*

Algorithmen und Datenstrukturen

– Sommersemester 2019 –

Kapitel 02: Komplexitätsanalyse

Prof. Dr. Adrian Ulges

B.Sc. AI / ITS / WI
Fachbereich DCSM
Hochschule RheinMain

Kosten von Algorithmen

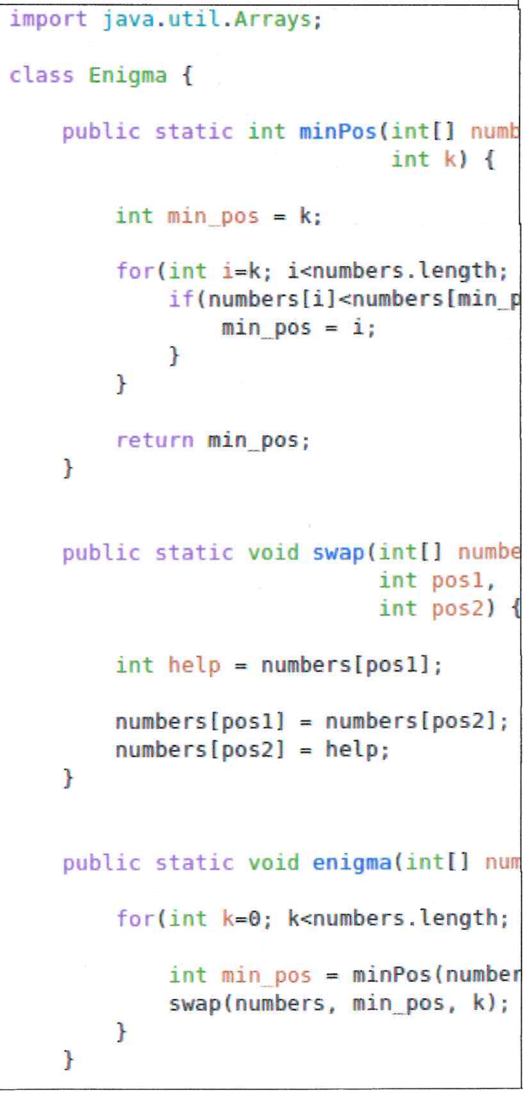
Wieviele Ressourcen (Laufzeit/Speicher) benötigt ein Algorithmus?

Ansätze

1. **Benchmarking**: Implementiere den Algorithmus in einer Programmiersprache und teste ihn mit verschiedenen Eingaben.
2. **Zählen der Elementaroperationen** des Algorithmus, Ableitung einer Kostenformel.

Nachteile von Benchmarking?

- Benchmarking-Ergebnisse sind abhängig von Kontextfaktoren *(Hardware, Sprache, Compiler, Implementierungsdetails, Last)*.
- In der Regel sind nicht alle möglichen Eingaben testbar.



```
import java.util.Arrays;

class Enigma {

    public static int minPos(int[] numb
                             int k) {

        int min_pos = k;

        for(int i=k; i<numbers.length;
            if(numbers[i]<numbers[min_p
                min_pos = i;
            }
        }

        return min_pos;
    }


    public static void swap(int[] numbe
                            int pos1,
                            int pos2) {

        int help = numbers[pos1];

        numbers[pos1] = numbers[pos2];
        numbers[pos2] = help;
    }


    public static void enigma(int[] num

        for(int k=0; k<numbers.length;

            int min_pos = minPos(number
            swap(numbers, min_pos, k);
        }
    }
```

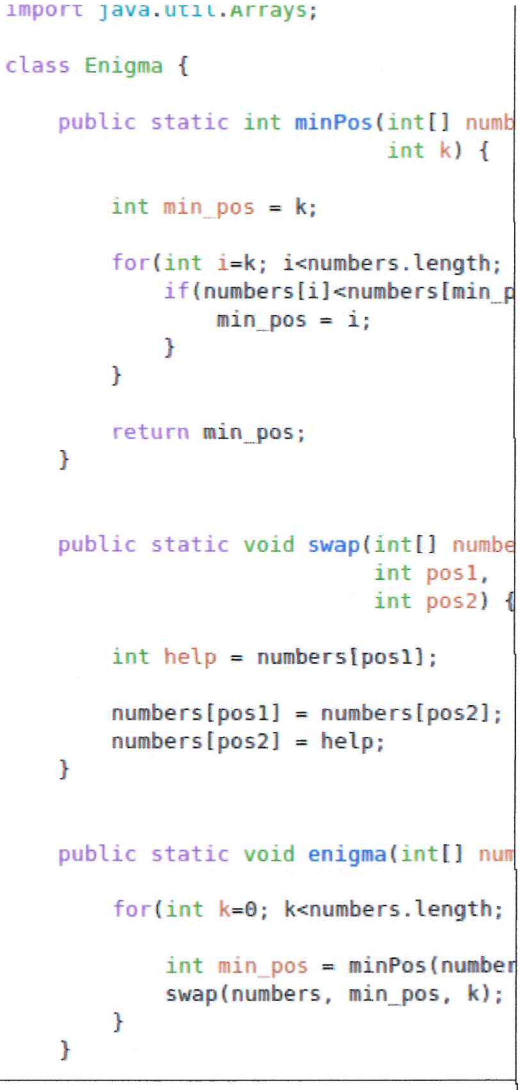
Kosten von Algorithmen

In ADS verfolgen wir Ansatz 2:

- Wir führen den Algorithmus gedanklich auf einer Maschine mit bestimmten Kosten für verschiedene Operationen aus.
- Wir zählen bestimmte Einzelschritte *(Feldzugriffe, Additionen, Vergleiche, ...)*
- **Schlüsselfrage**: Wie verhält sich der Algorithmus für große Eingaben?

Vorteile dieser Kostenschätzung

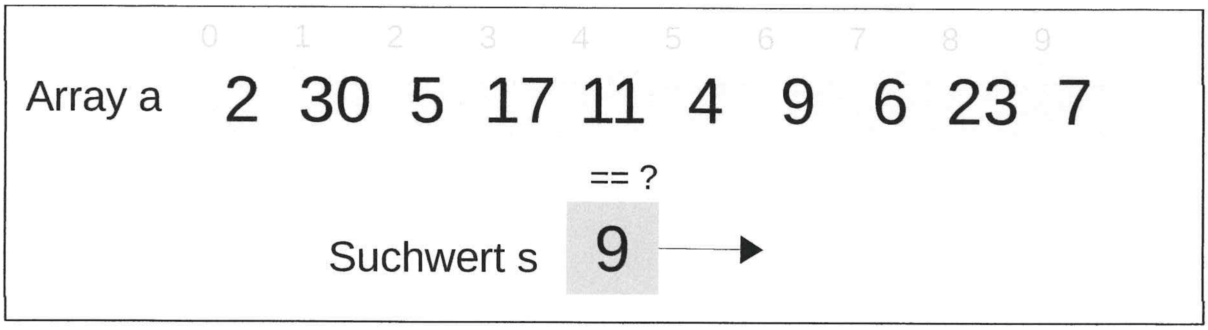
- Generelle Aussage, unabhängigkeit von Plattform+Implementierung.
- Betrachtung aller möglicher Eingaben.
- Aufwandsfrei *(keine Implementierung, kein Testen)*.



```
import java.util.Arrays;

class Enigma {

    public static int minPos(int[] numb
                             int k) {

        int min_pos = k;

        for(int i=k; i<numbers.length;
            if(numbers[i]<numbers[min_p
                min_pos = i;
            }
        }

        return min_pos;
    }


    public static void swap(int[] numbe
                            int pos1,
                            int pos2) {

        int help = numbers[pos1];

        numbers[pos1] = numbers[pos2];
        numbers[pos2] = help;
    }


    public static void enigma(int[] num

        for(int k=0; k<numbers.length;

            int min_pos = minPos(number
            swap(numbers, min_pos, k);
        }
    }
```

Outline

Beispiel: Lineare Suche



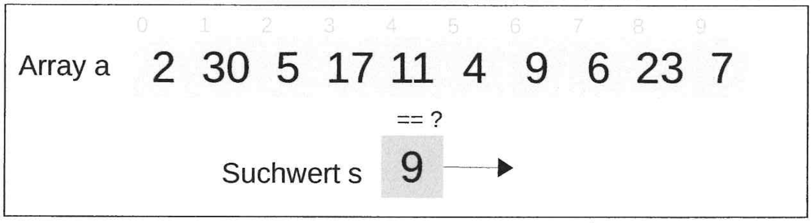
Problemstellung

- Gegeben: Ein Array $a[0], a[1], \ldots, a[n-1]$, ein Suchwert s.
- Gebe die Position zurück, an der der Suchwert im Array vorkommt. Ist der Wert nicht vorhanden, gebe n zurück.

Ansatz

- Durchlaufe das Array von links nach rechts mit Variable `pos`.
- Breche ab, falls `a[pos]` gleich dem Suchwert ist.

Beispiel: Lineare Suche



Pseucodode

```
1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
```

Kostenanalyse *(Beispiel rechts oben)*

Wir zählen **Vergleiche**, **Additionen**, **Feldzugriffe**, **Zuweisungen**:

- Initiale Zuweisung (Zeile 1): Kosten 1.
- 6 erfolglose Schleifendurchläufe (Zeile 2+3).
- Je Durchlauf: Kosten 5.
  (2 Vergleiche & 1 Feldzugriff (Zeile 2), 1 Addition & 1 Zuweisung (Zeile 3))
- 7. Schleifendurchlauf: Suchwert gefunden, Kosten 3.
  (2 Vergleiche & 1 Feldzugriff (Zeile 2))
- Verlassen der Schleife, Algorithmus ist terminiert.
- **Gesamtkosten: $1 + 6 \cdot 5 + 3 = 34$ Schritte**.

Effizienz von Algorithmen: Formalisierung

Generellere Aussage: Abstrahiere über die Eingabedaten
(a) Umfang: *Wie lang ist das zu durchsuchende Array?*
(b) Schwierigkeit: *Wo befindet sich der Suchwert im Array?*

(a) Umfang: Die Problemgröße

> *Gegeben ein zu lösendes Problem, bezeichnen wir den Umfang der Eingabedaten als* **Problemgröße** $n \in \mathbb{N}$.

Die Problemgröße kann *(je nach Art des zu lösenden Problems)* verschiedene Dinge bezeichnen:

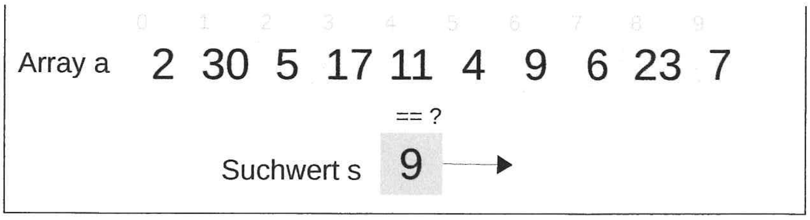
- Die Länge eines Arrays
- Die Anzahl der Knoten in einem Graph
- Die Länge eines kryptografischen Schlüssels in Bit
- Die Anzahl der zu planenden Züge eines Schachcomputers.
- ...

(b) Die Schwierigkeit

Gegeben die Problemgröße *n*, betrachten wir ...

1. den **besten Fall** *(engl. 'best case')*
   - Betrachte die "einfachste" Eingabe (der Größe n), welche die minimal mögliche Anzahl an Schritten verursacht.
   - Dies ist meist nicht besonders interessant.
2. den **mittleren Fall** *(engl. 'average case')*
   - Betrachte alle möglichen Eingaben (der Größe n) und mittle die Anzahl der benötigten Schritte.
   - Dies ist meist relevant, aber schwierig zu berechnen.
3. den **schlechtesten Fall** *(engl. 'worst case')*
   - Betrachte die „schwierigste“ Eingabe (der Größe n) mit der maximal möglichen Anzahl an Schritten.
   - Dies ist meist relevant und leicht zu berechnen.

Beispiel: Lineare Suche



Pseucodode

```
1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
```

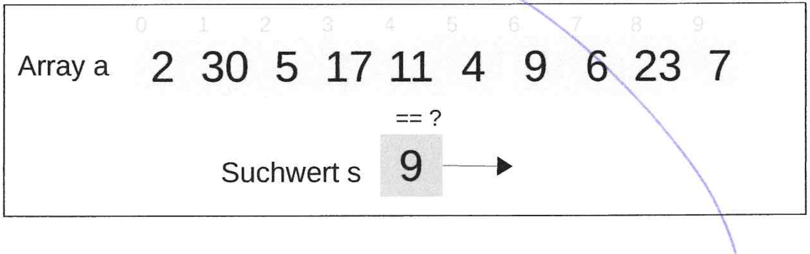
Best Case

- Suchwert befindet sich an der 1. Position im Array.
- Kosten: 4 *(1 Zuweisung (Zeile 1), 2 Vergleiche & 1 Feldzugriff (Zeile 2)*

Worst Case

- Suchwert befindet sich nicht im Array.
- n erfolglose Schleifendurchläufe, jeweils Kosten 5.
- Zusatzkosten: 2 *(1 Zuweisung (Zeile 1), 1 Schleifenabbruch (Zeile 2)*
- **Gesamtkosten:** $2 + 5 \cdot n$.

Beispiel: Lineare Suche



Pseucodode

```
1 pos = 0
2 while pos < n and a[pos] != s:
3     pos = pos+1
4 return pos
```

Average Case

- Annahme: n+1 gleich wahrscheinliche Fälle *(Der Suchwert befindet sich an Position 0, 1, 2, ..., n-1, oder er ist "nicht enthalten")*.

$$\frac{1}{n+1}\cdot\left(\underbrace{4+0\cdot5}_{\text{s an Position 0}} + \underbrace{4+1\cdot5}_{\text{s an Position 1}} + \dots + \underbrace{4+(n-1)\cdot5}_{\text{s an Position n-1}} + \overset{=4+n\cdot5-2}{\underbrace{2+n\cdot5}_{\text{s nicht enthalten}}}\right)$$

Beispiel: Lineare Suche (cont'd)

$$= \frac{1}{n+1} \cdot \left(\left(\sum_{i=0}^{n} 4 + i \cdot 5 \right) - 2 \right)$$

$$= \frac{1}{n+1} \cdot \left((n+1) \cdot 4 + 5 \cdot \sum_{i=0}^{n} i - 2 \right)$$

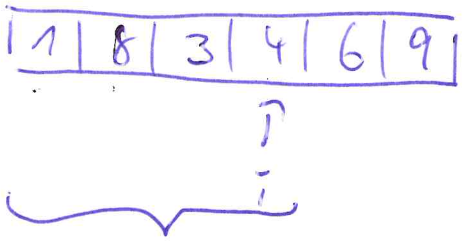
$$= \frac{1}{n+1} \left((n+1) \cdot 4 + 5 \cdot \frac{n \cdot (n+1)}{2} - 2 \right)$$

$$= \frac{1}{n+1} \left(4n + 4 + \frac{5}{2}n^2 + \frac{5}{2}n - 2 \right)$$

$$= \frac{1}{n+1} \cdot \left(\frac{5}{2}n^2 + \frac{13}{2}n + 2 \right) \approx \frac{5/2\, n^2}{\cancel{n+1}} \approx \frac{5}{2}n$$

Aufwandsschätzung: Do-it-Yourself

Berechnen Sie den **Worst-Case-Aufwand** des folgenden Algorithmus. Zählen Sie nur die **Feldzugriffe**.



16 = psum(a,i)

```
# Gegeben: Ein n-elementiges
#          Array a
b := ein n-elementiges Array
for i = 0,..., n-1:
    b[i] = psum(a,i)
return b
```

```
function psum(a,i):
    result = 0
    pos = 0
    while pos <= i:
        result += a[pos]
        pos += 1
    return result
```

psum summiert a[0], a[1], ..., a[i]

⇒ i+1 Feldzugriffe.

Best Case = Worst Case.

Aufwandsschätzung: Do-it-Yourself

Gesamtaufwand: je Durchlauf i (Zeile 4)
- 1 Zugriff b[i] (Zeile 5)
- i+1 Zugriffe in psum.

$$\Rightarrow \sum_{i=0}^{n-1} 1 + \overbrace{(i+1)}^{psum}$$

$$= 2n + \sum_{i=0}^{n-1} i$$

Gauß'sche Summenformel (schon wieder!!)

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

$$= \frac{1}{2}n^2 - \frac{3}{2}n.$$

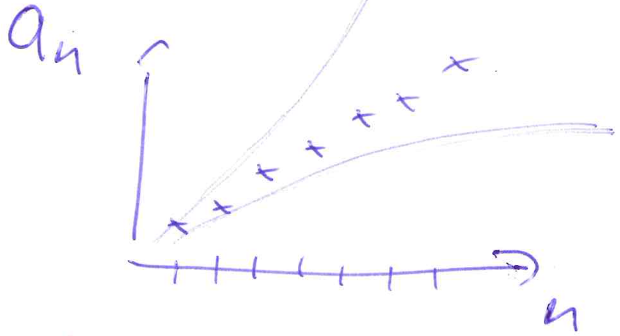
Lineare Suche

$5n+2$

Outline

1. Beispiel: Lineare Suche
2. Die O-Notation
3. Aufwandsabschätzung mit der O-Notation
4. Wichtige Aufwandsklassen
5. Fallbeispiel: Binäre Suche

Kostenfunktion



Kostenfunktionen

Definition (Kostenfunktion)

Gegeben sei ein Algorithmus $\mathcal{A}$. Die **Kostenfunktion** *(oder* **Laufzeit***) $a : \mathbb{N} \to \mathbb{R}^+$ ordnet jeder Problemgröße n den Ressourcenbedarf (z.B. die Anzahl der Operationen) $a(n)$ zu, die $\mathcal{A}$ zur Verarbeitung einer Eingabe der Größe n benötigt.*

Anmerkungen

- Wir können Kostenfunktionen für den Worst/Best/Average Case definieren. Für die lineare Suche gilt z.B. *(siehe oben)*:

$$a^{best}(n) = 4 \qquad a^{worst}(n) = 2 + 5n \qquad a^{avg}(n) = \frac{5/2 \cdot n^2 + 13/2 \cdot n - 2}{n + 1}$$

- Die Kostenfunktion ist eine mathematische Folge:
  Wir können für den Funktionswert a_n oder $a(n)$ schreiben.

Vereinfachung von Kostenfunktionen

Statt der exakten Anzahl der Einzelschritte reicht uns eine grobe Abschätzung. Dies führt zur **O-Notation**, dem zentralen Konzept der Aufwandsschätzung.

Schritt 1: Stärkstes Wachstum

- Wir konzentrieren uns auf den **am stärksten wachsenden Summanden** der Kostenfunktion:

$$4n^2+\xcancel{2n+5} \longrightarrow 4n^2$$

- Warum? Weil für große n der relative Fehler vernachlässigbar ist *(hier für n=10000: 0.005%)*.

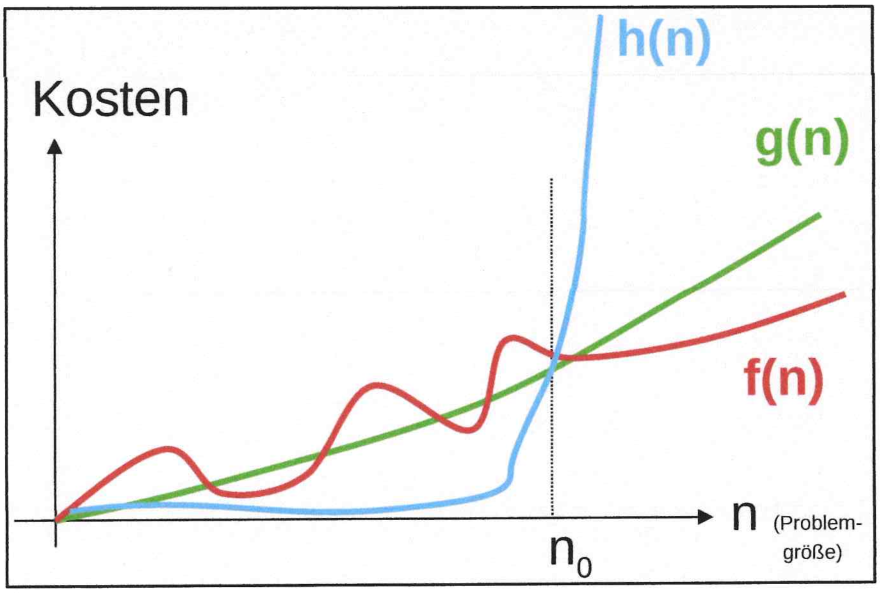
Vereinfachung von Kostenfunktionen

Schritt 2: Faktoren entfernen

$$\cancel{4}\,n^2 \longrightarrow n^2$$

- Konstante Faktoren beeinträchtigen die wichtigsten Aussagen nicht, wie z.B. "Bei einer Verdopplung der Eingabegröße braucht der Algorithmus doppelt so lange".
- Eine Konstante 4 könnte auch durch eine vier mal schnellere Maschine erreicht werden. Diese Details interessieren uns hier nicht *(sondern die generelle Güte eines Algorithmus)*.

O-Notation: Illustration



- f wächst "nicht viel schneller" als g, oder kurz: $f \in O(g)$.
- Es gilt auch: $g \in O(f)$ (g wächst nicht viel schneller als f).
- Es gilt auch: $g \in O(h)$ (g wächst nicht viel schneller als h).
- Es gilt **nicht**: $h \in O(g)$ (h wächst schneller als g).

Definition: O-Notation

Definition (O-Notation)

Es seien f und g zwei Kostenfunktionen. Wenn es eine Konstante $c \in \mathbb{R}$ und ein $n_0 \in \mathbb{N}$ gibt, so dass

$$\mathbf{f(n)\ \leq c \cdot g(n)\ \text{für alle } n \geq n_0},$$

dann schreiben wir $f \in O(g)$ (oder $f(n) \in O(g(n))$).

Anmerkungen

- **Umgangssprachlich** bedeutet $f \in O(g)$: *"f wächst nicht deutlich schneller als g"*.
- $O(g)$ ist demnach die **Menge aller Kostenfunktionen**, die nicht deutlich schneller wachsen als f.
- Wir sprechen: "f ist von der Ordnung g" oder auch **"f ist O von g"**.

Definition: O-Notation

Anmerkungen (cont'd)

- Mit der O-Notation fassen wir ähnliche Algorithmen/Aufwandsfunktionen zu **Klassen** zusammen: Algorithmen, deren Aufwand ähnlich schnell wächst, gehören zur gleichen Klasse (sie besitzen gleiche **Komplexität**).
- Gängig ist auch die Schreibweise $f = O(g)$ (statt $f \in O(g)$). Dies ist aber missverständlich, denn die O-Beziehung ist **nicht symmetrisch**: Aus $n = O(n^2)$ folgt nicht $n^2 = O(n)$.

Definition: O-Notation

Beispiel-Klassen

- "linear": n, $1000n + 3$
- "quadratisch": n^2, $7n^2 + 5n - 10$
- "logarithmisch": $log_2(n)$, $log_3(n)$, $log_8(n) + 4$
- "exponentiell": 2^n, $2^n + n^{10000} + 100000$

Weitere Anmerkungen

Wir unterscheiden im Folgenden zwischen der ...

- **Laufzeit** eines Algorithmus f_n *(= exakte Anzahl an Rechenschritten, umständlich zu berechnen).*
- **Komplexität** $O(f_n)$ (= grobe Abschätzung, leicht zu berechnen, "genau genug").

Man sollte die Komplexität möglichst präzise angeben.
Beispiel: Für $f_n = 2n$ gilt $f_n \in O(2^n)$, aber auch $f_n \in O(n)$ *(besser!).*

Komplexitätsklassen als Mengen

O-Notation: Beweis (Variante 1)

$f_n \le c \cdot g_n$ *

$$\cancel{4}\,n^2+\cancel{2n+5} \longrightarrow n^2$$

Wir zeigen per **vollständiger Induktion**: $\underbrace{4n^2+2n+5}_{f} \in \underbrace{O(n^2)}_{g}$.

Zu zeigen: $f \in O(g)$

$\exists\, \underline{c, n_0}: 4n^2+2n+5 \le c \cdot n^2$ für alle $n \ge n_0$

Wir wählen: $c=5$, $\underline{n_0}=1000$

$\boxed{\text{IA}}$ $4 \cdot 1000^2 + 2 \cdot 1000 + 5 \le 5 \cdot 1000^2$ ✓

$\boxed{\text{IV}}$ Für n gelte: $4n^2+2n+5 \le 5 \cdot n^2$

O-Notation: Beweis (Variante 1)

*

$\boxed{\text{Schritt}}$ $n \rightsquigarrow n+1$

Zu zeigen: $4(n+1)^2+2(n+1)+5 \le 5(n+1)^2$

$$\begin{aligned}
4(n+1)^2+2(n+1)+5 &= 4n^2+8n+4+2n+2+5\\
&= (4n^2+2n+5) + 8n+6\\
&\underset{(IV)}{\le} 5n^2 + 8n+6\\
&\le 5n^2 + 10n+5\\
&= 5(n+1)^2 \checkmark
\end{aligned}$$

$= 5n^2+10n+5$

O-Notation: Theorie

Theorem (Grenzwerte und die O-Notation)

1. *Ist* $\frac{f_n}{g_n}$ *konvergent, folgt* $f_n \in O(g_n)$
2. *Gilt* $\lim_{n\to\infty} \frac{f_n}{g_n} = \infty$, *folgt* $f_n \notin O(g_n)$.

Beweis (zu 1.)

$\frac{f_n}{g_n}$ sei konvergent

→ $\frac{f_n}{g_n}$ ist beschränkt *(siehe Analysis)*.

→ Es gibt eine Schranke $c \in \mathbb{R}$, so dass $\frac{f(n)}{g(n)} \leq c$ für alle $n \in \mathbb{N}$.

→ Es gibt ein $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$, so dass $\frac{f(n)}{g(n)} \leq c$ für alle $n \geq n_0$.

→ $f \in O(g)$.

O-Notation: Beweis (Variante 2)

$$\cancel{4}\, n^2 + \cancel{2n+5} \longrightarrow n^2$$

Wir zeigen per **Folgengrenzwert**: $\underbrace{4n^2+2n+5}_{f} \in \underbrace{O(n^2)}_{g}$.

$$\frac{f}{g} = \frac{4\cancel{n^2}+2n+5}{\cancel{n^2}} = 4 + \underset{\underset{0}{\downarrow}}{\frac{2}{n}} + \underset{\underset{0}{\downarrow}}{\frac{5}{n^2}} \xrightarrow{n\to\infty} 4$$

$\Rightarrow$ f/g ist konvergent.

$\Rightarrow$ $f \in O(g)$.