Algorithmen & Datenstrukturen

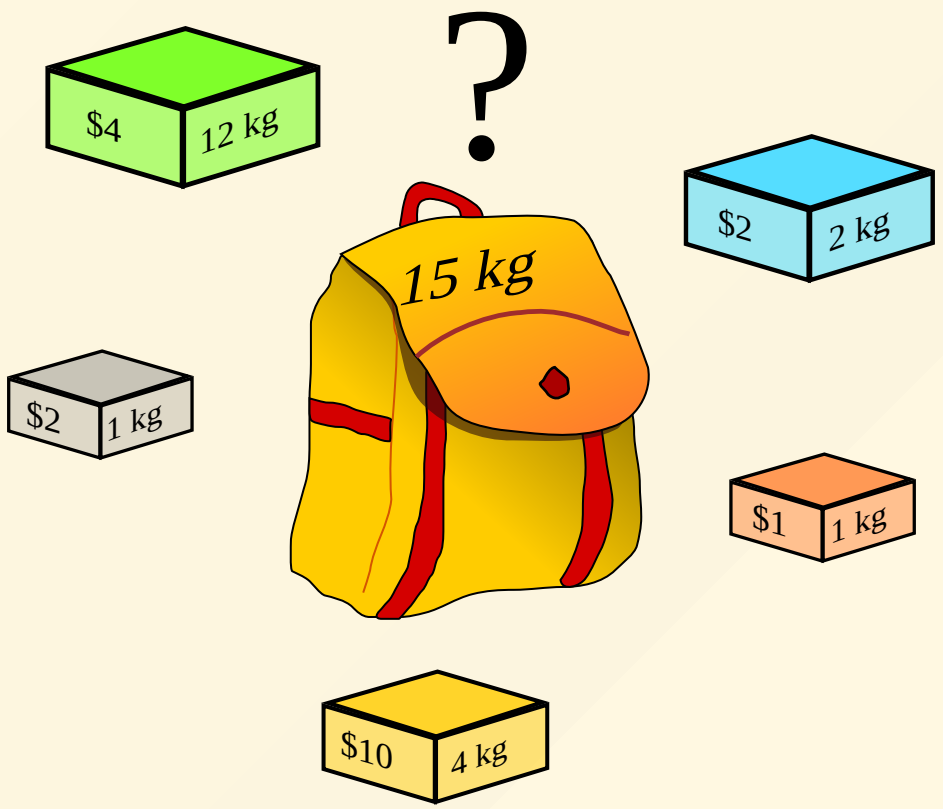
Julian Huber & Matthias Panny

Algorithmen

🎯 Lernziele

- Studierende wissen was Algorithmen sind
- Studierende wissen wie Algorithmen beschrieben werden
- Studierende kennen verschiedene Klassifizierungen von Algorithmen

Beispiel: Rucksackproblem



- Wie viel Wert passt maximal in den Rucksack, wenn das Gewicht die Auswahl limitiert? Welche Gegenstände müssen mitgenommen werden?
- Abstraktion realer Probleme in der Mechatronik, Logistik, Finanzwesen, etc:
  - Auswahl von Lieferungen bei Luftfracht
  - Scheduling von Berechnungen auf Mikroprozessoren bei gegebenem Speicherplatz, Energieverbrauch, etc.

Bild: https://de.wikipedia.org/wiki/Datei:Knapsack.svg

Lösung des Rucksackproblems

- Bei einer kleinen Anzahl von Gegenständen kann das Problem durch Ausprobieren aller möglichen Kombinationen gelöst werden
- Bei einer großen Anzahl von Gegenständen ergeben sich einige Fragen die sich stellen

Fragen die sich stellen:

- Sollten wir systematisch vorgehen?
- Können wir die optimale Lösung finden?
- Können wir abschätzen, wie lange wir dafür brauchen?

→ beschäftigen mit Algorithmen um diese Fragen zu beantworten

Begriffe

> Algorithms + Data Structures = Programs
> — *Niklaus Wirth*

> Ein Algorithmus ist ein Verfahren zur Lösung eines bestimmten Problems in einer endlichen Anzahl von Schritten für eine endlich große Eingabe.

Beispiel - Finde die größte Zahl in einer Liste

- Gewisse Eigenschaften des Algorithmus sind bereits bekannt
- Algorithmus soll nun präzisiert werden

Finde die größte Zahl in einer Liste

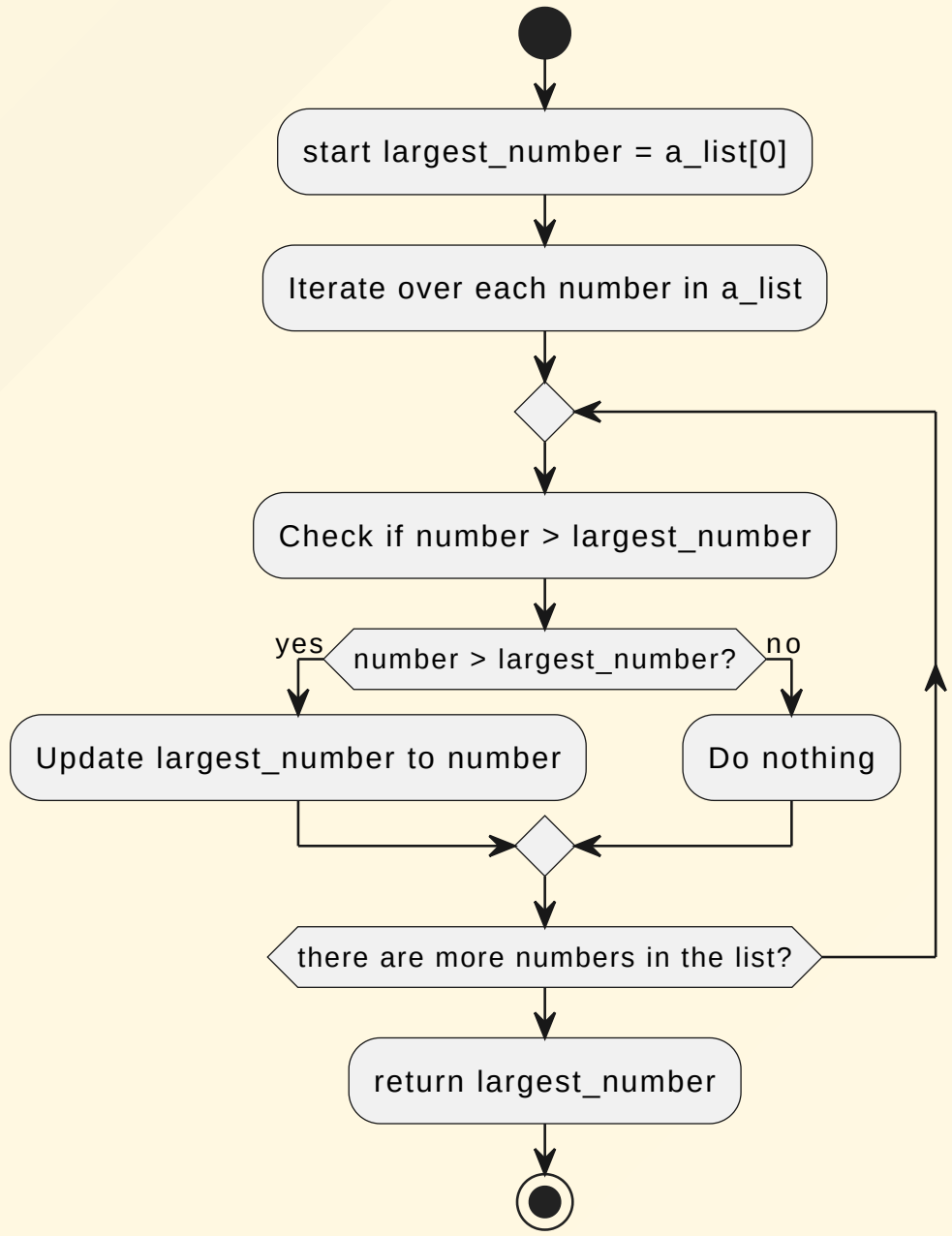
- Problem: Was ist die größte Zahl
- Eingabe: Liste (von Zahlen)
- Lösung: Größte Zahl
- Schritte: ?

Algorithmus zum Auffinden der größten Zahl in einer Liste

- Wir wollen die größte Zahl in einer Liste von Zahlen finden
- Welche "Bausteine" muss unser Algorithmus haben?

Beispiel - largest_number.py

```
def find_largest_number(a_list: list[int]) -> int:
    largest_number = a_list[0]

    for number in a_list:
        if number > largest_number:
            largest_number = number
    return largest_number
```



Bausteine

Definierter Input

- Vordefinierte Eingangsparameter z.B. die Liste → wie bei einer Funktion
- Zusätzliche Anforderungen z.B. ist die Liste vorsortiert? → denn der Algorithmus zum Finden der größten Zahl einer absteigenden sortierten Liste ist trivial

Definierter Output

- Zusicherung, dass der Output bestimmten Anforderungen entspricht z.B. Output ist eine Zahl und tatsächlich die Größte

Endliche Sequenz eindeutiger Instruktionen

- unabhängig von der Implementierung (Programmiersprache)
- Neben In- und Output können auch dabei Hilfsvariablen angelegt werden

Eigenschaften

Terminierung

- Der Algorithmus muss nach endlich vielen Schritten terminieren

Korrektheit

- Ein korrekter Algorithmus stoppt (terminiert) für jede Eingabeinstanz mit der durch die Eingabe-Ausgabe-Relation definierten Ausgabe
- Ein inkorrekter Algorithmus terminiert gar nicht oder mit einer nicht durch die Eingabe-Ausgabe-Relation vorgegebenen Ausgabe

Effizienz

- Bedarf an Speicherplatz und Rechenzeit
- Wachstum (Wachstumsgrad, Wachstumsrate) der Rechenzeit bei steigender Anzahl der Eingabe-Elemente (Laufzeitkomplexität)

Beschreibungslevel

High Level Beschreibung

- Gibt das grobe Vorgehen an → verbal oder grafische
- nicht alle Variablen sind zwangsweise definiert

Beispiel

- Wenn es keine Zahlen in der Liste gibt, dann gibt es auch keine höchste Zahl
- Nimm zunächst an, die erste Zahl in der Liste ist die größte Zahl in der Liste
- Für jede verbleibende Zahl in der Liste gehe wie folgt vor: Wenn diese Zahl größer ist als die aktuell größte Zahl, wird diese Zahl als größte Zahl der Liste betrachtet
- Wenn es keine Zahlen mehr in der Liste gibt, über die man iterieren kann, wird die aktuell größte Zahl als größte Zahl der Liste betrachtet

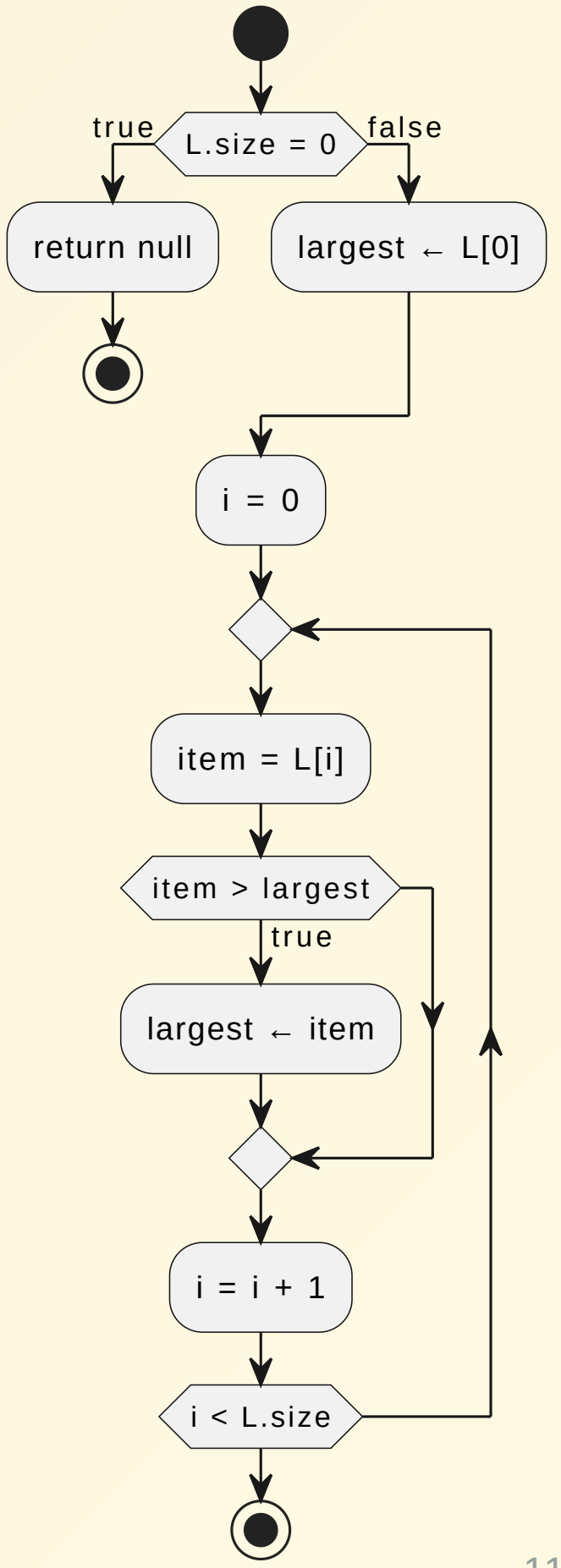
Beschreibungslevel

Low Level Beschreibung

- Formale Beschreibung
- Definiert alle relevanten Zustände
- z.B. als Pseudo-Code

Beispiel Pseudo-Code

```
Algorithm LargestNumber

Input: A list of numbers L.
Output: The largest number in the list L.

if L.size = 0 return null
largest ← L[0]
for each item in L, do
    if item > largest, then
        largest ← item
return largest
```



Andere Implementierung

- Die identische Problembeschreibung kann auch zu anderen Umsetzungen führen

Beispiel - `largest_number_std.py`

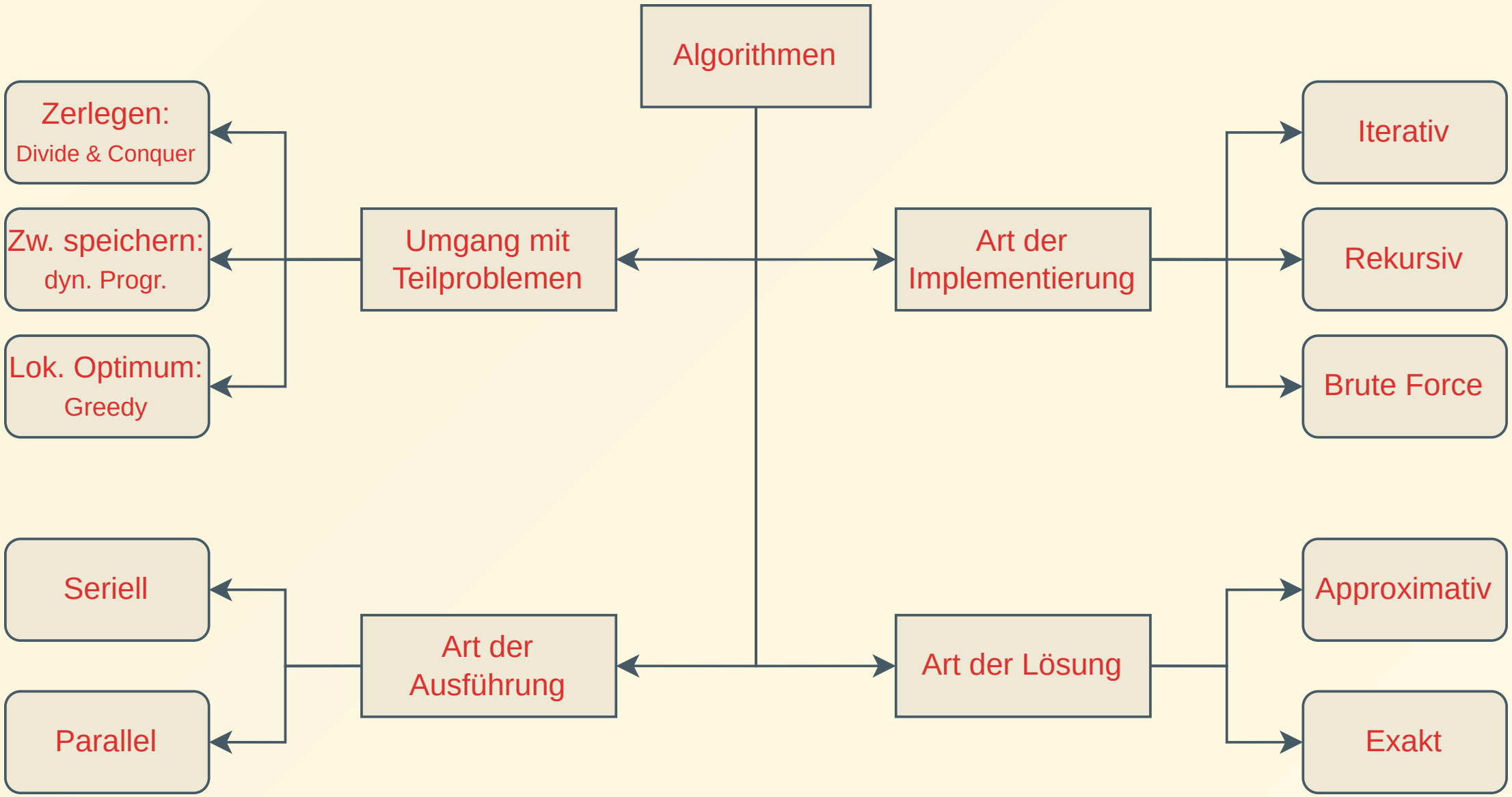
- Implementierung mit Hilfe der Python-Standardfunktionen & Exception Handling
- Der implementierte Algorithmus ist wesentlich komplexer → die Komplexität wird aber mit den Standardfunktionen von Python verborgen

```
my_list = [9, 42, 3, 8]

def find_largest_number(a_list: list[int]) -> int:
    if a_list:
        a_list.sort()
        return a_list[-1]
    else:
        raise IndexError("Provided list is empty")

def find_largest_number_simple(a_list: list[int]) -> int:
    if a_list:
        return max(a_list)
    else:
        raise IndexError("Provided list is empty")
```

Klassifizierung von Algorithmen

Klassifizierung von Algorithmen

- Algorithmen lassen sich unterschiedlich klassifizieren
- Algorithmen können auch mehreren Klassen angehören



Klassifizierung nach Art der Implementierung

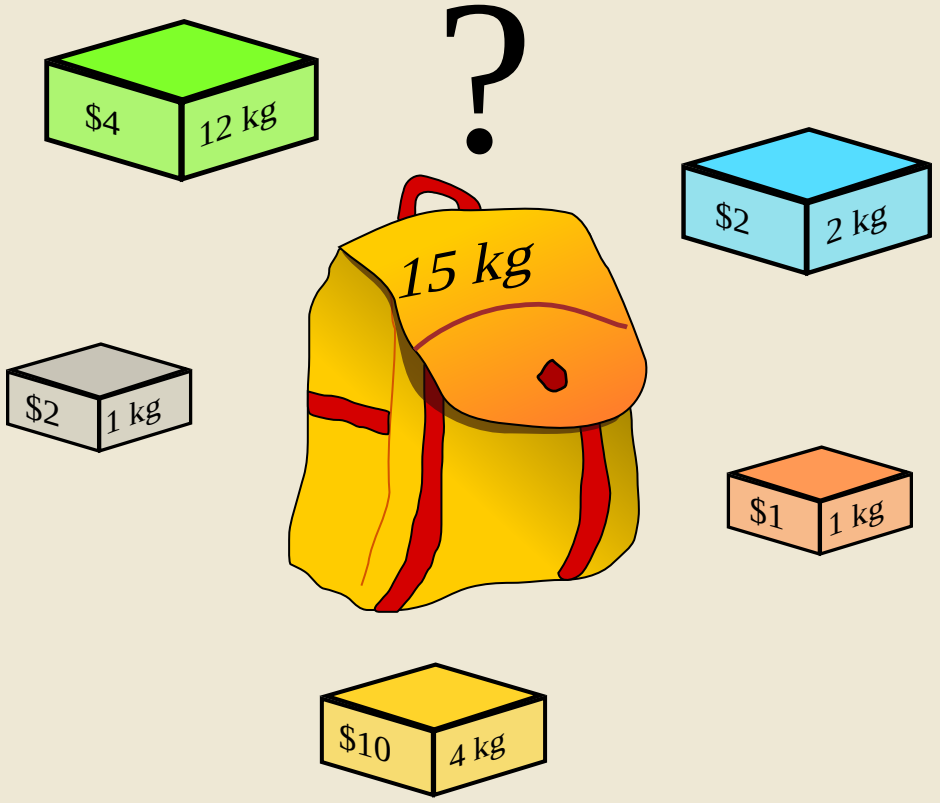
Brute Force

- Sowohl das Rucksackproblem als auch das Finden der größten Zahl in einer Liste, lassen sich durch vollständiges Ausprobieren aller Möglichkeiten lösen
- → Einsatz von (möglichst viel) Rechenleistung: Brute Force
- je länger die Liste wird, desto mehr Rechenleistung wird benötigt (die *Laufzeitkomplexität* wird später bei den Datenstrukturen noch eine Rolle spielen)

Klassifizierung nach Art der Lösung

Exakte Algorithmen

- sind in der Lage sind, für alle Inputs eines Problems eine optimale Lösung zu finden
- z.B. Rucksackproblem: Durch *brute force* können alle möglichen Kombinationen ausprobiert werden, um die *optimale Lösung* zu finden und dazu die *exakt* benötigten Gegenstände zu bestimmen



Lösung des Rucksackproblems mittels Brute Force

- Opt. Lösung durch vollständiges Absuchen des Lösungsraums
- Problem: Anzahl an Kombinationen steigt exponentiell mit der Anzahl der Gegenstände

Pseudo-Code

```
ALGORITHMUS optimaleLoesung:
    Übergabe: Liste von Gegenständen, Grenzgewicht des Rucksacks
    erzeuge eine erste Kombination, z.B. '00000'
    maxKombination = Kombination
    maxWert = Gesamtwert von Kombination
    SOLANGE noch nicht alle Kombinationen durchlaufen sind:
        erzeuge eine neue Kombination
        WENN der Gesamtwert von Kombination > maxWert und
             das Gesamtgewicht von Kombination <= grenzgewichtRucksack:
            maxKombination = Kombination
            maxWert = Gesamtwert von Kombination
    Rückgabe: (maxKombination, maxWert, Gesamtgewicht von maxKombination)
```

- Es gibt Probleme die mathematisch beweisbar lösbar sind, aber praktisch nicht lösbar sind → Zahl an Rechenschritten steigt zu schnell

Klassifizierung nach Art der Lösung

- In der theoretischen Informatik beschäftigt sich die Komplexitätstheorie damit, ob und wie effizient bestimmte Probleme algorithmisch gelöst werden können

Approximative Algorithmen

- in der Praxis sind Probleme häufig mit vertretbarem Ressourceneinsatz nur approximativ zu lösen
- nähern die exakte Lösung an → oft sinnvoller, als z.B. alle möglichen Kombinationen auszuprobieren
- dabei kommen häufig Abbruchkriterien zum Einsatz (vgl. `tol`, `max_iter`)
- Eine wichtige Gruppe von approximativen Algorithmen sind Heuristiken

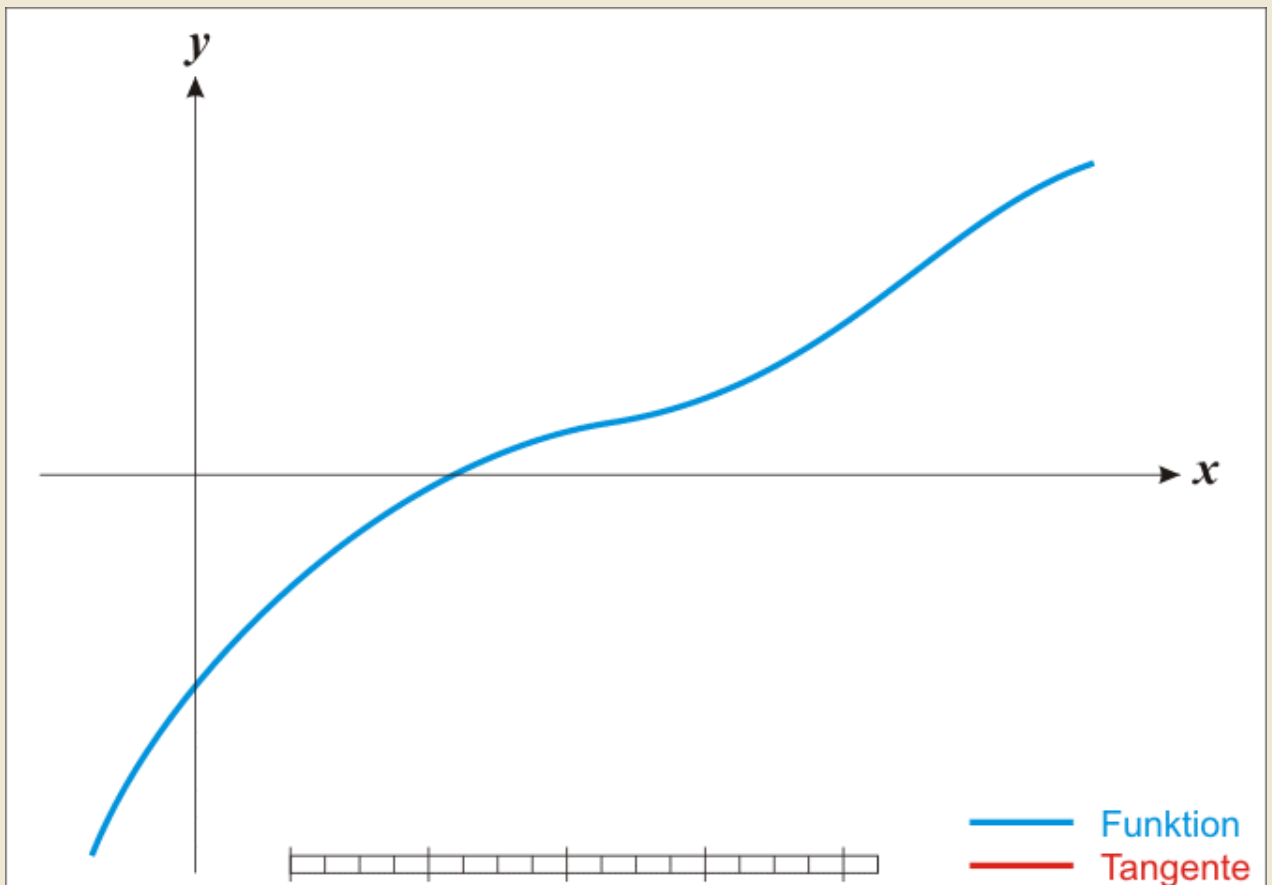
Klassifizierung nach Art der Implementierung

Iteration

- Nutzt Schleifenstrukturen und/oder Datenstrukturen (Stack, Queue, etc.) → wiederholtes Ausführen der gleichen Anweisungen
- Jeder rekursive Algorithmus kann als iterativer Algorithmus implementiert werden und umgekehrt

Beispiel - Newton-Raphson-Verfahren

- Finden von Nullstellen von Funktionen iterativ & approximativ



Klassifizierung nach Art der Implementierung

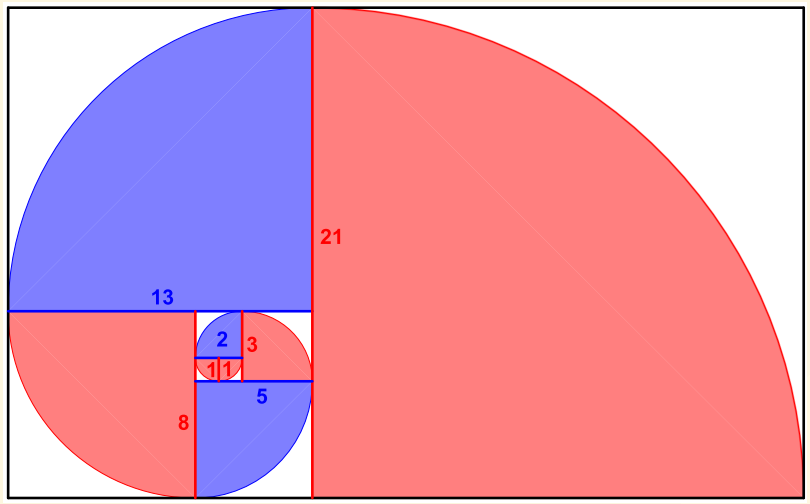
Newton-Raphson-Verfahren - newton_raphson.py

```python
def newton_raphson(func, func_deriv, init_guess, tol=1e-6, max_iter=100):
    """Use Newton-Raphson to find the root of a function."""
    x = init_guess
    iteration = 0
    while iteration < max_iter:
        f_x = func(x)
        f_prime_x = func_deriv(x)

        if abs(f_prime_x) < tol:
            raise ValueError("Derivative is too close to zero")

        x_new = x - f_x / f_prime_x

        if abs(x_new - x) < tol:
            return x_new

        x = x_new
        iteration += 1
    raise Exception("Newton-Raphson did not converge")

# Define the function and its derivative
def f(x):
    return x**2 - 4
def f_derivative(x):
    return 2 * x
# Call the Newton-Raphson function to find the root
root = newton_raphson(f, f_derivative, init_guess=1.5)
print("Approximate root:", root)
```

Beispiel

Wo sind die Nullstellen von $x^2 - 4 = 0$

Klassifizierung nach Art der Implementierung



Rekursion - Fibonacci-Folge - `fibonacci.py`

- `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...`
- Algorithmus ruft sich selbst wieder auf → Abbruch bei Erreichen einer Grundbedingung

```
def generate_fibonacci_recursive(n):
    """Generate the n-th Fibonacci number using recursion."""
    if n <= 1:
        return n
    else:
        return generate_fibonacci_recursive(n - 1) + generate_fibonacci_recursive(n - 2)

def gen_fibonacci_list(n):
    """Generates a list of the first n Fibonacci numbers."""
    fibonacci_sequence = []
    for i in range(n):
        fibonacci_sequence.append(generate_fibonacci_recursive(i))
    return fibonacci_sequence
```

Dyn. Prog. - Fibonacci-Folge - `fibonacci_dyn_prog.py`

- wollen wir nun die Fibonacci-Folge für `n=4` berechnen, so ergibt sich folgender Rekursionsbaum:

```
1. gfr(4)
2. gfr(3) + gfr(2)
3. (gfr(2) + gfr(1)) + gfr(2)
4. ((gfr(1) + gfr(0)) + gfr(1)) + gfr(2)
5. ((gfr(1) + gfr(0)) + gfr(1)) + (gfr(1) + gfr(0))
```

- → `gfr(2)` wird z.B. mehrfach berechnet

- Mittels dynamischer Programmierung bauen wir die Ergebnisse von unten nach oben auf und speichern sie für spätere Berechnungen:

```
def fib(n):
    if n == 0:
        return 0
    else:
        previous_fib, current_fib = 0, 1
        for _ in range(n - 1):
            new_fib = previous_fib + current_fib
            previous_fib = current_fib
            current_fib = new_fib
        return current_fib
```

Klassifizierung nach Umgang mit Teilproblemen

Greedy (gierige) Algorithmen

- Gruppe von iterativen Algorithmen
- bei jedem Schritt eine Entscheidung für das lokale Optimum getroffen → keine Berücksichtigung der zukünftigen Konsequenzen

Beispiele - Newton-Raphson-Verfahren & Rucksackproblem

- Newton-Raphson-Verfahren:
  - es wird immer nur die aktuelle Ableitung betrachtet ohne z.B. die grundlegende Gestalt der Funktion zu berücksichtigen
- Rucksackproblem:
  - Gegenstand mit dem höchsten Wert pro Gewichtseinheit zuerst einpacken → greedy_knapsack.py
  - Dies kann zu einer suboptimalen Lösung führen

Klassifizierung nach Umgang mit Teilproblemen

Zerlegung

- Zerlegung des Gesamtproblems in kleinere/einfachere Teilproblemen → z.B Divide-and-Conquer → meist rekursive Implementierungen

Zwischenspeicherung

- Systematisches Speicherung und Nutzen von Zwischenergebnissen → z.B. dynamische Programmierung
- z.B. beim Lösen des Rucksackproblems mittels Brute Force (oben), berechnen wir für jede mögliche Kombination erneut den Wert und das Gewicht → mittels dynamischer Programmierung können wir Zwischenergebnisse speichern und wiederverwenden

🤓 Weitere Klassifizierungen

Weitere Klassifizierungen

Serielle Algorithmen

- Jede Anweisung wird nacheinander ausgeführt
- meist das Standardvorgehen

Parallelen Algorithmen

- das Problem in Teilprobleme aufgeteilt und auf verschiedenen Prozessoren ausgeführt wird
- Wenn parallele Algorithmen auf verschiedene Maschinen verteilt werden, nennt man sie verteilte Algorithmen
  - vgl. Forks und Joins beim Activity Diagramm (VL 3.1)
  - Beliebte Pakete `multiprocessing` (meistens) und `threading` (manchmal) in Python

Weitere Klassifizierungen

Backtracking:

- für kombinatorische Probleme, die nur eine einzige Lösung haben. Solche Probleme haben mehrere Stufen und in jeder Stufe gibt es mehrere Optionen.
  Jede verfügbare Option ist in jeder Phase einzeln zu untersuchen.
- z.B. Lösen eines Sudoku oder finden eines Pfads (mehr dazu in der Graphentheorie)

Klassif. nach Lösungsansatz

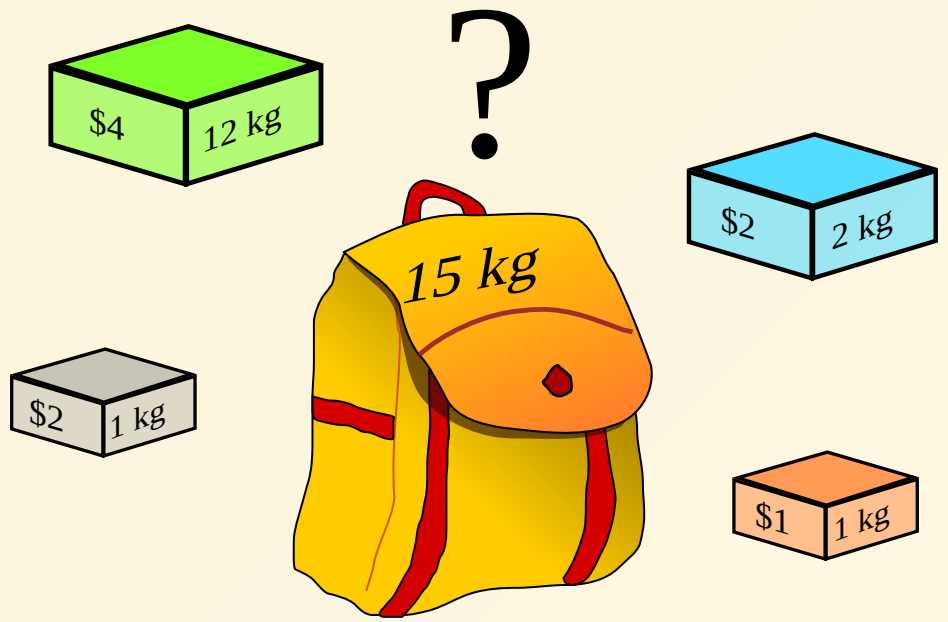
Divide and Conquer (Teilen und Bezwingen):

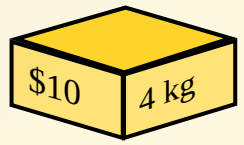
- Bei der Divide-and-Conquer-Strategie wird das Problem in Teilprobleme aufgeteilt, rekursiv gelöst und dann wieder zusammengefügt, um die endgültige Antwort zu erhalten.
- z.B. Merge sort, Quick sort (später mehr)

Reduction (Transform and Conquer):

- schwieriges Problem wird in bekanntes Problem umgewandelt
- z.B. Finden des Medians in einer Liste: 1. Liste sortieren 2. Wert bei halber Listen-Länge (vgl. Hausübung 1)

Hausaufgabe





- Lösen Sie das oben abgebildete Rucksackproblem: Wie viel Wert passt maximal in den Rucksack? Welche Gegenstände müssen mitgenommen werden?
- Stellen Sie den Lösungsansatz als UML-Activity-Diagramm dar und fügen Sie dieses im Notebook mit ein
- Lösen Sie das Problem dann in Python z.B. mittels Brute Force