IICM

TU Graz

706.088 INFORMATIK 1

ANALYTISCHE PROBLEMLÖSUNG, FERMI-PROBLEME, TÜRME VON HANOI, MODULE & KLASSEN IN PYTHON

ANALAYISCHE PROBLEMLÖSUNG

ANALYTISCHES DENKEN

- Sachliche Analyse einer Situation
- Pro/Contra Abwägung

ANALYTISCHES DENKEN

Beschreibt Fähigkeit von Personen Probleme (und Teilprobleme eines Problems) zu erkennen.

1. **Erkennen** eines **Problems**
2. Erkennen der **Teilprobleme**
    - Art, Umfang, Kontext, Schwierigkeit
3. Formen von **Lösungsansätzen**
    - Für **alle Teilprobleme** individuell
4. **Zusammenführung** von Lösungsansätzen
    - Erstellung eines übergeordneten Lösungsplans

BEISPIELE

- Design an evacuation plan for San Francisco
- How many times a day do a clock's hand overlap?
- How many piano tuners are there in the entire world?
- How many golf balls could you fit into a classic American school bus?
- How many vacuum's are made per year in the US?
- How many quarters (placed on top of each other) would it require to reach the top of the Empire State Building?

HOW MANY QUARTERS (PLACED ON TOP OF EACH OTHER) WOULD IT REQUIRE TO REACH THE TOP OF THE EMPIRE STATE BUILDING?

1. Problem: Wie viele Münzen brauche ich?
2. Teilprobleme
    1. Wie hoch ist das Empire State Building?
    2. Wie dick ist eine Quarter-Dollar Münze?
3. Teil-Lösungen:
    1. Höhe abschätzen (ca. 400m)
    2. Dicke abschätzen (ca. 1,5mm == 0,0015m)
4. Zusammenführung:
    1. Schlussrechnung: 400/0.0015 = 266666.66

FERMI-PROBLEME

Bezeichnet die quantitative Abschätzung für ein Problem zu dem man kaum Daten zur Verfügung hat

- geht zurück auf Enrico Fermi (Physiker)
  - war bekannt für gute Abschätzungen
  - Bsp: Papierschnipsel in der Luft (Druckwelle) um beim Test der ersten Atombombe die Sprengkraft der Bombe zu testen
- Idee: Kaum direkte Information vorhanden, aber Information über Zusammenhänge im Umfeld. Diese Zusammenhänge kann man sich zu Nutze machen um 'indirekt' das Problem zu lösen!

FERMI-PROBLEME

› **Problem**: Gewisses Allgemeinwissen ("gesunder Hausverstand") ist nötig!

› Um aber tatsächlich eine Lösung zu finden, muss das Vorwissen quantifiziert werden

› Prozess der Quantifizierung muss zusätzlich begründet und 'logisch untermauert' werden

› Durch 'Teilabschätzungen' (Wolkenkratzer und Münze) kann man Gesamtergebnis anschätzen

› Bei Fermi Problemen ist der Lösungsweg meist interessanter als das Ergebnis.

ALGORITHMUS

DEFINITION EINES ALGORITHMUS

› Ein Algorithmus ist eine **eindeutige Anleitung zur Lösung eines Problems**.

› Algorithmen bestehen aus **endlich vielen**, explizit definierten **Einzelschritten**.

› Sie können als Programm implementiert werden.

› Aber natürlich auch in normaler Sprache formuliert werden.

› Generell gilt: Ein Algorithmus überführt eine **bestimmte Eingabe** in eine **bestimmte Ausgabe**.

EUKLIDISCHER ALGORITHMUS

```
1   EUCLID_OLD(a,b)
2       wenn a = 0
3           dann return b
4       sonst solange b ≠ 0
5           wenn a > b
6               dann a ← a - b
7           sonst b ← b - a
8       return a
```

EIGENSCHAFTEN VON ALGORITHMEN

- **Determiniertheit**
  - » Bei gleichen Eingaben liefert der Algorithmus immer die gleichen Ausgaben/Ergebnisse.
- **Determinismus**
  - » Jeder Schritt ist eindeutig definiert. Die Abarbeitung des Algorithmus erfolgt immer eindeutig und klar definiert. Nicht-Deterministisch: zB: Zufälliges Start-Element wählen.

EIGENSCHAFTEN VON ALGORITHMEN

- **Finitheit**
  - Der Algorithmus hat endlich viele Befehle/Anweisungen
- **Terminiertheit**
  - Algorithmus ist terminierend, wenn jede Eingabe nach einer endlichen Anzahl von Schritten zu
    - einem gewollten Abbruch oder
    - einem Ergebnis führt
- **Effektivität**
  - Der Effekt/die Auswirkung eines Befehls innerhalb eines Algorithmus muss eindeutig sein.

TURINGMASCHINE

- Erfunden 1936 von Mathematiker Alan Turing
- Simuliert die Arbeitsweise eines Computers
  - besonders einfach und mathematisch gut zu analysieren
- Eine Turingmaschine macht die Berechenbarkeit eines Algorithmus mathematisch erfassbar
- D.h. eine Turingmaschine ist ein mathematisches Objekt und kann mit mathematischen Methoden untersucht werden

TURINGMASCHINE

Es gilt:

- Eine Turingmaschine repräsentiert einen Algorithmus bzw. ein Programm
- Eine Berechnung mit einer Turingmaschine erfolgt
  - über die (schrittweise) Manipulation von Zahlen/Symbolen
  - Regeln bestimmen wie diese Zeichen geschrieben bzw. gelesen werden.

Ein Programm das bzw. ein Algorithmus der anhand einer Turingmaschine berechnet werden kann, wird **Turing-berechenbar** oder **einfach berechenbar** genannt.

TURINGMASCHINE

Im Steuerwerk der Turingmaschine befindet sich das Programm/der Algorithmus.

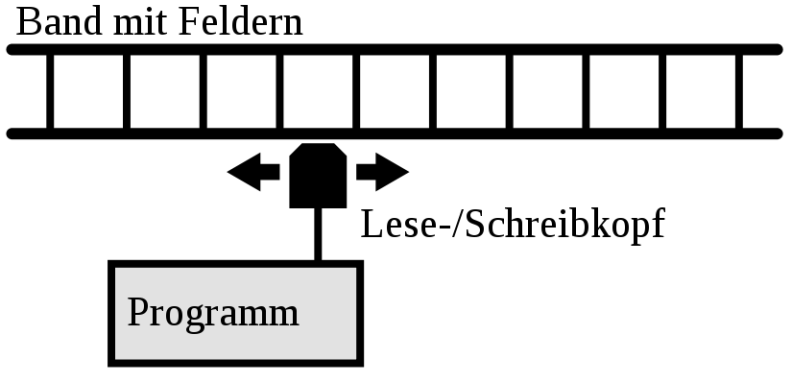
Jede Turingmaschine hat außerdem ein unendlich langes Speicherband mit unendlich vielen Feldern.

Pro Feld kann genau ein Zeichen gespeichert werden.

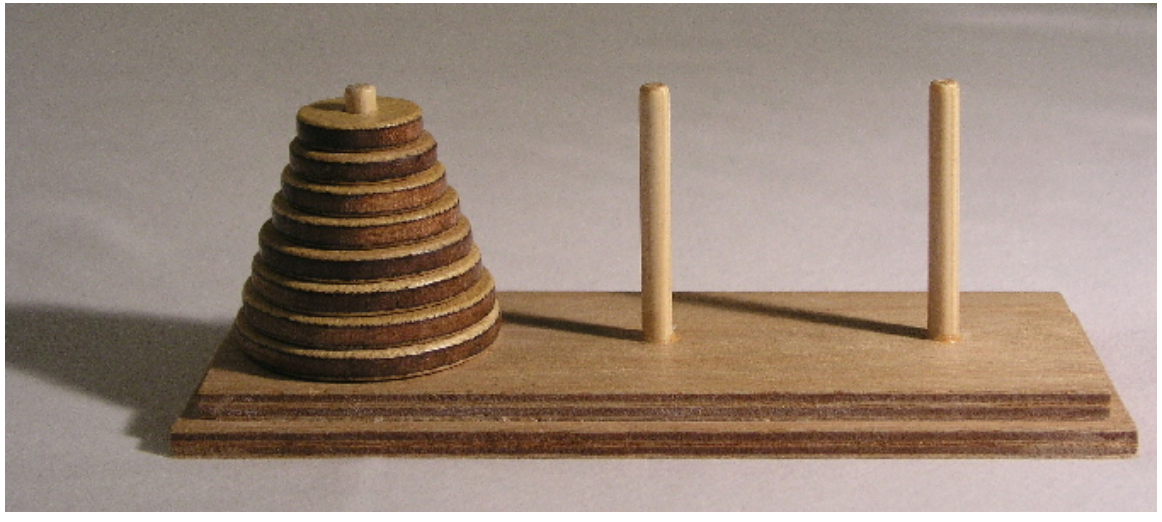
Zusätzlich gibt es ein definiertes Eingabealphabet und ein Symbol für 'leere Felder'.

Der Lese- und Schreibkopf ist Programmgesteuert und bewegt sich feldweise, entsprechend der Regeln des Programmes, auf dem Speicherband und kann dort die Zeichen verändern.

TURINGMASCHINE



TÜRME VON HANOI



TÜRME VON HANOI

Spielregeln

- Turm muss von A nach C (B = Hilfsstapel)
- nur oberste Scheiben dürfen bewegt werden
- Scheiben dürfen nur auf leere Stäbe, oder auf grösseren Scheiben zu liegen kommen

TÜRME VON HANOI

Von Aka - Eigenes Werk, CC BY-SA 2.5, Link

TÜRME VON HANOI

ALGORITHMUS

```
n=1 Scheibe
S1-AC

n=2 Scheiben
S1-AB, S2-AC, S1-BC

n=3 Scheiben
S1-AC, S2-AB, S1-CB, S3-AC, S1-BA, S2-BC, S1-AC
```

TÜRME VON HANOI

```
def hanoi(ring,stab1="stab1",stab2="stab2",stab3="stab3"):
    if ring > 0:
        hanoi(ring-1,stab1,stab3,stab2)
        move(ring,stab1,stab3)
        hanoi(ring-1,stab2,stab1,stab3)
        sleep(1)
```

TÜRME VON HANOI

```
bewege(3,a,b,c) {
    bewege(2,a,c,b) {
        bewege(1,a,b,c) {
            bewege(0,a,c,b){};
            verschiebe oberste Scheibe von a nach c;
            bewege(0,b,a,c){};
        };
        verschiebe   oberste Scheibe   von a nach b;
        bewege(1,c,a,b){
            bewege(0,c,b,a){};
            verschiebe oberste Scheibe von c nach b;
            bewege(0,a,c,b){};
        };
    };
    verschiebe     oberste Scheibe     von a nach c;
    bewege(2,b,a,c){
        bewege(1,b,c,a){
            bewege(0,b,a,c){};
            verschiebe oberste Scheibe von b nach a;
            bewege(0,c,b,a){};
        };
        verschiebe   oberste Scheibe   von b nach c;
        bewege(1,a,b,c){
            bewege(0,a,c,b){};
            verschiebe oberste Scheibe von a nach c;
            bewege(0,b,a,c){};
        };
    };
};
```

Quelle: Randolffscher Algorithmus

LAUFZEIT TÜRME VON HANOI

PRAKITSCHE (UN)LÖSBARKEIT

1 Sekunde / Zug:

Scheiben	Züge	Zeit
1	1	1 s
5	31	31 s
10	1.023	17 min
20	1.048.575	12 d
30	1.073.741.823	34 a

MODULE IN PYTHON

- helfen Programme aufzuteilen
- Programmcode kann wieder verwendet werden
- einzubinden über `import`
- externe Module/Bibliotheken können importiert werden

MODULE IN PYTHON

- Bsp:
  - `math`
    - `sin(),cos(),log()`
    - `pi,e`
  - `os`
    - `listdir(),name(),...`
  - `datetime`
  - `random`

EINBINDEN EIGENER MODULE

- Bsp:
  - Helfer Funktionen in `myfunctions.py`
  - `import myfunctions`
    - stellt Funktionen von `myfunctions.py` bereit

```
# myfunctions.py
def helper(test):
    return test*100
```

```
# myprogram.py
import myfunctions

print(myfunctions.helper(10))
```

EINBINDEN EIGENER MODULE

```
# myfunctions.py
def helper(test):
    return test*100
```

```
# myprogram.py
import myfunctions as mf

print(mf.helper(10))
```

MODULE ZU PAKETEN ZUSAMMENFASSEN

- Ordner mit Paketnamen
- Ein File für jedes Modul

```
mypackage/
  __init__.py # optional
  myfunctions.py
  module2.py
  module3.py
```

```
from mypackage import myfunctions
from mypackage import * # alles importieren, setzt __init__.py vorau
```

Siehe Doku

OBJEKTORIENTIERUNG IN PYTHON

OBJEKTORIENTIERUNG IN PYTHON

- Hilft bei der Abstrahierung von Problemstellungen
- Datenstrukturen und dazu passende Funktionen werden zu einem Objekt zusammengefasst

KLASSEN

```
class MyClass:
  globalClassCount = 0
  def __init__(self, initialData):
    self.data = initialData
    MyClass.globalClassCount += 1

  def getData(self):
    return self.data
  def getClassCount(self):
    return MyClass.globalClassCount
```

DEMO

VERERBUNG

- Verbessert die Wiederverwendbarkeit und Erweiterbarkeit von Code
- Basisklasse vererbt Eigenschaften und Fähigkeiten
- Tochterklasse wird um Funktionalität und Eigenschaften erweitert

VERERBUNG

```
class A:
    def __init__(self):
        self.X = 2
        self.Y = 3
        print("Konstruktor in A")
    def getX(self):
        return self.X
class B(A):
    def getArea(self):
        return self.X * self.Y

b = B()
print(b.getArea())
print(b.getX())
```

```
Konstruktor in A
6
2
```

FRAGEN?

NÄCHSTES MAL

2016-12-14 16:00