Universität Bremen

Praktische Informatik 2
Komplexität

Thomas Röfer

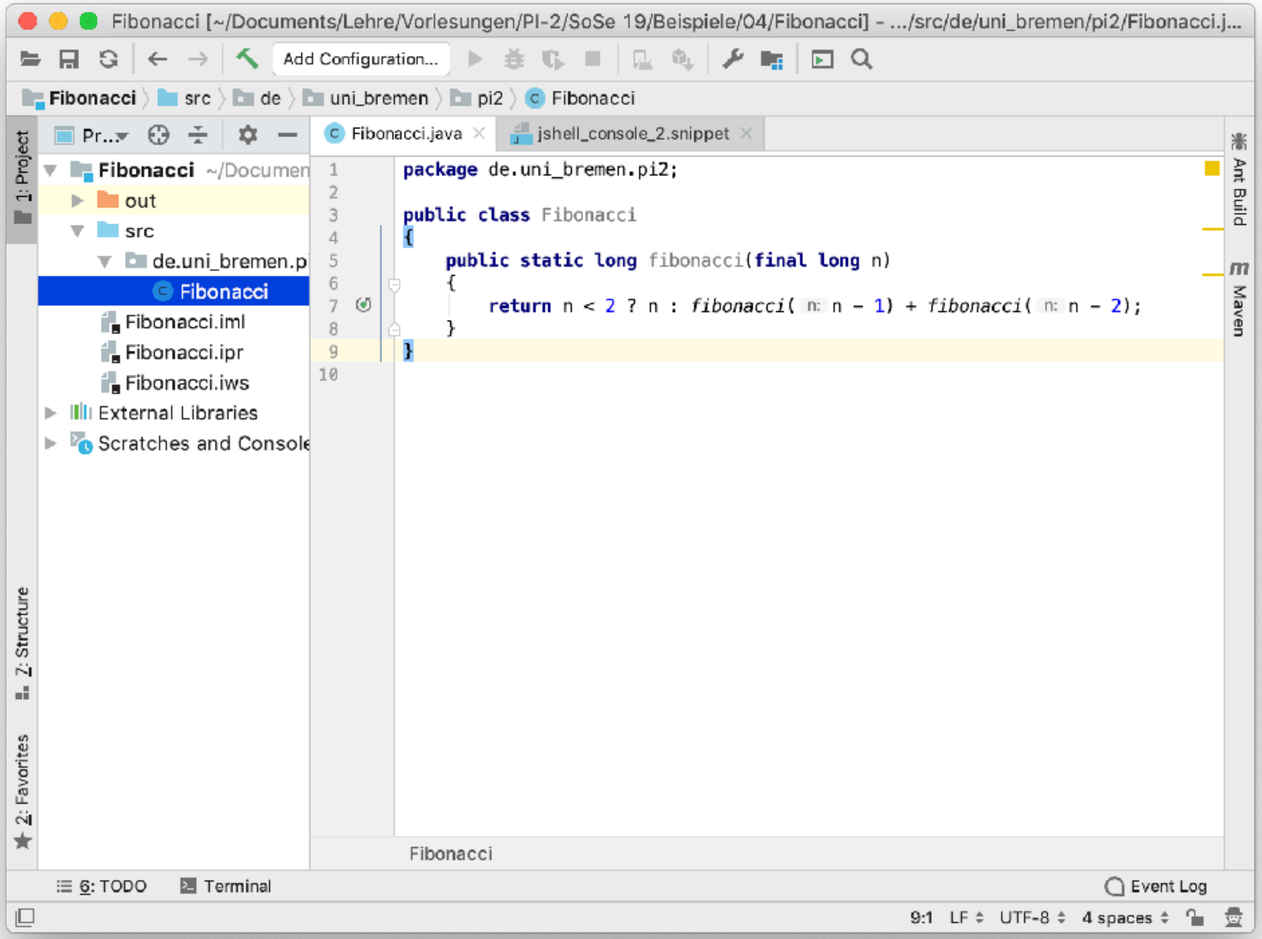
Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

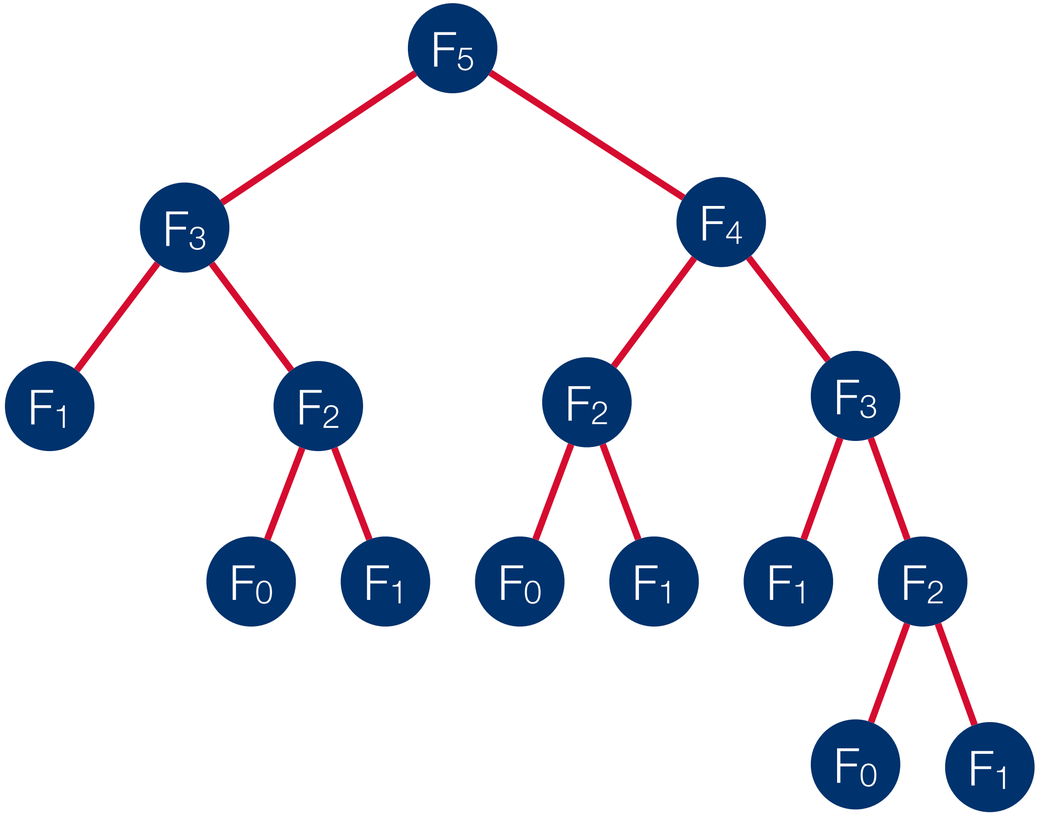
Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

Motivation





Universität Bremen

Komplexität

- Die **Komplexität** eines Algorithmus ist der **Aufwand** in Abhängigkeit von der **Größe** der Eingabe
- **Zeitkomplexität**: Anzahl der benötigten Programmschritte (ist üblicherweise mit „Komplexität“ gemeint)
- **Platzkomplexität**: Größe des benötigten Speicherplatzes
- Zu betrachtende Fälle
  - Schlechtester Fall (worst case)
  - Bester Fall (best case)
  - Durchschnittlicher Fall (average case)

```
int sqr(final int x)
{
    return x * x;
}

int factorial(final int n)
{
    return n == 0  ? 1 :
            n * factorial(n - 1);
}
```

Exakte Bestimmung

n	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(final int n)
{
    int res = 1;
    for (int j = 1; j < n; ++j) {
        for (int i = 1; i < j; ++i) {
            res = res * i;
        }
    }
    return res;
}
```

$$
\begin{aligned}
M(n) &= (n-2) + M(n-1) \\
&= (n-2) + (n-3) + M(n-2) \\
&= \sum_{k=1}^{n-2} k \\
&= \frac{(n-1)(n-2)}{2}
\end{aligned}
$$

Exakte Bestimmung

n	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(final int n)
{
    int res = 1;
    for (int j = 1; j < n; ++j) {
        for (int i = 1; i < j; ++i) {
            res = res * i;
        }
    }
    return res;
}
```

$$I(n) = (n-2) + 1 + I(n-1)$$

$$= \sum_{k=1}^{n-1} k$$

$$= \frac{n(n-1)}{2}$$

Exakte Bestimmung

n	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(final int n)
{
    int res = 1;
    for (int j = 1; j < n; ++j) {
        for (int i = 1; i < j; ++i) {
            res = res * i;
        }
    }
    return res;
}
```

$$Z(n) = 1 + 1 + I(n)$$

$$= 2 + \frac{n(n-1)}{2}$$

Exakte Bestimmung

n	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(final int n)
{
    int res = 1;
    for (int j = 1; j < n; ++j) {
        for (int i = 1; i < j; ++i) {
            res = res * i;
        }
    }
    return res;
}
```

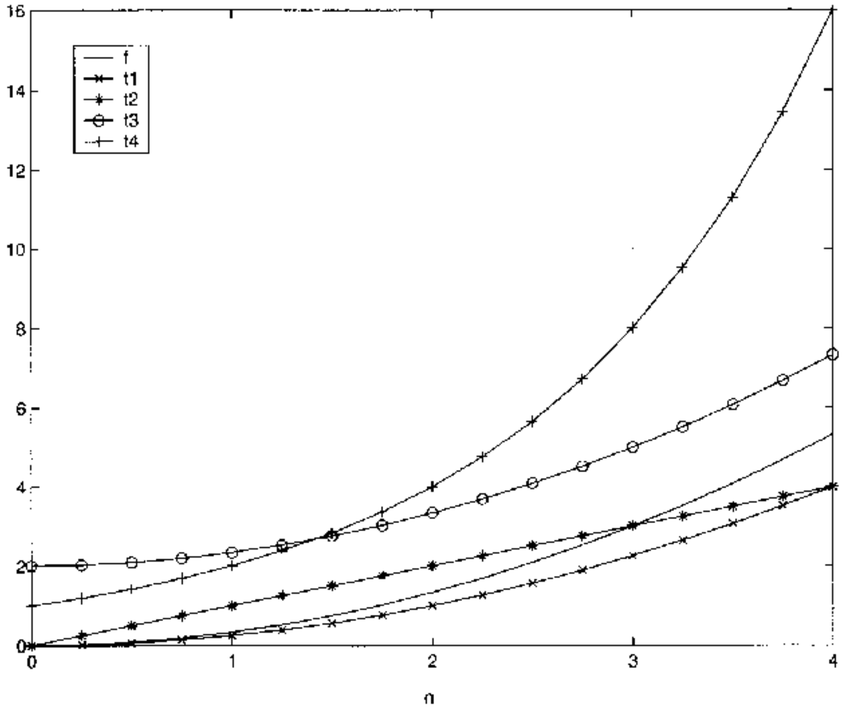
$$V(n) = n + V(n-1)$$

$$= \sum_{k=1}^{n} k$$

$$= \frac{(n+1)n}{2}$$

$$G(n) = M(n) + I(n) + Z(n) + V(n)$$

Asymptotische Komplexität



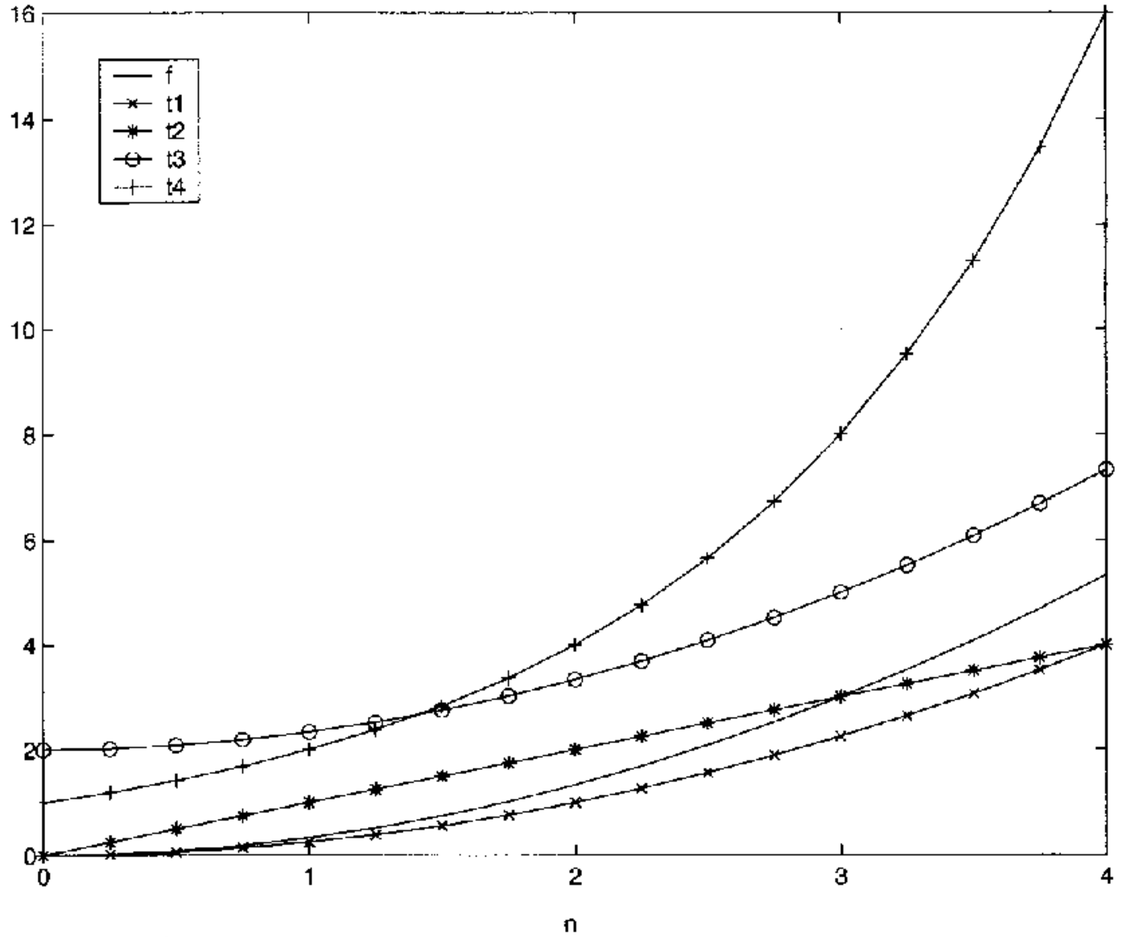
- Sei $f : \mathbb{N} \to \mathbb{R}_0^+$
- Die **Ordnung von f** ist die Menge aller Funktionen, für die $f(n)$, multipliziert mit einer festen Konstante c, ab einer Mindesteingabegröße n_0, immer größer oder gleich ist:

$$O(f(n)) = \{t : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$$

- $O(f(n))$ charakterisiert das **Wachstum von f**
- Sie enthält alle Funktionen, deren Graph maximal so stark wächst wie der Graph von f (bis auf Umbezeichnung der Einheiten auf der y-Achse)

Asymptotische Komplexität: Beispiele

- $f(n) = \frac{1}{3}\, n^2$
- $t_1(n) = \frac{1}{4}\, n^2$
  $\in O(f(n)),\ n_0 = 1,\ c = 1$
- $t_2(n) = n$
  $\in O(f(n)),\ n_0 = 3,\ c = 1$
- $t_3(n) = \frac{1}{3}\, n^2 + 2$
  $\in O(f(n)),\ n_0 = 3,\ c = 2$
- $t_4(n) = 2^n$



$$O(f(n)) = \{t : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$$

Universität Bremen

Beweise

- $n^2 \in O(n^3)$?
  - Zu zeigen: $\exists c \in \mathbb{R}^+$ und $\exists n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ gilt: $n^2 \leq c \cdot n^3$
  - Dies gilt z.B. ab $c = 1$, $n_0 = 1$, da $1 \leq 1 \cdot n$
- $n^3 \in O(n^2)$?
  - Zu zeigen: $\exists c \in \mathbb{R}^+$ und $\exists n_0 \in \mathbb{N}$ so dass für alle $n \geq n_0$ gilt: $n^3 \leq c \cdot n^2$
  - Dies erfordert ein c, so dass für alle n gilt: $n \leq c$
  - Ein solches c kann es nicht geben, daher gilt: $n^3 \notin O(n^2)$

Universität Bremen

Weitere Aussagen

- $f(n) \cdot g(n) \in O(f(n) \cdot g(n))$
  - Für die Aufwandsbestimmung bedeutet dies, dass ineinander verschachtelte Aufwände multipliziert werden
  - Achtung: Wenn der Aufwand in einem Schleifenrumpf nicht immer in derselben Aufwandsklasse liegt, sollten die verschiedenen Aufwände getrennt betrachtet werden
- $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$
  - Für die Aufwandsbestimmung bedeutet dies, dass bei hintereinander anfallenden Aufwänden nur der größte den Gesamtaufwand bestimmt

Universität Bremen

Bestimmung des Aufwands

- *int res = 1*, *int j = 1*, *j < n*, *++j*, *int i = 1*, *i < j*, *++i*, *res = res * i*, *return res*: Alle $O(1)$
- Rumpf innere Schleife: $O(1 + 1 + 1) = O(\max\{1, 1, 1\}) = O(1)$
- Innere Schleife: $O(1 \cdot (j - 1)) = O(\max\{j, -1\}) = O(j)$
- Rumpf äußere Schleife: $O(1 + 1 + 1 + j) = O(\max\{1, 1, 1, j\}) = O(j)$
- Äußere Schleife: $O(\sum_{j=1}^{n-1} j) = O(\frac{1}{2}(n^2 - n)) = O(n^2 - n) = O(\max\{n^2, -n\}) = O(n^2)$
- Gesamte Funktion: $O(1 + 1 + n^2 + 1) = O(\max\{1, 1, n^2, 1\}) = O(n^2)$

```
int f1(final int n)
{
    int res = 1;
    for (int j = 1; j < n; ++j) {
        for (int i = 1; i < j; ++i) {
            res = res * i;
        }
    }
    return res;
}
```

Universität Bremen

Typische Aufwandsklassen

- Konstant: $O(1)$, z.B. **sqr(n)**
- Logarithmisch: $O(\log n)$, z.B. binäre Suche
- Linear: $O(n)$, z.B. **factorial(n)**
- Linearithmisch: $O(n \log n)$, z.B. Sortieren mit Divide and Conquer
- Quadratisch, polynomial: $O(n^2)$, $O(n^3)$ …, allgemein: $O(n^c)$, $c \in \mathbb{R}^+$, z.B. **f1(n)**
- Exponentiell: $O(2^n)$, $O(3^n)$ …, allgemein: $O(c^n)$, $c \in \mathbb{R}^+$, z.B. **rucksack(n**: Gepäckanzahl**)**
- Faktoriell: $O(n!)$, z.B. Rundreise-Problem

Universität Bremen

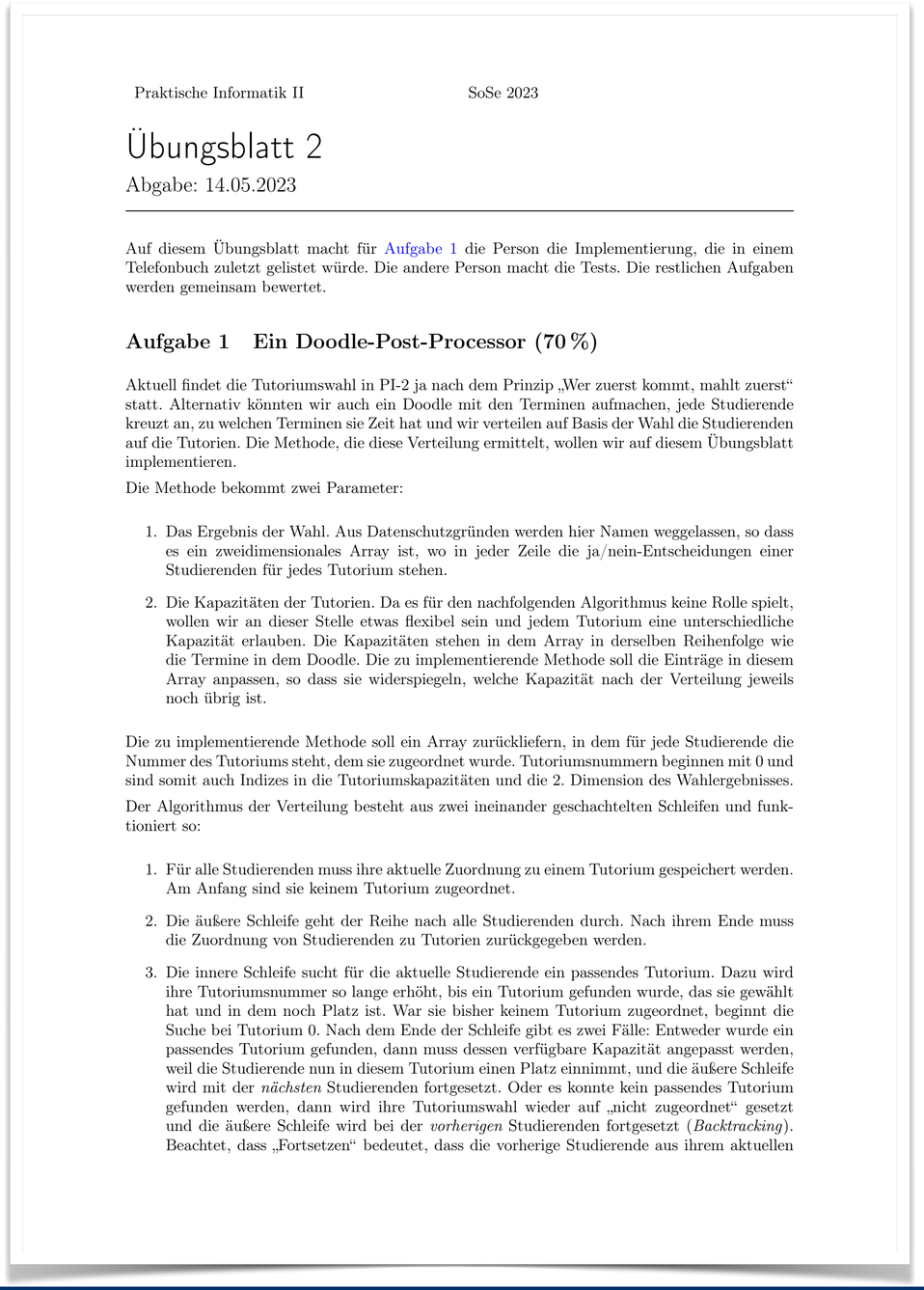
Zusammenfassung der Konzepte

- **Zeitkomplexität** und **Platzkomplexität**
- **Asymptotische Komplexität**
- **O-Kalkül**
- **Typische Aufwandsklassen**

Universität Bremen

Übungsblatt 2

- Aufgabe 1: Backtracking
  - Zuordnung von Studierenden zu Tutorien mit Wahlmöglichkeit
- Aufgabe 2: Warum **@Disabled**?
- Aufgabe 3: Worst-Case-Aufwand von Aufgabe 2
- Aufgabe 4: O-Kalkül-Aufwand von **Array** rückwärts besser als vorwärts?



Praktische Informatik II SoSe 2023

Übungsblatt 2

Abgabe: 14.05.2023

Auf diesem Übungsblatt macht für Aufgabe 1 die Person die Implementierung, die in einem Telefonbuch zuletzt gelistet würde. Die andere Person macht die Tests. Die restlichen Aufgaben werden gemeinsam bewertet.

Aufgabe 1 Ein Doodle-Post-Processor (70 %)

Aktuell findet die Tutoriumswahl in PI-2 ja nach dem Prinzip „Wer zuerst kommt, mahlt zuerst“ statt. Alternativ könnten wir auch ein Doodle mit den Terminen aufmachen, jede Studierende kreuzt an, zu welchen Terminen sie Zeit hat und wir verteilen auf Basis der Wahl die Studierenden auf die Tutorien. Die Methode, die diese Verteilung ermittelt, wollen wir auf diesem Übungsblatt implementieren.

Die Methode bekommt zwei Parameter:

1. Das Ergebnis der Wahl. Aus Datenschutzgründen werden hier Namen weggelassen, so dass es ein zweidimensionales Array ist, wo in jeder Zeile die ja/nein-Entscheidungen einer Studierenden für jedes Tutorium stehen.
2. Die Kapazitäten der Tutorien. Da es für den nachfolgenden Algorithmus keine Rolle spielt, wollen wir an dieser Stelle etwas flexibel sein und jedem Tutorium eine unterschiedliche Kapazität erlauben. Die Kapazitäten stehen in dem Array in derselben Reihenfolge wie die Termine in dem Doodle. Die zu implementierende Methode soll die Einträge in diesem Array anpassen, so dass sie widerspiegeln, welche Kapazität nach der Verteilung jeweils noch übrig ist.

Die zu implementierende Methode soll ein Array zurückliefern, in dem für jede Studierende die Nummer des Tutoriums steht, dem sie zugeordnet wurde. Tutoriumsnummern beginnen mit 0 und sind somit auch Indizes in die Tutoriumskapazitäten und die 2. Dimension des Wahlergebnisses.

Der Algorithmus der Verteilung besteht aus zwei ineinander geschachtelten Schleifen und funktioniert so:

1. Für alle Studierenden muss ihre aktuelle Zuordnung zu einem Tutorium gespeichert werden. Am Anfang sind sie keinem Tutorium zugeordnet.
2. Die äußere Schleife geht der Reihe nach alle Studierenden durch. Nach ihrem Ende muss die Zuordnung von Studierenden zu Tutorien zurückgegeben werden.
3. Die innere Schleife sucht für die aktuelle Studierende ein passendes Tutorium. Dazu wird ihre Tutoriumsnummer so lange erhöht, bis ein Tutorium gefunden wurde, das sie gewählt hat und in dem noch Platz ist. War sie bisher keinem Tutorium zugeordnet, beginnt die Suche bei Tutorium 0. Nach dem Ende der Schleife gibt es zwei Fälle: Entweder wurde ein passendes Tutorium gefunden, dann muss dessen verfügbare Kapazität angepasst werden, weil die Studierende nun in diesem Tutorium einen Platz einnimmt, und die äußere Schleife wird mit der *nächsten* Studierenden fortgesetzt. Oder es konnte kein passendes Tutorium gefunden werden, dann wird ihre Tutoriumswahl wieder auf „nicht zugeordnet“ gesetzt und die äußere Schleife wird bei der *vorherigen* Studierenden fortgesetzt (*Backtracking*). Beachtet, dass „Fortsetzen“ bedeutet, dass die vorherige Studierende aus ihrem aktuellen