicht wahnsinnig Es ist ziemlich sinnfrei, sich am Tag der Prüfung noch alle alten Aufgaben reinzuhämmern oder ähnliches. Fokussiert euch darauf, euer Wissen zu zeigen. Besonders wichtig - lasst euch nicht von anderen wahnsinnig machen. Wenn jemand über Probleme diskutiert, von denen ihr nie zuvor gehört habt, hilft es wahrscheinlich nicht, sich damit auseinanderzusetzen. Im schlimmsten Fall seid ihr nachher nur noch mehr verwirrt.

Lest die Frage genau Ihr solltet euch zu 100% im Klaren sein, was für eine Aufgabe ihr löst. Nichts schmerzt mehr, als die falsche Aufgabe richtig zu lösen, und dafür dann 0 Punkte zu bekommen.

Besonders bei Theoriefragen gibt es viele kleine, unscheinbare Dinge, die ihr im Prüfungsstress übersehen könntet: Betrachtet ihr einen Min- oder einen Max-Heap? Wird spezifiziert, ob das Array sortiert oder unsortiert ist? Selection oder Insertion Sort?

Checkt wieviele Punkte eine Aufgabe gibt Dies ist ein gutes Indiz dafür, wie aufwendig diese Aufgabe sein sollte. Falls das Format beibehalten wird, habt ihr bei den Programmieraufgaben auch eine empfohlene Zeitangabe. Falls ihr für eine 5-Minuten-Aufgabe was super kompliziertes anfangt, seid ihr vermutlich auf dem Holzweg.

Löst zuerst die einfachen / kurzen Theorieaufgaben Ich empfehle euch, zuerst die "einfachen Punkte" zu ergattern. Dafür gibt es mehrere Gründe:

- Nervosität könnt ihr wohl am Besten abbauen, in dem ihr Aufgaben löst. Kurze / Einfache Aufgaben helfen dabei.
- Die Zeit die ihr pro erreichtem Punkt verwendet, ist klein.
- Die Wahrscheinlichkeit, sich irgendwo zu vertun und Zeit "zu verschwenden" ist viel geringer als bei Programmieraufgaben
- Ihr müsst nicht am Ende der Prüfung im Stress die einfachen Aufgaben lösen oder gar zurücklassen

Löst Programmieraufgaben auf Papier So paradox es auch klingt, es ist oft empfehlenswert, die Code-Aufgaben zuerst auf Papier zu lösen, und dann den Code nur noch "abzuschreiben". Besonders effektiv funktioniert dieser Ansatz bei DP (wie sieht mein Table aus? Wie definiere ich die Rekurrenz? Was sind meine Basecases? Berechnungsreihenfolge?), aber auch bei den weniger komplexen Aufgaben kann es helfen, sich kurz zu überlegen, wie man die Aufgabe angehen will, bevor in die Tasten zu hauen: Was für Operationen mache ich auf welche Elemente der Liste? Wie könnte meine List Comprehension aussehen? Welche Fälle muss ich abdecken? etc.

Submit often Wenn ihr auf Code Expert etwas submitted, erstellt ihr auch automatisch einen Backup-point. Es zählt nur euer bestes Ergebnis, also submitted regelmässig, auch wenn ihr "nur" eine Teilaufgabe gelöst habt. Es kostet euch nichts, und ein Backup kann sehr, sehr hilfreich sein.