Algorithmen & Datenstrukturen

Julian Huber & Matthias Panny

Beispiel: Wahl der Jahrgangsvertretung

- Wir möchten die Jahrgangsvertretung wählen, der Wahlsieger kann nur durch eine absolute Mehrheit bestimmt werden
- Es gibt 42 mögliche Wählende und 3 Kandidaten, es haben 38 Personen von ihrem Wahlrecht Gebrauch gemacht
- Die abgegebenen Stimmen sind in folgender Liste zusammengefasst:

```
votes = ['Matthias', 'Julian', 'Franz', 'Julian', 'Julian', 'Matthias', 'Julian',
'Franz', 'Julian', 'Matthias', 'Matthias', 'Franz', 'Matthias', 'Julian',
'Franz', 'Franz', 'Matthias', 'Franz', 'Franz', 'Franz', 'Matthias', 'Franz',
'Julian', 'Franz', 'Matthias', 'Franz', 'Julian', 'Julian', 'Julian',
'Julian', 'Franz', 'Julian', 'Julian', 'Matthias', 'Franz',
'Franz', 'Franz', 'Matthias']
```

- Wir können nun durch die Liste iterieren und z.B. jede Stimme mit allen anderen vergleichen
- Ist dieser Ansatz noch gangbar wenn bei der österreichischen Nationalratswahl 4.929.745 Stimmen für 12 Parteien abgegeben wurden?

Analyse von Algorithmen

- Algorithmus liefert für definierten Input in endlicher Anzahl an Schritten einen definierten Output
- Es gibt viele mögliche Implementierungen für einen allgemeinen Algorithmus *vgl. Lösung des Rucksackproblems*
- Es müssen also Bewertungskriterien definiert werden

Bewertungskriterien

- Geschwindigkeit
- Speicherbedarf
- Komplexität in Implementierung & Wartung
- Erweiterbarkeit
- etc.

Geschwindigkeit → Zeitkomplexität

- Simpler Ansatz: Wir messen die Laufzeit die unser Algorithmus braucht
- Dieser empirische Ansatz ist abhängig von z.B. der Hardware, der Programmiersprache, der genauen Implementierung, der aktuellen Systemauslastung, etc.

Beispiel - `simple_benchmark.py`

- `perf_counter()` aus dem `time` Modul liefert relativen Zeitstempel mit der höchsten verfügbaren Auflösung

```python
import time
start_time = time.perf_counter()

# Algorithmus
sum = 0
for i in range(0, 10000):
  sum += i**2

end_time = time.perf_counter()
total_time = end_time - start_time
print(F"Took {(total_time*1e3):.4f} milliseconds")

#Took 1.7423 milliseconds
#Took 1.5552 milliseconds
#Took 1.6228 milliseconds
#Took 1.5278 milliseconds
```

Zeitkomplexität

- Vergleich so nur sinnvoll wenn wir die selbe Hardware, Programmiersprache, Systemauslastung, etc. garantieren können

Alternativer Ansatz:

- *Robustere Idee:* Wir zählen die Anzahl der Operationen die unser Algorithmus ausführt
- Wir definieren eine Funktion $f(n)$ die die Anzahl der Operationen in Abhängigkeit der Eingabegröße n angibt
- Operationen in `simple_benchmark.py`:
  - Initialisierung von `sum = 0`
  - Quadrieren von `i` in jedem Schleifendurchlauf (n mal)
  - Addition von `i**2` zu `sum` in jedem Schleifendurchlauf (n mal)
  - $f(n) = 1 + n(1 + 1)$ für n Schleifendurchläufe

✍️ Aufgabe

- Bestimmen Sie die Anzahl an Operationen $f(n)$ für folgende Funktion aus `contains_duplicates.py`
- Was wird hier sinnvollerweise mit n bezeichnet?

```python
def contains_duplicates(array):
  for i, outer in enumerate(array):
    for j, inner in enumerate(array):
      if i != j and outer == inner:
        return True
  return False

array = [0, 2, 5, 8, 3, 5, 3, 1, 7, 9]
has_dupes = contains_duplicates(array)
# has_dupes = True
```

Zeitkomplexität

- Offene Fragen in der Definition von $f(n)$:
  - Wie definieren wir n?
  - Welche Operationen zählen wir, z.B. wenn `if`-Anweisungen vorkommen
- Exakte Bestimmung von $f(n)$ ist oft nicht möglich

neuer Ansatz:

- Verhalten für große n ist entscheidend $\rightarrow$ exakte Bestimmung von $f(n)$ nicht erforderlich!
- Nur mehr Programmteile mit dominierender Komplexität werden gezählt: z.B.: $f(n) = n^3 + 2n^2 + 5n + 1 \rightarrow f(n) \approx n^3$
- Wird ausgedrückt mit $\mathcal{O}(n)$ (Landau-Symbol od. Big-O-Notation)
- Math. präziser:
  - $\mathcal{O}$ gibt asymptotische obere Schranke für f an
  - $\mathcal{O}(g(n)) = f(n) \rightarrow f(n)$ wächst höchstens so schnell wie $g(n)$

Zeitkomplexität

- Hieraus ergeben sich bestimmte Komplexitätsklassen die üblicherweise angegeben werden

Übliche Komplexitätsklassen:

- $\mathcal{O}(1)$: konstante Laufzeit
- $\mathcal{O}(\log(n))$: logarithmische Laufzeit *
- $\mathcal{O}(n)$: lineare Laufzeit
- $\mathcal{O}(n \log(n))$: linearithmische Laufzeit *
- $\mathcal{O}(n^2)$: quadratische Laufzeit
- $\mathcal{O}(n^m)$: polynomielle Laufzeit
- $\mathcal{O}(2^n)$: exponentielle Laufzeit

* Im Allgemeinen wird hier mit $\log$ der Logarithmus zur Basis 2 gemeint $\rightarrow \log_2$

Übliche Komplexitätsklassen

- Primitive Operationen: $\mathcal{O}(1)$

```
i = i + 1              # O(1)
j = 10 * i             # O(1)
```

- Schleifen über `n`: Komplexität des Schleifenkörpers mal Anzahl der Schleifendurchläufe

```
i = 0
while i < 10:
  i = i + 1            # O(1)
                       # = O(n * 1) = O(n)
```

- Sequenzen von Operationen: Summe (= Maximum) der Komplexität der Sequenz

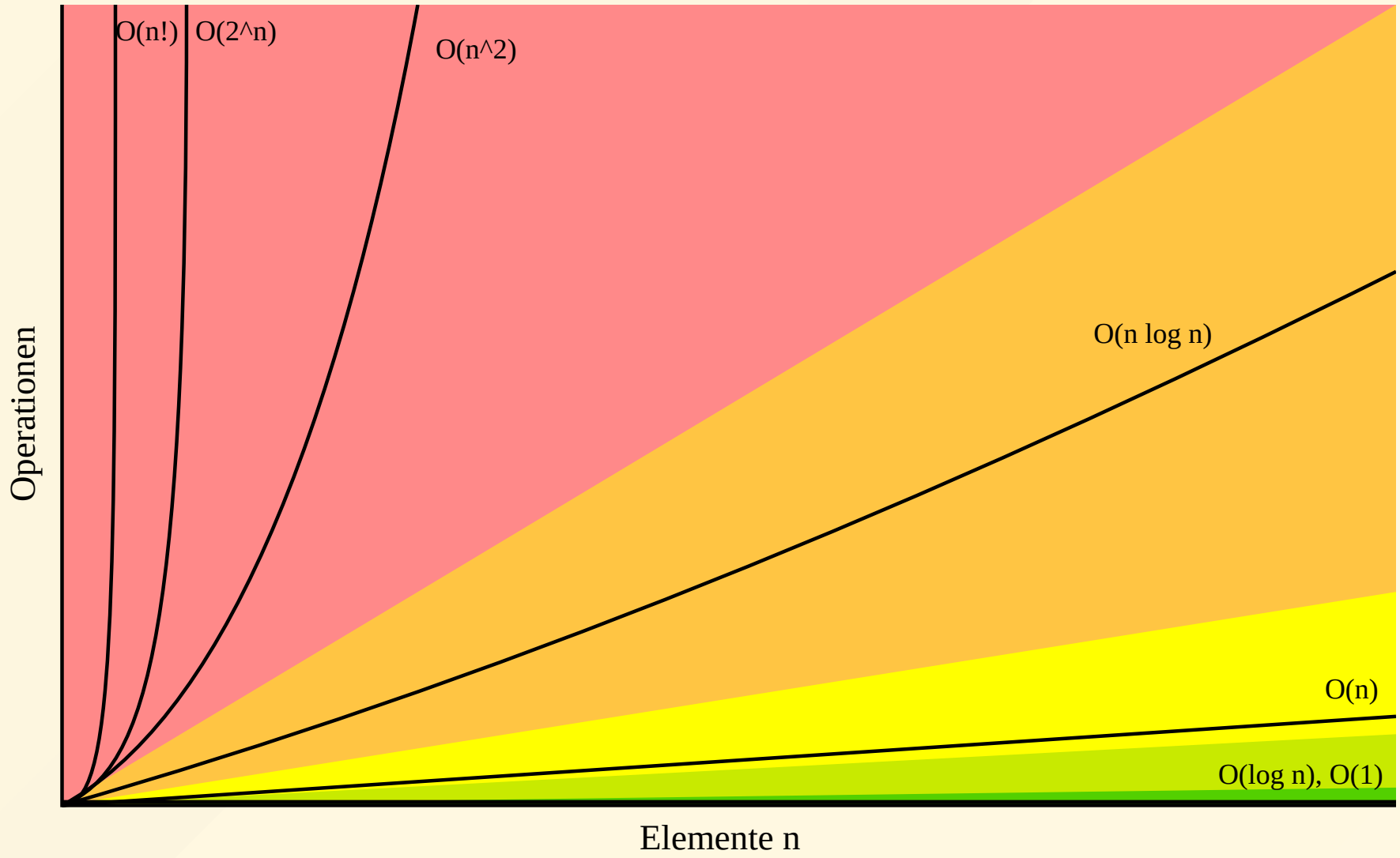
```
i = 0                  # O(1)
while i < 10:
  #do something...     # O(...)
  i = i + 1            # O(n)
j = 10 * i             # O(1)
                       # = O(n) + O(1) + O(...)
```

- Schleifen mit exponentiellem Inkrement

```
i = 1
while i < 10:
  i = i * 2            # O(1)
                       # = O(log(n)), da Inkrement exponentiell
```

Übliche Komplexitätsklassen

- In der Praxis sind die meisten Algorithmen in $\mathcal{O}(1)$, $\mathcal{O}(\log(n))$ oder $\mathcal{O}(n)$



✍️ Aufgabe

- Bestimmen Sie die Komplexität $\mathcal{O}$ für folgende Funktion aus `element_in_list.py`

```python
def element_in_list(array, element):
  for array_elem in array:
    if array_elem == element:
      return True
  return False

array = [0, 2, 5, 8, 3, 4, 6, 1, 7, 9]
has_element = element_in_list(array, 4)
# has_element = True
```

Übliche Komplexitätsklassen

- Vielfach wird noch in die Fälle best-case, average-case und worst-case unterschieden
- Ausgangslage für Analyse faktisch immer der worst-case

Beispiel: Lineare Suche in einem Array

- Best-case:
  - Gesuchtes Element an Index `0` → $\mathcal{O}(1)$
- Average-case:
  - Im Durchschnitt ist das gesuchte Element in der Mitte der Liste → $\mathcal{O}(n/2) = \mathcal{O}(n)$
- Worst-case:
  - Gesuchtes Element nicht in der Liste → $\mathcal{O}(n)$

Zeitkomplexität von Datenstrukturen

- In Algorithmen arbeiten wir für gewöhnlich mit den uns bekannten Datenstrukturen
- Für deren gängige Operationen kann eine Zeitkomplexität angegeben werden
- Kommt auch wieder auf die genaue Implementierung des abstrakten Datentyps an

Beispiele

- `list.insert(0, x)` fügt ein Element `x` am Anfang einer Liste ein
- Die Länge der Liste hat keine Auswirkung auf die Operation: $\mathcal{O}(1)$
- `np.insert(arr, i, x)` fügt `x` an der Position `i` in ein Array ein
- Die Länge des Arrays hat einen Einfluss auf die Operation: $\mathcal{O}(n)$

Zeitkomplexität von Datenstrukturen

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Bildquelle: [Althoff 2021]

Zeitkomplexität

- Praktische Bedeutung von Zeitkomplexität an einem gängigen Algorithmus der Mechatronik

Beispiel DFT vs FFT: `time_complexity_dft.ipynb`

- DFT - Discrete Fourier Transform: $\mathcal{O}(n^2)$
- FFT - Fast Fourier Transform: $\mathcal{O}(n \log(n))$

Auswirkungen

- Schon bei $n = 1000$ ist die FFT um den Faktor 10^2 schneller als die DFT
- Bei einer Samplerate von $44100\,\mathrm{Hz}$ entspricht $n = 1000$ einer verarbeiteten Datenmenge von $22.7\,\mathrm{ms}$
- Echtzeitanwendungen faktisch nur mit FFT möglich

Video on FFT

Platzkomplexität

Platzkomplexität

- Analog zur Zeitkomplexität kann auch die Platzkomplexität $\mathcal{O}$ angegeben werden
- Speicherbedarf (RAM) wird analysiert
- Großer Unterschied: Platz kann wiederverwendet werden, Zeit nicht

Teilbereiche

- Fixer Speicher: für Algorithmus selbst
- Variabler Speicher: für Eingabedaten
- Speicher für temporäre Variablen: für Zwischenergebnisse während der Ausführung

Beispiel:

```
x = 1                          # O(1)
n = 5                          # O(1)
for i in range(1, n + 1): # range is like a generator (saves memory)
  x = x * i                    # O(1), da x immer wieder überschrieben
                               # = O(1)
```

Platzkomplexität

Beispiel time_and_space_complexity_factorial.py

```python
def factorial_iterative(n):
  """Iterative implementation of the factorial function with O(1)."""
  result = 1
  for i in range(1, n + 1):
    result *= i
  return result

def factorial_recursive(n):
  """Recursive implementation of the factorial function with O(n)."""
  if n == 0:
    return 1
  else:
    return n * factorial_recursive(n - 1)
```

🏆 Hausübung

- Definieren und Implementieren eines Algorithmus um zu bestimmen ob in ein Kandidat einer Wahl die absolute Mehrheit hat.