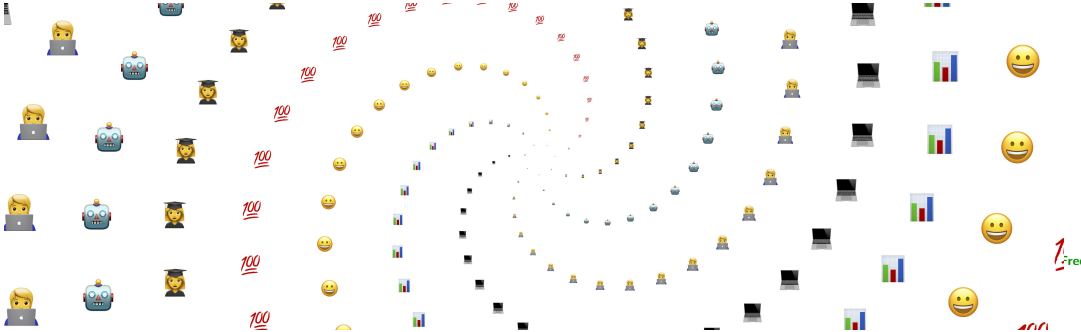
Laufzeitkomplexität

Thomas Graf

Fachschaft Informatik
Kantonsschule Im Lee



Motivation

- Bislang stand für uns hauptsächlich die Frage

  „Wie können wir ein Problem **überhaupt** algorithmisch lösen?“

  im Zentrum.



Motivation

- Bislang stand für uns hauptsächlich die Frage

  „Wie können wir ein Problem **überhaupt** algorithmisch lösen?“

  im Zentrum.
- Dabei haben wir uns (grob gesagt) **mit jeder korrekten algorithmischen Lösung begnügt**.



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?

- Eleganz?



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?

- Eleganz?
- Effizienz?



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?

- Eleganz?
- Effizienz?
- Lesbarkeit und Verständlichkeit?



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?

- Eleganz?
- Effizienz?
- Lesbarkeit und Verständlichkeit?
- Länge des Codes? (Anzahl Zeilen / Zeichen)



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?



Diskussion im Plenum

Welche Aspekte eines Algorithmus könnten uns (neben seiner Korrektheit) noch interessieren?

- Eleganz
- **Effizienz** ← unser Fokus für heute
- Lesbarkeit und Verständlichkeit
- Länge des Codes (Anzahl Zeilen)



Effizienz (Laufzeit) ist eine wichtige Eigenschaft eines Algorithmus

- Kollisionserkennung: Unterschiede in den Framerates



Effizienz (Laufzeit) ist eine wichtige Eigenschaft eines Algorithmus

- Kollisionserkennung: Unterschiede in den Framerates
- Google: „Loading times longer than a blink of an eye (or 400 milliseconds) are too long“:
  https://current360.com/400-milliseconds-or-the-blink-of-an-eye/



Effizienz (Laufzeit) ist eine wichtige Eigenschaft eines Algorithmus

- Kollisionserkennung: Unterschiede in den Framerates
- Google: „Loading times longer than a blink of an eye (or 400 milliseconds) are too long“:
  https://current360.com/400-milliseconds-or-the-blink-of-an-eye/
- Demonstration: Vergleich der Laufzeiten von `bubble_sort` und `merge_sort`



Lernziele

- Sie besitzen eine intuitive Vorstellung für die Komplexität alltäglicher algorithmischer Szenarien.



Lernziele

- Sie besitzen eine intuitive Vorstellung für die Komplexität alltäglicher algorithmischer Szenarien.
- Sie können die Begriffe *Worst-Case* und *Best-Case* erklären.



Lernziele

- Sie besitzen eine intuitive Vorstellung für die Komplexität alltäglicher algorithmischer Szenarien.
- Sie können die Begriffe *Worst-Case* und *Best-Case* erklären.
- Sie können die Laufzeit einfacher Algorithmen bestimmen.



Bestimmung der Laufzeit ★☆☆☆☆

Aufgabe 0.1

```
def a():
    for _ in range(5):
        for _ in range(4):
            print("Informatik")
```

Programm: `a.py`

Wie oft wird die Funktion `print` aufgerufen?



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.1

Die Funktion `print` wird $5 \cdot 4 = 20$ mal aufgerufen.



Bestimmung der Laufzeit ★☆☆☆☆

Aufgabe 0.2

```python
def b(n):
    for _ in range(n):
        for _ in range(n):
            print("Laufzeitkomplexität")
```

Programm: `b.py`

Wie oft wird die Funktion `print` aufgerufen?



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.2

Die Funktion `print` wird n^2 mal aufgerufen.



Bestimmung der Laufzeit ★★☆☆☆

Aufgabe 0.3

```python
def c(L):
    n = len(L)  # Grösse der Probleminstanz
    x = 7
    for k in range(n):
        if L[k] == x:
            return True
    return False
```

Programm: `c.py`

Wie oft wird das `if`-Statement ausgeführt? 🤔



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.3

Die genaue Anzahl hängt von der konkreten **Probleminstanz** — in diesem Fall von der gegebenen Liste `L` — ab. Wir unterscheiden zwei extreme Fälle:



✔ Lösungsvorschlag zu Aufgabe 0.3

Die genaue Anzahl hängt von der konkreten **Probleminstanz** — in diesem Fall von der gegebenen Liste `L` — ab. Wir unterscheiden zwei extreme Fälle:

Günstigster Fall (Best-Case): Das erste Element (`L[0]`) ist bereits das gesuchte Element `x` $\Rightarrow$ das `if`-Statement wird nur einmal ausgeführt.



✔ Lösungsvorschlag zu Aufgabe 0.3

Die genaue Anzahl hängt von der konkreten **Probleminstanz** — in diesem Fall von der gegebenen Liste `L` — ab. Wir unterscheiden zwei extreme Fälle:

Günstigster Fall (Best-Case): Das erste Element (`L[0]`) ist bereits das gesuchte Element `x` ⇒ das `if`-Statement wird nur einmal ausgeführt.

Teuerster Fall (Worst-Case): Das gesuchte Element `x` ist entweder das letzte Element der Liste oder kommt in der Liste gar nicht vor ⇒ das `if`-Statement wird `n = len(L)` mal ausgeführt (jedes Element von `L` muss überprüft werden).



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.3

Die genaue Anzahl hängt von der konkreten **Probleminstanz** — in diesem Fall von der gegebenen Liste `L` — ab. Wir unterscheiden zwei extreme Fälle:

Günstigster Fall (Best-Case): Das erste Element (`L[0]`) ist bereits das gesuchte Element `x` ⇒ das `if`-Statement wird nur einmal ausgeführt.

Teuerster Fall (Worst-Case): Das gesuchte Element `x` ist entweder das letzte Element der Liste oder kommt in der Liste gar nicht vor ⇒ das `if`-Statement wird `n = len(L)` mal ausgeführt (jedes Element von `L` muss überprüft werden).



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.3

Die genaue Anzahl hängt von der konkreten **Probleminstanz** — in diesem Fall von der gegebenen Liste `L` — ab. Wir unterscheiden zwei extreme Fälle:

Günstigster Fall (Best-Case): Das erste Element (`L[0]`) ist bereits das gesuchte Element `x` ⇒ das `if`-Statement wird nur einmal ausgeführt.

Teuerster Fall (Worst-Case): Das gesuchte Element `x` ist entweder das letzte Element der Liste oder kommt in der Liste gar nicht vor ⇒ das `if`-Statement wird `n = len(L)` mal ausgeführt (jedes Element von `L` muss überprüft werden).

Natürlich können auch alle Fälle zwischen diesen beiden Extremfällen (1 bis `len(L)` Ausführungen) auftreten.



Worst-Case und *Best-Case*

- Die *Worst-Case* Laufzeit $T_{\text{worst}}(n)$ ist die maximale Anzahl der Operationen, die ein Algorithmus A für eine Eingabe (Probleminstanz) der Grösse n benötigt.



Worst-Case und *Best-Case*

- Die *Worst-Case* Laufzeit $T_{\text{worst}}(n)$ ist die maximale Anzahl der Operationen, die ein Algorithmus A für eine Eingabe (Probleminstanz) der Grösse n benötigt.
- Die *Best-Case* Laufzeit $T_{\text{best}}(n)$ ist die minimale Anzahl der Operationen, die ein Algorithmus A für eine Eingabe (Probleminstanz) der Grösse n benötigt.



Bemerkung

$T_{\text{worst}}(n)$ ist die **obere Schranke**: Sie garantiert, dass der Algorithmus A für **jede** Probleminstanz der Grösse n nach spätestens dieser Anzahl an Operationen fertig ist.



Bestimmung der Laufzeit ★★☆☆☆

Aufgabe 0.4

```python
def d(n):
    # n >= 1 ist eine ganze Zahl
    result = 0
    i = 1
    while i <= n:
        result += i
        # verdopple i in jedem Schleifendurchlauf
        i *= 2
    return result
```

Programm: `d.py`

Wie oft wird die Zeile `i *= 2` ungefähr ausgeführt?



✔ Lösungsvorschlag zu Aufgabe 0.4

Wir beginnen mit `i = 1` und `i` wird in jedem Durchlauf der `while`-Schleife verdoppelt. Wie häufig können wir den Wert von `i` verdoppeln, bis die `while`-Bedingung `i <= n` nicht mehr erfüllt ist? Ungefähr $\log_2(n)$ mal.



✔ Lösungsvorschlag zu Aufgabe 0.4

Wir beginnen mit `i = 1` und `i` wird in jedem Durchlauf der `while`-Schleife verdoppelt. Wie häufig können wir den Wert von `i` verdoppeln, bis die `while`-Bedingung `i <= n` nicht mehr erfüllt ist? Ungefähr $\log_2(n)$ mal.

Exakte Analyse: Die Zeile `i *= 2` wird $\lceil \log_2(n) \rceil + 1$ mal ausgeführt.



FreeFlower

Aufgaben Teil 1 auf Moodle

Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.



Aufgaben Teil 1 auf Moodle

Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.

- Lösen Sie die **Aufgaben Teil 1** (Bearbeitungszeit: 20 Minuten).



Aufgaben Teil 1 auf Moodle

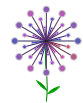
Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.

- Lösen Sie die **Aufgaben Teil 1** (Bearbeitungszeit: 20 Minuten).
- Bearbeiten Sie die Aufgaben alleine.



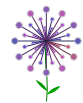
Typisches Problem: „alle mit allen“

- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.



Typisches Problem: „alle mit allen“

- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.



Typisches Problem: „alle mit allen“

- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



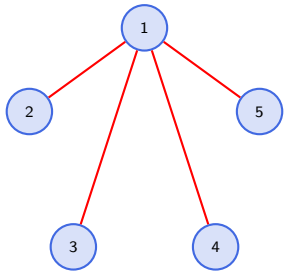
Typisches Problem: „alle mit allen“

- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



Typisches Problem: „alle mit allen“

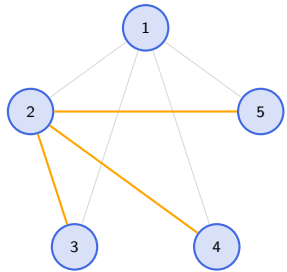
- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



Person 1: $n - 1$ Anstösse

Typisches Problem: „alle mit allen“

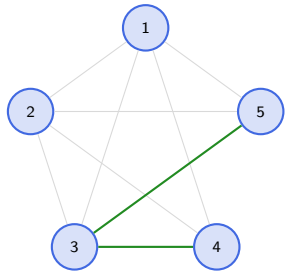
- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



Person 2: $n - 2$ neue Anstösse

Typisches Problem: „alle mit allen“

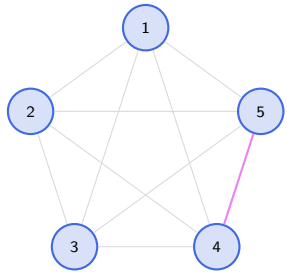
- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



Person 3: $n - 3$ neue Anstösse

Typisches Problem: „alle mit allen“

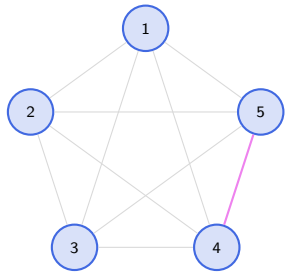
- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



Person 4: $n - 4$ neue Anstösse

Typisches Problem: „alle mit allen“

- Stellen Sie sich vor, n Personen stossen bei einem Fest miteinander an.
- Jede Person stösst mit jeder anderen Person genau einmal an.
- Wie viele Anstösse (Begegnungen) gibt es insgesamt?



Person 4: $n - 4$ neue Anstösse

Insgesamt haben wir also $(n-1) + (n-2) + (n-3) + \ldots + 1$ Anstösse.

„alle mit allen“

- Die Summe $(n-1)+(n-2)+(n-3)+\ldots+1$ ist Ihnen wohlbekannt!



„alle mit allen“

- Die Summe $(n-1)+(n-2)+(n-3)+\ldots+1$ ist Ihnen wohlbekannt!
- $(n-1)+(n-2)+(n-3)+\ldots+1 \overset{\text{Gausssche Summenformel}}{=} \frac{n(n-1)}{2}$



„alle mit allen“

- Die Summe $(n-1)+(n-2)+(n-3)+\ldots+1$ ist Ihnen wohlbekannt!
- $(n-1)+(n-2)+(n-3)+\ldots+1 \overset{\text{Gausssche Summenformel}}{=} \frac{n(n-1)}{2}$
- Beachten Sie den quadratischen Term: $\frac{n(n-1)}{2} = \frac{1}{2} \cdot n^2 - \frac{n}{2}$.



Situation: „alle mit allen“

Aufgabe 0.5

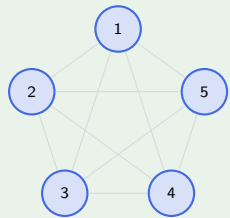
Das Prinzip „Jeder mit Jedem“ begegnet uns an vielen Stellen im täglichen Leben. Wo tritt dieses Muster noch auf? Nennen Sie drei konkrete Situationen, in denen alle Beteiligten untereinander in Kontakt stehen.



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.5

- Tischtennisturnier: Jeder Spieler spielt genau einmal gegen jeden anderen Spieler.
- Fotos von Zweierpaaren: In einer Gruppe von Personen werden Fotos von allen möglichen Zweierpaaren gemacht.
- Freundschaftsnetzwerk: In einem sozialen Netzwerk schreibt jeder Nutzer eine Nachricht an jeden anderen Nutzer (vollständiger Graph):



FreeFlower

Aufgaben Teil 2 auf Moodle

Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.



Aufgaben Teil 2 auf Moodle

Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.

- Lösen Sie die **Aufgaben Teil 2** (Bearbeitungszeit: 20 Minuten).
- Bearbeiten Sie die Aufgaben alleine oder zu zweit.



Aufgabe 0.6

Gegeben sind zwei Algorithmen A und B für ein algorithmisches Problem.

- Algorithmus A: benötigt 2^n Rechenoperationen,
- Algorithmus B: benötigt n^2 Rechenoperationen

zur Berechnung einer Lösung für eine Problemgrösse n.

Annahme: Eine Operation dauert genau 1 Mikrosekunde.

Wie gross darf n maximal sein, damit die Berechnung in 24 Stunden fertig sein wird? (für A und B)



✔ Lösungsvorschlag zu Aufgabe 0.6

- Algorithmus A: Wir müssen die Ungleichung $2^n \leq 24 \cdot 60 \cdot 60 \cdot 10^6$ lösen, da es $24 \cdot 60 \cdot 60 \cdot 10^6$ Mikrosekunden in 24 Stunden gibt. Das ergibt $n \leq \log_2 (24 \cdot 60 \cdot 60 \cdot 10^6) \approx 36.33 \Rightarrow$ gesuchte Problemgrösse: 36.



✔ Lösungsvorschlag zu Aufgabe 0.6

- Algorithmus A: Wir müssen die Ungleichung $2^n \leq 24 \cdot 60 \cdot 60 \cdot 10^6$ lösen, da es $24 \cdot 60 \cdot 60 \cdot 10^6$ Mikrosekunden in 24 Stunden gibt. Das ergibt $n \leq \log_2 (24 \cdot 60 \cdot 60 \cdot 10^6) \approx 36.33 \Rightarrow$ gesuchte Problemgrösse: 36.
- Algorithmus B: Wir müssen die Ungleichung $n^2 \leq 24 \cdot 60 \cdot 60 \cdot 10^6$ lösen. Das ergibt $n \leq \sqrt{24 \cdot 60 \cdot 60 \cdot 10^6} \approx 293938.77 \Rightarrow$ gesuchte Problemgrösse: 293938.



Aufgabe 0.7

Gegeben sind zwei Algorithmen A und B für ein algorithmisches Problem. In diesem Kontext existieren nur Problemgrössen $n \in \mathbb{N}$, die **gerade** sind.

- Algorithmus A: benötigt $4.5n + 15$ Rechenoperationen.
- Algorithmus B: benötigt $0.5n^2 + n$ Rechenoperationen.

Hinweis: Wir gehen davon aus, dass jede Rechenoperation die gleiche Zeit beansprucht.

Welches ist die kleinste gerade Zahl $n > 0$, für welche Algorithmus A das Problem schneller löst (also weniger Rechenoperationen benötigt) als Algorithmus B?



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.7

Wir müssen die Ungleichung $4.5n+15 < 0.5n^2+n$ lösen. Das ergibt $n^2-7n-30 > 0$. Die Lösungen dieser Ungleichung sind $n < -3$ sowie $n > 10$. Da wir nur gerade Zahlen $n > 0$ betrachten, ist die gesuchte Problemgrösse 12.



Bestimmung der Laufzeit ★★☆☆☆

Aufgabe 0.8

```
def anstossen(personen):
    """
    jede Person soll mit jeder anderen genau einmal anstossen
    niemand stösst mit sich selber an
    """
    n = len(personen)
    for i in range(0, n - 1):
        for j in range(i + 1, n):
            print(personen[i], "mit", personen[j])
```

Programm: `anstossen.py`

Wie oft wird die Funktion `print` aufgerufen?

FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n - 1$ mal,

FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n - 1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n - 2$ mal,



✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n - 1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n - 2$ mal,
- $\vdots$



✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n - 1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n - 2$ mal,
- $\vdots$
- im letzten Durchlauf der äusseren Schleife (Fall `i = n - 2`) genau 1 mal



✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n - 1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n - 2$ mal,
- $\vdots$
- im letzten Durchlauf der äusseren Schleife (Fall `i = n - 2`) genau 1 mal



✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n - 1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n - 2$ mal,
- $\vdots$
- im letzten Durchlauf der äusseren Schleife (Fall `i = n - 2`) genau 1 mal

aufgerufen.



✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n-1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n-2$ mal,
- $\vdots$
- im letzten Durchlauf der äusseren Schleife (Fall `i = n - 2`) genau 1 mal

aufgerufen.
Total haben wir also

$$1+2+3+\ldots+(n-2)+(n-1) \overset{\text{Gausssche Summenformel}}{=} \frac{n(n-1)}{2} =$$

$$= \frac{1}{2}\cdot n^2 - \frac{n}{2}$$

FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.8

Die Funktion `print` wird

- im ersten Durchlauf der äusseren Schleife (Fall `i = 0`) genau $n-1$ mal,
- im zweiten Durchlauf der äusseren Schleife (Fall `i = 1`) genau $n-2$ mal,
- $\vdots$
- im letzten Durchlauf der äusseren Schleife (Fall `i = n - 2`) genau 1 mal

aufgerufen.
Total haben wir also

$$1+2+3+\ldots+(n-2)+(n-1) \overset{\text{Gausssche Summenformel}}{=} \frac{n(n-1)}{2} =$$
$$= \frac{1}{2}\cdot n^2 - \frac{n}{2}$$

Aufrufe von `print`.



FreeFlower

Exakte Analyse eines Algorithmus

Definition (Anzahl Wiederholungen pro Zeile)

Bei einem gegebenen Algorithmus A, bezeichnen wir mit $w_k(n)$ die Anzahl der Ausführungen von Zeile k bei Eingabegrösse n.



Beispiel einer exakten Analyse (Problemgrösse n)

Wir analysieren ein einfaches Programm, das die Summe der ersten n natürlichen Zahlen berechnet:

```
1 def berechne_summe(n):
2     s = 0
3     for i in range(n):
4         s = s + i
5     return s
```



Beispiel einer exakten Analyse (Problemgrösse n)

Wir analysieren ein einfaches Programm, das die Summe der ersten n natürlichen Zahlen berechnet:

```
1 def berechne_summe(n):
2     s = 0
3     for i in range(n):
4         s = s + i
5     return s
```

$w_1(n) = 1$



Beispiel einer exakten Analyse (Problemgrösse n)

Wir analysieren ein einfaches Programm, das die Summe der ersten n natürlichen Zahlen berechnet:

```
1 def berechne_summe(n):
2     s = 0
3     for i in range(n):
4         s = s + i
5     return s
```

$w_1(n) = 1$

$w_2(n) = 1$



Beispiel einer exakten Analyse (Problemgrösse n)

Wir analysieren ein einfaches Programm, das die Summe der ersten n natürlichen Zahlen berechnet:

```
1 def berechne_summe(n):
2     s = 0
3     for i in range(n):
4         s = s + i
5     return s
```

$w_1(n) = 1$

$w_2(n) = 1$

$w_3(n) = n + 1$



Beispiel einer exakten Analyse (Problemgrösse n)

Wir analysieren ein einfaches Programm, das die Summe der ersten n natürlichen Zahlen berechnet:

```
1 def berechne_summe(n):          w1(n) = 1
2     s = 0                       w2(n) = 1
3     for i in range(n):          w3(n) = n + 1
4         s = s + i               w4(n) = n
5     return s
```



Beispiel einer exakten Analyse (Problemgrösse n)

Wir analysieren ein einfaches Programm, das die Summe der ersten n natürlichen Zahlen berechnet:

```
1 def berechne_summe(n):          w1(n) = 1
2     s = 0                       w2(n) = 1
3     for i in range(n):          w3(n) = n + 1
4         s = s + i               w4(n) = n
5     return s                    w5(n) = 1
```



- $w_3(n) = n + 1$, da die `for`-Schleife n mal läuft und am Ende einmal die Abbruchbedingung prüft.



- $w_3(n) = n + 1$, da die `for`-Schleife n mal läuft und am Ende einmal die Abbruchbedingung prüft.
- Wir bezeichnen mit c_k die Kosten (**c**ost) einer einmaligen Ausführung von Programmzeile k.



- $w_3(n) = n + 1$, da die `for`-Schleife n mal läuft und am Ende einmal die Abbruchbedingung prüft.
- Wir bezeichnen mit c_k die Kosten (**c**ost) einer einmaligen Ausführung von Programmzeile k.
- Unter Kosten verstehen wir die Anzahl der elementaren Operationen, die in Zeile k ausgeführt werden (z. B. Zuweisungen, Vergleiche, arithmetische Operationen, etc.).



- $w_3(n) = n + 1$, da die `for`-Schleife n mal läuft und am Ende einmal die Abbruchbedingung prüft.
- Wir bezeichnen mit c_k die Kosten (**c**ost) einer einmaligen Ausführung von Programmzeile k.
- Unter Kosten verstehen wir die Anzahl der elementaren Operationen, die in Zeile k ausgeführt werden (z. B. Zuweisungen, Vergleiche, arithmetische Operationen, etc.).
- Wir nehmen an, dass die c_k unabhängig von n sind.



- $w_3(n) = n + 1$, da die `for`-Schleife n mal läuft und am Ende einmal die Abbruchbedingung prüft.
- Wir bezeichnen mit c_k die Kosten (**c**ost) einer einmaligen Ausführung von Programmzeile k.
- Unter Kosten verstehen wir die Anzahl der elementaren Operationen, die in Zeile k ausgeführt werden (z. B. Zuweisungen, Vergleiche, arithmetische Operationen, etc.).
- Wir nehmen an, dass die c_k unabhängig von n sind.
- Die Gesamtlaufzeit $T(n)$ ergibt sich aus den Kosten c_k pro Zeile:



Beispiel einer exakten Analyse (Fortsetzung)

Wir stellen die Formel für $T(n)$ auf:

$$, \qquad a > 0.$$



Beispiel einer exakten Analyse (Fortsetzung)

Wir stellen die Formel für $T(n)$ auf:

$$T(n) = c_1 w_1(n) + c_2 w_2(n) + c_3 w_3(n) + c_4 w_4(n) + c_5 w_5(n) =$$

, $a > 0$.



Beispiel einer exakten Analyse (Fortsetzung)

Wir stellen die Formel für $T(n)$ auf:

$$T(n) = c_1 w_1(n) + c_2 w_2(n) + c_3 w_3(n) + c_4 w_4(n) + c_5 w_5(n) =$$
$$= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 =$$

, $a > 0$.



FreeFlower

Beispiel einer exakten Analyse (Fortsetzung)

Wir stellen die Formel für $T(n)$ auf:

$$\begin{aligned} T(n) &= c_1 w_1(n) + c_2 w_2(n) + c_3 w_3(n) + c_4 w_4(n) + c_5 w_5(n) = \\ &= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 = \\ &= \underbrace{(c_3 + c_4)}_{=:a} n + \underbrace{(c_1 + c_2 + c_3 + c_5)}_{=:b} = \\ &\quad , \qquad a > 0. \end{aligned}$$



Beispiel einer exakten Analyse (Fortsetzung)

Wir stellen die Formel für $T(n)$ auf:

$$\begin{aligned} T(n) &= c_1 w_1(n) + c_2 w_2(n) + c_3 w_3(n) + c_4 w_4(n) + c_5 w_5(n) = \\ &= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 = \\ &= \underbrace{(c_3 + c_4)}_{=:a} n + \underbrace{(c_1 + c_2 + c_3 + c_5)}_{=:b} = \\ &= an + b, \qquad a > 0. \end{aligned}$$



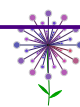
Beispiel einer exakten Analyse (Fortsetzung)

Wir stellen die Formel für $T(n)$ auf:

$$\begin{aligned} T(n) &= c_1 w_1(n) + c_2 w_2(n) + c_3 w_3(n) + c_4 w_4(n) + c_5 w_5(n) = \\ &= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 = \\ &= \underbrace{(c_3 + c_4)}_{=:a} n + \underbrace{(c_1 + c_2 + c_3 + c_5)}_{=:b} = \\ &= an + b, \qquad a > 0. \end{aligned}$$

Bemerkung

Dieses Programm hat eine **lineare Laufzeit**.



FreeFlower

Aufgaben Teil 3 auf Moodle

Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.



Aufgaben Teil 3 auf Moodle

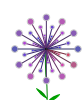
Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.

- Lösen Sie die **Aufgaben Teil 3** (Bearbeitungszeit: 8 Minuten).
- Bearbeiten Sie die Aufgaben alleine oder zu zweit.



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```



FreeFlower

Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```

$w_1(n) = 1$



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):                          # w1(n) = 1
2     n = len(A)                               # w2(n) = 1
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):                          w1(n) = 1
2     n = len(A)                               w2(n) = 1
3     for i in range(n - 1):                   w3(n) = n
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```

$w_1(n) = 1$

$w_2(n) = 1$

$w_3(n) = n$

$w_5(n)$



FreeFlower

Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```

$w_1(n) = 1$

$w_2(n) = 1$

$w_3(n) = n$

$w_5(n)$

$w_6(n) = w_5(n)$



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```

$w_1(n) = 1$

$w_2(n) = 1$

$w_3(n) = n$

$w_5(n)$

$w_6(n) = w_5(n)$

$w_7(n) = 1$



FreeFlower

Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j+1] = A[j+1], A[j]
7     return A
```

$w_1(n) = 1$

$w_2(n) = 1$

$w_3(n) = n$

$w_4(n) = w_5(n) + n - 1$

$w_5(n)$

$w_6(n) = w_5(n)$

$w_7(n) = 1$



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

- Der Worst-Case liegt vor, wenn die gegebene Liste **absteigend** sortiert ist (z. B. `A = [9, 5, 3, 0]`).



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

- Der Worst-Case liegt vor, wenn die gegebene Liste **absteigend** sortiert ist (z. B. `A = [9, 5, 3, 0]`).
- Im Worst-Case müssen die Elemente in **jedem** inneren Schleifendurchlauf getauscht werden $\Rightarrow w_6(n) = w_5(n)$.



Analyse von `bubble_sort` für Problemgrösse n, Worst-Case

- Der Worst-Case liegt vor, wenn die gegebene Liste **absteigend** sortiert ist (z. B. `A = [9, 5, 3, 0]`).
- Im Worst-Case müssen die Elemente in **jedem** inneren Schleifendurchlauf getauscht werden $\Rightarrow w_6(n) = w_5(n)$.
- $w_4(n)$ ist grösser als $w_5(n)$, da die `for`-Bedingung pro äusserem Durchlauf einmal öfter geprüft wird (Abbruchprüfung).



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:

- Wenn `i = 0`, läuft `j` von 0 bis `n - 2` $\Rightarrow (n-1)$ Mal



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:

- Wenn `i = 0`, läuft `j` von 0 bis `n - 2` $\Rightarrow (n-1)$ Mal
- Wenn `i = 1`, läuft `j` von 0 bis `n - 3` $\Rightarrow (n-2)$ Mal



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:

- Wenn `i = 0`, läuft `j` von 0 bis `n - 2` $\Rightarrow (n-1)$ Mal
- Wenn `i = 1`, läuft `j` von 0 bis `n - 3` $\Rightarrow (n-2)$ Mal
- $\vdots$



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:

- Wenn `i = 0`, läuft `j` von 0 bis `n - 2` $\Rightarrow (n-1)$ Mal
- Wenn `i = 1`, läuft `j` von 0 bis `n - 3` $\Rightarrow (n-2)$ Mal
- $\vdots$
- Wenn `i = n - 2`, läuft `j` von 0 bis 0, wegen `range(0, 1)` $\Rightarrow$ 1 Mal



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:

- Wenn `i = 0`, läuft `j` von 0 bis `n - 2` $\Rightarrow (n-1)$ Mal
- Wenn `i = 1`, läuft `j` von 0 bis `n - 3` $\Rightarrow (n-2)$ Mal
- $\vdots$
- Wenn `i = n - 2`, läuft `j` von 0 bis 0, wegen `range(0, 1)` $\Rightarrow$ 1 Mal



Genaue Analyse von bubble_sort, Worst-Case

```
1 def bubble_sort(A):
2     n = len(A)
3     for i in range(n - 1):
4         for j in range(0, n - i - 1):
5             if A[j] > A[j + 1]:
6                 A[j], A[j + 1] = A[j + 1], A[j]
7     return A
```

Programm: bubble_sort.py

Wir müssen nun die totale Anzahl der Ausführungen von Zeile 5, also $w_5(n)$, bestimmen:

- Wenn `i = 0`, läuft `j` von 0 bis `n - 2` $\Rightarrow (n-1)$ Mal
- Wenn `i = 1`, läuft `j` von 0 bis `n - 3` $\Rightarrow (n-2)$ Mal
- $\vdots$
- Wenn `i = n - 2`, läuft `j` von 0 bis 0, wegen `range(0, 1)` $\Rightarrow$ 1 Mal

Wir erhalten die bekannte Summe:

$$w_5(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$



Genaue Analyse von `bubble_sort`, Worst-Case

$$
\begin{aligned}
T_{\text{worst}}(n) &= c_1 w_1(n) + c_2 w_2(n) + \ldots + c_7 w_7(n) = \\
&= c_1 + c_2 + c_3 n + c_4 \left(w_5(n) + n - 1\right) + c_5 w_5(n) + c_6 w_5(n) + c_7 = \\
&= c_1 + c_2 + c_3 n + c_4 \left(\frac{n(n-1)}{2} + n - 1\right) + (c_5 + c_6) \left(\frac{n(n-1)}{2}\right) + c_7
\end{aligned}
$$



Genaue Analyse von `bubble_sort`

Wir vereinfachen die Terme und gruppieren nach den Potenzen von n, um den folgenden Ausdruck zu erhalten:



Genaue Analyse von `bubble_sort`

Wir vereinfachen die Terme und gruppieren nach den Potenzen von n, um den folgenden Ausdruck zu erhalten:

$$T_{\text{worst}}(n) = \underbrace{\left(\frac{c_4 + c_5 + c_6}{2}\right)}_{=:a} n^2 + \underbrace{\left(c_3 + \frac{c_4 - c_5 - c_6}{2}\right)}_{=:b} n + \underbrace{(c_1 + c_2 - c_4 + c_7)}_{=:c} =$$

$$= an^2 + bn + c, \qquad a > 0.$$



Challenge-Aufgaben auf Moodle

Sie finden die Aufgaben unter `Moodle` → `Theorie` → `Laufzeitkomplexität`.

Lösen Sie die **Challenge-Aufgaben**.



Analyse von insertion_sort

Aufgabe 0.9

Führen Sie eine vollständige Laufzeitanalyse von `insertion_sort` für den **Worst-Case** durch:

```
1  def insertion_sort(A):
2      n = len(A)
3      for j in range(1, n):
4          key = A[j]
5          i = j - 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] = A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```

Programm: `insertion_sort.py`

FreeFlower

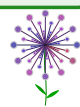
✔ Lösungsvorschlag zu Aufgabe 0.9

Im Worst-Case muss jedes Element mit allen vorherigen Elementen verglichen werden. Das Element bei Index j wird $j+1$ Mal in der `while`-Bedingung überprüft.



✔ Lösungsvorschlag zu Aufgabe 0.9

```python
def insertion_sort(A):
    n = len(A)
    for j in range(1, n):
        key = A[j]
        i = j - 1
        while i >= 0 and A[i] > key:
            A[i + 1] =  A[i]
            i -= 1
        A[i + 1] = key
    return A
```



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):
2      n = len(A)
3      for j in range(1, n):
4          key = A[j]
5          i = j - 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```

$w_1(n) = 1$



✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):
2      n = len(A)
3      for j in range(1, n):
4          key = A[j]
5          i = j - 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```

$w_1(n) = 1$

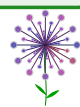
$w_2(n) = 1$



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.9

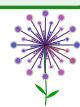
```
1  def insertion_sort(A):                    # w1(n) = 1
2      n = len(A)                            # w2(n) = 1
3      for j in range(1, n):                 # w3(n) = n
4          key = A[j]
5          i = j - 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```



FreeFlower

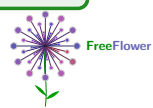
✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):                    w1(n) = 1
2      n = len(A)                            w2(n) = 1
3      for j in range(1, n):                 w3(n) = n
4          key = A[j]                        w4(n) = n − 1
5          i = j - 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```



✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):                  # w1(n) = 1
2      n = len(A)                          # w2(n) = 1
3      for j in range(1, n):               # w3(n) = n
4          key = A[j]                      # w4(n) = n − 1
5          i = j - 1                       # w5(n) = n − 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```



✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):
2      n = len(A)
3      for j in range(1, n):
4          key = A[j]
5          i = j - 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]
8              i -= 1
9          A[i + 1] = key
10     return A
```

$w_1(n) = 1$

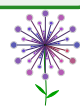
$w_2(n) = 1$

$w_3(n) = n$

$w_4(n) = n - 1$

$w_5(n) = n - 1$

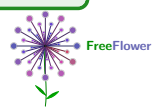
$w_7(n)$



FreeFlower

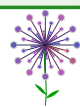
✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):                    w1(n) = 1
2      n = len(A)                            w2(n) = 1
3      for j in range(1, n):                 w3(n) = n
4          key = A[j]                        w4(n) = n − 1
5          i = j - 1                         w5(n) = n − 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]              w7(n)
8              i -= 1                        w8(n) = w7(n)
9          A[i + 1] = key
10     return A
```



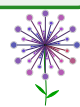
✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):                    w1(n) = 1
2      n = len(A)                            w2(n) = 1
3      for j in range(1, n):                 w3(n) = n
4          key = A[j]                        w4(n) = n − 1
5          i = j - 1                         w5(n) = n − 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]              w7(n)
8              i -= 1                        w8(n) = w7(n)
9          A[i + 1] = key                    w9(n) = n − 1
10     return A
```



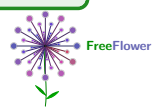
✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):                    w1(n) = 1
2      n = len(A)                            w2(n) = 1
3      for j in range(1, n):                 w3(n) = n
4          key = A[j]                        w4(n) = n − 1
5          i = j - 1                         w5(n) = n − 1
6          while i >= 0 and A[i] > key:
7              A[i + 1] =  A[i]              w7(n)
8              i -= 1                        w8(n) = w7(n)
9          A[i + 1] = key                    w9(n) = n − 1
10     return A                              w10(n) = 1
```



✔ Lösungsvorschlag zu Aufgabe 0.9

```
1  def insertion_sort(A):                          w1(n) = 1
2      n = len(A)                                  w2(n) = 1
3      for j in range(1, n):                       w3(n) = n
4          key = A[j]                              w4(n) = n − 1
5          i = j - 1                               w5(n) = n − 1
6          while i >= 0 and A[i] > key:            w6(n) = w7(n) + (n − 1)
7              A[i + 1] =  A[i]                    w7(n)
8              i -= 1                              w8(n) = w7(n)
9          A[i + 1] = key                          w9(n) = n − 1
10     return A                                    w10(n) = 1
```



✔ Lösungsvorschlag zu Aufgabe 0.9

$$w_7(n) = 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

Für jeden der $n-1$ Durchläufe der äusseren Schleife muss im Kopf der `while` -Schleife am Ende noch einmal die Abbruch-Bedingung überprüft werden $\Rightarrow$ $w_6(n) = w_7(n) + (n-1)$.



✔ Lösungsvorschlag zu Aufgabe 0.9

Zusammengefasst erhalten wir:

$$\tilde{T}_{\text{worst}}(n) = \tilde{c}_1 w_1(n) + \tilde{c}_2 w_2(n) + \ldots + \tilde{c}_{10} w_{10}(n) = \sum_{k=1}^{10} \tilde{c}_k w_k(n) =$$
$$= \tilde{c}_1 + \tilde{c}_2 + \tilde{c}_3 n + \tilde{c}_4 (n-1) + \tilde{c}_5 (n-1) +$$
$$+ \tilde{c}_6 \left(\frac{n(n-1)}{2} + (n-1) \right) + \tilde{c}_7 \left(\frac{n(n-1)}{2} \right) +$$
$$+ \tilde{c}_8 \left(\frac{n(n-1)}{2} \right) + \tilde{c}_9 (n-1) + \tilde{c}_{10}$$



✔ Lösungsvorschlag zu Aufgabe 0.9

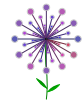
Wir vereinfachen die Terme und gruppieren nach den Potenzen von n, um den folgenden Ausdruck zu erhalten:

$$\begin{aligned}\tilde{T}_{\text{worst}}(n) = &\underbrace{\left(\frac{\tilde{c}_6 + \tilde{c}_7 + \tilde{c}_8}{2}\right)}_{=:\tilde{a}} n^2 + \\ &+ \underbrace{\left(\tilde{c}_3 + \tilde{c}_4 + \tilde{c}_5 + \frac{\tilde{c}_6 - \tilde{c}_7 - \tilde{c}_8}{2} + \tilde{c}_9\right)}_{=:\tilde{b}} n + \\ &+ \underbrace{(\tilde{c}_1 + \tilde{c}_2 - \tilde{c}_4 - \tilde{c}_5 - \tilde{c}_6 - \tilde{c}_9 + \tilde{c}_{10})}_{=:\tilde{c}} = \\ &= \tilde{a}n^2 + \tilde{b} + \tilde{c}, \qquad \tilde{a} > 0.\end{aligned}$$



FreeFlower

Ergänzungsfach



Das Modell der Random Access Machine (RAM)

- Um die Laufzeit von Algorithmen präzise und unabhängig von konkreter Hardware zu analysieren, verwenden wir ein abstraktes Computermodell: die **Random Access Machine (RAM)**.



Das Modell der Random Access Machine (RAM)

- Um die Laufzeit von Algorithmen präzise und unabhängig von konkreter Hardware zu analysieren, verwenden wir ein abstraktes Computermodell: die **Random Access Machine (RAM)**.
- Bisher war unsere Analyse mit Kosten c_k pro Zeile zwar exakt, aber auch aufwendig. Das RAM-Modell vereinfacht die Analyse erheblich.



Das Modell der Random Access Machine (RAM)

- Um die Laufzeit von Algorithmen präzise und unabhängig von konkreter Hardware zu analysieren, verwenden wir ein abstraktes Computermodell: die **Random Access Machine (RAM)**.
- Bisher war unsere Analyse mit Kosten c_k pro Zeile zwar exakt, aber auch aufwendig. Das RAM-Modell vereinfacht die Analyse erheblich.
- Die RAM ist ein idealisierter Computer (ähnlich der Von-Neumann-Architektur) mit folgenden Eigenschaften:



Das Modell der Random Access Machine (RAM)

- Um die Laufzeit von Algorithmen präzise und unabhängig von konkreter Hardware zu analysieren, verwenden wir ein abstraktes Computermodell: die **Random Access Machine (RAM)**.
- Bisher war unsere Analyse mit Kosten c_k pro Zeile zwar exakt, aber auch aufwendig. Das RAM-Modell vereinfacht die Analyse erheblich.
- Die RAM ist ein idealisierter Computer (ähnlich der Von-Neumann-Architektur) mit folgenden Eigenschaften:
  - **Speicher:** Genügend viele Speicherzellen, auf die direkt zugegriffen werden kann (Random Access).



Das Modell der Random Access Machine (RAM)

- Um die Laufzeit von Algorithmen präzise und unabhängig von konkreter Hardware zu analysieren, verwenden wir ein abstraktes Computermodell: die **Random Access Machine (RAM)**.
- Bisher war unsere Analyse mit Kosten c_k pro Zeile zwar exakt, aber auch aufwendig. Das RAM-Modell vereinfacht die Analyse erheblich.
- Die RAM ist ein idealisierter Computer (ähnlich der Von-Neumann-Architektur) mit folgenden Eigenschaften:
  - **Speicher:** Genügend viele Speicherzellen, auf die direkt zugegriffen werden kann (Random Access).
  - **Programm:** Eine Folge von einfachen Befehlen (elementare Operationen), die sequentiell abgearbeitet werden.



Kostenmodell der RAM

- Wie messen wir die „Zeit“ in diesem Modell?



Kostenmodell der RAM

- Wie messen wir die „Zeit“ in diesem Modell?
- Wir verwenden das **Einheitskostenmodell (Uniform Cost Model):**



Kostenmodell der RAM

- Wie messen wir die „Zeit“ in diesem Modell?
- Wir verwenden das **Einheitskostenmodell (Uniform Cost Model):**
  - Jede elementare Operation (Addition, Subtraktion, Vergleich, Zuweisung, Speicherzugriff) benötigt genau **einen Zeitschritt**.



Kostenmodell der RAM

- Wie messen wir die „Zeit“ in diesem Modell?
- Wir verwenden das **Einheitskostenmodell (Uniform Cost Model):**
  - Jede elementare Operation (Addition, Subtraktion, Vergleich, Zuweisung, Speicherzugriff) benötigt genau **einen Zeitschritt**.
  - Die Grösse der Operanden spielt dabei (vereinfachend) keine Rolle.



Kostenmodell der RAM

- Wie messen wir die „Zeit“ in diesem Modell?
- Wir verwenden das **Einheitskostenmodell (Uniform Cost Model):**
  - Jede elementare Operation (Addition, Subtraktion, Vergleich, Zuweisung, Speicherzugriff) benötigt genau **einen Zeitschritt**.
  - Die Grösse der Operanden spielt dabei (vereinfachend) keine Rolle.
- **Bedeutung für die Analyse:**



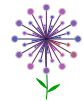
Kostenmodell der RAM

- Wie messen wir die „Zeit“ in diesem Modell?
- Wir verwenden das **Einheitskostenmodell (Uniform Cost Model):**
  - Jede elementare Operation (Addition, Subtraktion, Vergleich, Zuweisung, Speicherzugriff) benötigt genau **einen Zeitschritt**.
  - Die Grösse der Operanden spielt dabei (vereinfachend) keine Rolle.
- **Bedeutung für die Analyse:**
  - Die Laufzeit eines Algorithmus entspricht der Anzahl der ausgeführten elementaren Operationen. Wir müssen also nur noch die Operationen zählen.



Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**



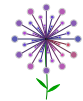
Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.



Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.
  - Interessant ist, wie sich die Laufzeit verhält, wenn die Problemgrössse n wächst ($n \to \infty$).



Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.
  - Interessant ist, wie sich die Laufzeit verhält, wenn die Problemgrössse n wächst $(n \to \infty)$.
- **Praktische Betrachtung:**



Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.
  - Interessant ist, wie sich die Laufzeit verhält, wenn die Problemgrössse n wächst ($n \to \infty$).
- **Praktische Betrachtung:**
  - Exakte Formeln wie $T_0(n) = 12n^2 + 5n + 100$ oder $T_1(n) = 10n^2 + 8n + 213$ sind mühsam zu handhaben.



Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.
  - Interessant ist, wie sich die Laufzeit verhält, wenn die Problemgrössse n wächst ($n \to \infty$).
- **Praktische Betrachtung:**
  - Exakte Formeln wie $T_0(n) = 12n^2 + 5n + 100$ oder $T_1(n) = 10n^2 + 8n + 213$ sind mühsam zu handhaben.
  - Inwiefern unterscheiden sich die Laufzeiten $T_0(n)$ und $T_1(n)$ wirklich?



Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.
  - Interessant ist, wie sich die Laufzeit verhält, wenn die Problemgrössse n wächst $(n \to \infty)$.
- **Praktische Betrachtung:**
  - Exakte Formeln wie $T_0(n) = 12n^2 + 5n + 100$ oder $T_1(n) = 10n^2 + 8n + 213$ sind mühsam zu handhaben.
  - Inwiefern unterscheiden sich die Laufzeiten $T_0(n)$ und $T_1(n)$ wirklich?
  - In der Praxis stellte sich heraus, dass es am sinnvollsten ist, das wesentliche Wachstumsverhalten von Algorithmen zu betrachten.



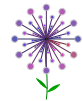
Motivation für die asymptotische Analyse

- **Fokus auf grosse Eingaben (Skalierbarkeit):**
  - Für kleine Problemgrössen n sind fast alle Algorithmen schnell genug berechenbar.
  - Interessant ist, wie sich die Laufzeit verhält, wenn die Problemgrössse n wächst ($n \to \infty$).
- **Praktische Betrachtung:**
  - Exakte Formeln wie $T_0(n) = 12n^2 + 5n + 100$ oder $T_1(n) = 10n^2 + 8n + 213$ sind mühsam zu handhaben.
  - Inwiefern unterscheiden sich die Laufzeiten $T_0(n)$ und $T_1(n)$ wirklich?
  - In der Praxis stellte sich heraus, dass es am sinnvollsten ist, das wesentliche Wachstumsverhalten von Algorithmen zu betrachten.
  - Bezogen auf $T_0(n)$ und $T_1(n)$ heisst dies, dass beide Algorithmen im Wesentlichen quadratisch wachsen.



Motivation für die asymptotische Analyse

- **Unabhängigkeit von Hardware und exakter Implementierung:**



Motivation für die asymptotische Analyse

- **Unabhängigkeit von Hardware und exakter Implementierung:**
  - Die exakten Laufzeitformeln hängen stark von der konkreten Hardware und der genauen Implementierung algorithmischer Ideen ab.



Motivation für die asymptotische Analyse

- **Unabhängigkeit von Hardware und exakter Implementierung:**
  - Die exakten Laufzeitformeln hängen stark von der konkreten Hardware und der genauen Implementierung algorithmischer Ideen ab.
  - Konstante Kosten von 100 Operationen wie in dem Term $T_0(n) = 12n^2 + 5n + 100$ sind z.B. auf eine anfängliche Initialisierung von Variablen zurückzuführen, die auf verschiedenen Maschinen unterschiedlich lange dauern kann.



Motivation für die asymptotische Analyse

- **Unabhängigkeit von Hardware und exakter Implementierung:**
  - Die exakten Laufzeitformeln hängen stark von der konkreten Hardware und der genauen Implementierung algorithmischer Ideen ab.
  - Konstante Kosten von 100 Operationen wie in dem Term $T_0(n) = 12n^2 + 5n + 100$ sind z.B. auf eine anfängliche Initialisierung von Variablen zurückzuführen, die auf verschiedenen Maschinen unterschiedlich lange dauern kann.
  - In der asymptotischen Analyse (bei grossen Probleminstanzen) sind solche Kosten vernachlässigbar.



Motivation für die asymptotische Analyse

- **Unabhängigkeit von Hardware und exakter Implementierung:**
  - Die exakten Laufzeitformeln hängen stark von der konkreten Hardware und der genauen Implementierung algorithmischer Ideen ab.
  - Konstante Kosten von 100 Operationen wie in dem Term $T_0(n) = 12n^2 + 5n + 100$ sind z.B. auf eine anfängliche Initialisierung von Variablen zurückzuführen, die auf verschiedenen Maschinen unterschiedlich lange dauern kann.
  - In der asymptotischen Analyse (bei grossen Probleminstanzen) sind solche Kosten vernachlässigbar.
  - Ähnlich verhält sich dies bei Termen tieferer Ordnung wie $5n$ im Term $T_0(n) = 12n^2 + 5n + 100$.



asymptotische Betrachtung

Lasst uns die Quotienten $\frac{\tilde{T}(n)}{T(n)}$ und $\frac{T(n)}{\tilde{T}(n)}$ der beiden Algorithmen `insertion_sort` und `bubble_sort` betrachten:

$$\lim_{n\to\infty} \frac{\tilde{T}(n)}{T(n)} = \lim_{n\to\infty} \frac{\tilde{a}n^2 + \tilde{b}n + \tilde{c}}{an^2 + bn + c} = \frac{\tilde{a}}{a}$$

$$\lim_{n\to\infty} \frac{T(n)}{\tilde{T}(n)} = \lim_{n\to\infty} \frac{an^2 + bn + c}{\tilde{a}n^2 + \tilde{b}n + \tilde{c}} = \frac{a}{\tilde{a}}$$

Die Grenzwerte der Quotienten unterscheiden sich also lediglich um konstante Faktoren.



asymptotische Betrachtung

$$\lim_{n\to\infty} \frac{n^2}{T(n)} = \lim_{n\to\infty} \frac{n^2}{an^2 + bn + c} = \frac{1}{a}$$

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = \lim_{n\to\infty} \frac{an^2 + bn + c}{n^2} = a$$

(Die analoge Betrachtung kann natürlich auch für `bubble_sort`) durchgeführt werden.)

Die asymptotische (Grenzwert für Problemgrössse $n \to \infty$) Betrachtung zeigt auf, dass sich die Laufzeiten von sowohl `insertion_sort` als auch `bubble_sort` von einer beliebigen Polynomfunktion vom Grad 2 lediglich um eine (multiplikative) Konstante unterscheiden. Dies gilt also insbesondere für die quadratische Funktion $n \mapsto n^2$.



asymptotische Betrachtung

Intuitiv gesprochen, möchten wir sagen, dass in einer asymptotischen Betrachtung, die exakten Wert der Konstanten a, b, c mit $a \neq 0$ in einem Ausdruck $T(n) = an^2 + bn + c$ „nicht wirklich wichtig sind“.



asymptotische Betrachtung

„Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort [...], the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself. When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.“



asymptotische Betrachtung (Landau-Symbole)

Diese intuitive Betrachtung hat man in der Informatik und Mathematik formalisiert. Dies führt uns zu den sogenannten *Landau-Symbolen*, die in der Mathematik und Informatik eine grosse Bedeutung haben.

Es sei g eine Funktion. Wir führen nun die **Mengen von Funktionen** (Landau-Symbole) $O(g(n))$, $\Omega(g(n))$ und $\Theta(g(n))$ ein, die uns helfen, das asymptotische Verhalten von Algorithmen zu beschreiben.



Definition (O-Notation)
Intuitiv bedeutet $f(n) \in O(g(n))$, dass $f(n)$ asymptotisch (für grosse n) höchstens so schnell wächst wie $g(n)$. Noch einfacher gesagt: Für genügend grosse n liegt $f(n)$ unterhalb eines Vielfachen von $g(n)$. Mit „genügend grosse n“ meinen wir, dass es eine Konstante n_0 gibt, sodass die Aussage für alle $n \geq n_0$ gilt.

f wächst also höchstens so schnell wie g.

Genauer:

$$O(g(n)) := \{f(n) \ ; \text{ es existieren positive Konstanten } c, n_0, \text{ sodass } 0 \leq f(n) \leq cg(n) \text{ für alle } n \geq n_0\} \quad (1)$$



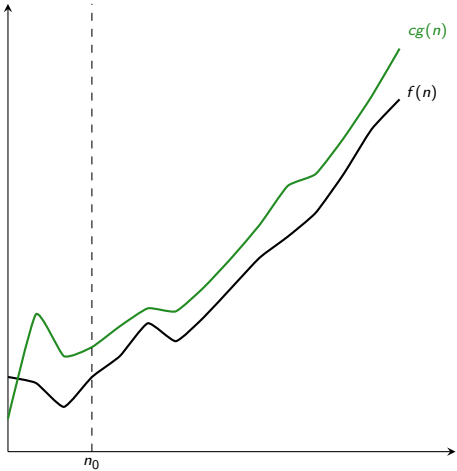


Abbildung: $f(n) \in O(g(n))$. Ab n_0 verläuft die Funktion $f(n)$ nicht mehr oberhalb der Kurve von $cg(n)$. Bitte beachte, dass das hier gezeigte n_0 nicht der kleinste mögliche Wert ist. Der kleinste mögliche Wert wäre in der gezeigten Situation der x-Koordinate des Schnittpunkts der beiden Kurven.



Beispiel (O-Notation)

Aufgabe 0.10

Zeige, dass $f(n) = 3n^2 + 2n + 5 \in O(n^2)$.

Lösungsvorschlag zu Aufgabe 0.10

Wir müssen positive Konstanten c und n_0 finden, sodass $0 \leq 3n^2 + 2n + 5 \leq cn^2$ für alle $n \geq n_0$ gilt. Wähle z.B. $c = 10$. Dann ist $3n^2 + 2n + 5 \leq 10n^2$ für alle $n \geq 1$. Denn $10n^2 - (3n^2 + 2n + 5) = 7n^2 - 2n - 5$, was für alle $n \geq 1$ nicht negativ ist. Also können wir $c := 10$ und $n_0 := 1$ wählen. Somit ist $3n^2 + 2n + 5 \in O(n^2)$ gezeigt.



FreeFlower

Definition (Ω-Notation)
Intuitiv bedeutet $f(n) \in \Omega(g(n))$, dass $f(n)$ asymptotisch (für grosse n) mindestens so schnell wächst wie $g(n)$. Noch einfacher gesagt: Für genügend grosse n liegt $f(n)$ oberhalb eines Vielfachen von $g(n)$. Mit „genügend grosse n“ meinen wir, dass es eine Konstante n_0 gibt, sodass die Aussage für alle $n \geq n_0$ gilt.

f wächst also mindestens so schnell wie g.

Genauer:

$$\Omega(g(n)) := \{f(n) \ ; \text{ es existieren positive Konstanten } c, n_0, \text{ sodass } 0 \leq cg(n) \leq f(n) \text{ für alle } n \geq n_0\} \quad (2)$$



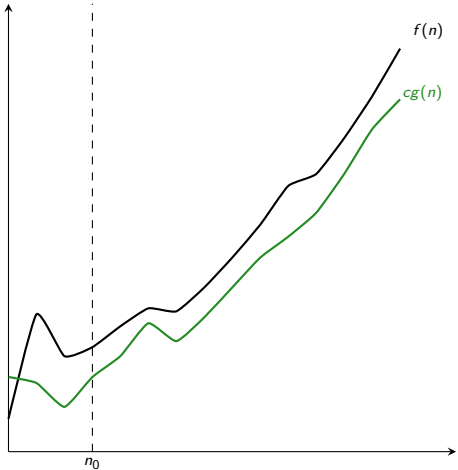


Abbildung: $f(n) \in \Omega(g(n))$. Ab n_0 verläuft die Funktion $f(n)$ nicht mehr unterhalb der Kurve von $cg(n)$. Bitte beachte, dass das hier gezeigte n_0 nicht der kleinste mögliche Wert ist. Der kleinste mögliche Wert wäre in der gezeigten Situation der x-Koordinate des Schnittpunkts der beiden Kurven.

FreeFlower

Beispiel (Ω-Notation)

Aufgabe 0.11

Zeige, dass $f(n) = 3n^2 + 2n + 5 \in \Omega(n^2)$.

Lösungsvorschlag zu Aufgabe 0.11

Wir müssen positive Konstanten c und n_0 finden, sodass $0 \leq cn^2 \leq 3n^2 + 2n + 5$ für alle $n \geq n_0$ gilt. Wähle $c = 1$. Dann ist $n^2 \leq 3n^2 + 2n + 5$ für alle $n \geq 1$. Denn $3n^2 + 2n + 5 - n^2 = 2n^2 + 2n + 5$, was für $n \geq 1$ immer positiv ist. Also können wir $c := 1$ und $n_0 := 1$ wählen. Somit ist $3n^2 + 2n + 5 \in \Omega(n^2)$ gezeigt.



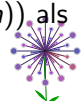
Definition (Θ-Notation)
Intuitiv bedeutet $f(n) \in \Theta(g(n))$, dass $f(n)$ asymptotisch (für grosse n) genauso schnell wächst wie $g(n)$. Noch einfacher gesagt: Für genügend grosse n liegt $f(n)$ zwischen zwei Vielfachen von $g(n)$. Mit „genügend grosse n“ meinen wir, dass es eine Konstante n_0 gibt, sodass die Aussage für alle $n \geq n_0$ gilt.

f wächst also weder wesentlich schneller noch wesentlich langsamer als g.

Genauer:

$$\Theta(g(n)) := \{f(n) \text{ ; es existieren positive Konstanten } c_1, c_2, n_0, \text{ sodass } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ für alle } n \geq n_0\} \quad (3)$$

Man beachte, dass $f(n) \in \Theta(g(n))$ genau dann gilt, wenn sowohl $f(n) \in O(g(n))$ als auch $f(n) \in \Omega(g(n))$ gilt.



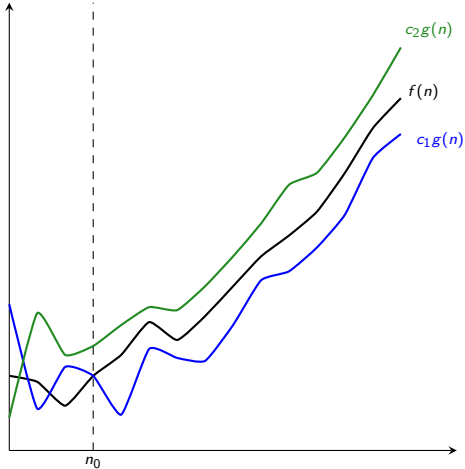


Abbildung: $f(n) \in \Theta(g(n))$. Hier ist der kleinste mögliche Wert für n_0 gezeigt; jeder grössere Wert würde auch funktionieren.

FreeFlower

Beispiel (Θ-Notation)

Aufgabe 0.12

Zeige, dass $f(n) = 3n^2 + 2n + 5 \in \Theta(n^2)$.

Lösungsvorschlag zu Aufgabe 0.12

Wir haben bereits gezeigt, dass $f(n) = 3n^2 + 2n + 5 \in O(n^2)$ und $f(n) = 3n^2 + 2n + 5 \in \Omega(n^2)$. Also gilt $f(n) = 3n^2 + 2n + 5 \in \Theta(n^2)$.



äquivalente Definitionen durch Grezwerte

Man kann zeigen[1], dass die Definitionen von O, Ω und Θ (in den Gleichungen 1, 2 und 3) äquivalent sind zu den folgenden Grenzwertdefinitionen (wir haben noch weitere Symbole definiert):

[1] Dies folgt (vereinfacht gesagt) aus der Definition der Landau-Symbole und der Tatsache, dass die Grenzwerte existieren, wenn die Funktionen asymptotisch gleich wachsen.

Landau-Symbole (allgemein)

Es seien f und g Folgen reeller Zahlen.

Notation	Definition	Bedeutung
$f \in o(g)$	$\lim\limits_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert = 0$	f wächst langsamer als g
$f \in O(g)$	$\limsup\limits_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert < \infty$	f wächst höchstens so schnell wie g
$f \in \Omega(g)$	$\liminf\limits_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert > 0$	f wächst mindestens so schnell wie g, $g \in O(f)$
$f \in \Theta(g)$	$f \in O(g) \cap \Omega(g)$, $0 < \liminf\limits_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert \leq \leq \limsup\limits_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert < \infty$	f wächst gleich schnell wie g, sowohl $f \in O(g)$ als auch $f \in \Omega(g)$
$f \in \omega(g)$	$\lim\limits_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert = \infty$	f wächst schneller als g, $g \in o(f)$



Landau-Symbole (Komplexitätstheorie)

Im Kontext der Komplexitätstheorie seien f und g Folgen natürlicher Zahlen (sie beschreiben jeweils eine Anzahl von Rechenoperationen).

Notation	**Definition**	**Bedeutung**
$f \in o(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	f wächst langsamer als g
$f \in O(g)$	$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	f wächst höchstens so schnell wie g
$f \in \Omega(g)$	$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$	f wächst mindestens so schnell wie g, $g \in O(f)$
$f \in \Theta(g)$	$f \in O(g) \cap \Omega(g)$, $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	f wächst gleich schnell wie g, sowohl $f \in O(g)$ als auch $f \in \Omega(g)$
$f \in \omega(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	f wächst schneller als g, $g \in o(f)$

Wir gehen davon aus, dass die von uns untersuchten Folgen $h(n)$ stets gilt:

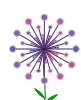
$$\limsup_{n\to\infty} h(n) = \liminf_{n\to\infty} h(n) = \lim_{n\to\infty} h(n).$$



Aufgabe 0.13

Entscheide, ob die folgenden Aussagen zutreffen:

1. Es sei $T(n)$ die worst-case Laufzeit von `insertion_sort`. Zeige, dass $T \in \Theta(n^2)$.
2. Es seien a, b zwei positive reelle Zahlen mit $a \neq 1$, $b \neq 1$. Zeige, dass $\log_a \in \Theta(\log_b)$.



Aufgabe 0.14

Entscheide, ob die folgenden Aussagen zutreffen:

1. $25n^3 \in O\left(n^4\right)$
2. $5n^3 \in O\left(n^3\right)$
3. $2^n \in \Theta\left(2^{n+a}\right)$ für jede positive Konstante $a \in \mathbb{N}$
4. $2^{bn} \in \Theta\left(2^n\right)$ für jede positive Konstante $b \in \mathbb{N}$



Aufgabe 0.15

Entscheide, ob die folgenden Aussagen zutreffen:

1. $\ln(n) \in \Theta(\log_2(n))$
2. $n \in O(\log_2(n))$
3. $n! \in O(n^n)$
4. $n! \in \Theta(n^n)$
5. $2^{2n} \in O(2^n)$
6. $\frac{1}{n} \in O(1)$



Algorithmus präsentieren

1. Bildet 3er-Gruppen.
2. Wählt einen Algorithmus aus dieser Sammlung (oder auch einen eigenen) aus: https://github.com/TheAlgorithms/Python/blob/master/DIRECTORY.md
3. Bereitet eine kurze Präsentation ($\approx$ 5 Minuten) für nächstes Mal vor.

Inhalt der Präsentation:

- Was macht der Algorithmus?
- Wie arbeitet der Algorithmus?
- Diskussion der worst-case Laufzeit $T(n)$ des Algorithmus



Sei A eine $m \times n$-Matrix mit Einträgen

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix}$$

und x ein Vektor der Länge n mit Einträgen

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}.$$



Matrix-Vektor-Multiplikation

Das Matrix-Vektor-Produkt Ax ist genau dann definiert, wenn die „Breite“ (Anzahl der Spalten) von A der „Höhe“ (Länge) von x entspricht. Die „Höhe“ des Produkts Ax entspricht der „Höhe“ der Matrix A.

Betrachten wir nun den allgemeinen Fall: Sei A eine $m \times n$-Matrix mit Einträgen

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix}$$

und x ein Vektor der Länge n mit Einträgen

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}.$$

FreeFlower

Matrix-Vektor-Multiplikation

```
def mvm(A, x):
    num_cols = len(x)
    num_rows = len(A) // num_cols
    b = [0] * num_rows

    for i in range(num_rows):
        for j in range(num_cols):
            b[i] += A[i * num_cols + j] * x[j]

    return b
```

Programm: `mvm.py`



Matrix-Vektor-Multiplikation

Dann ist das Produkt $b := Ax$ ein Vektor der Länge m:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ a_{3,1} & a_{3,2} & \dots & a_{3,n} \\ a_{4,1} & a_{4,2} & \dots & a_{4,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_m \end{pmatrix}$$



Dabei ist zum Beispiel die Komponente b_3 von b gegeben durch die Zahl

$$a_{3,1}x_1 + a_{3,2}x_2 + \ldots + a_{3,n}x_n.$$

Allgemein sind die Komponenten des Matrix-Vektor-Produkts $b = Ax$ gegeben durch

$$b_i := \sum_{k=1}^{n} a_{i,k}x_k = a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,n}x_n, \quad (i = 1, 2, \ldots, m).$$



Matrix-Vektor-Multiplikation

Dabei ist zum Beispiel die Komponente b_3 des Vektors b gegeben durch die Zahl

$$a_{3,1}x_1 + a_{3,2}x_2 + \ldots + a_{3,n}x_n.$$

Allgemein sind die Komponenten des Matrix-Vektor-Produkts $b = Ax$ gegeben durch

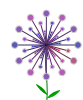
$$b_i := \sum_{k=1}^{n} a_{i,k}x_k = a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,n}x_n, \quad (i = 1, 2, \ldots, m).$$



Laufzeitkomplexität von `mvm`

Aufgabe 0.16

Was ist die worst-case Laufzeit von `mvm`?



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.16

Wir müssen jeden der m Einträge des Vektors b berechnen. Für jeden Eintrag ist eine Summe von n Produkten zu bestimmen. Für jeden Eintrag des Vektors b sind also n Multiplikationen und $(n-1)$ Additionen notwendig. Insgesamt sind also $m \cdot (n + (n-1))$ Operationen notwendig. Die asymptotische Laufzeit von `mvm` ist somit $\Theta(m \cdot n)$. Falls A eine quadratische $n \times n$ Matrix ist, dann ist die Laufzeit gegeben durch $\Theta(n^2)$ (quadratische Laufzeit).



Matrix-Matrix-Multiplikation

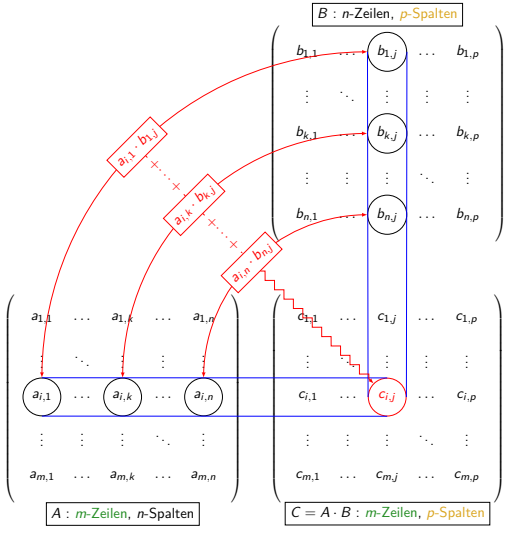


Abbildung: Veranschaulichung des Matrixprodukts $C = AB$.

Im Allgemeinen ist der Eintrag $c_{i,j}$ von C das Vektor-Vektor-Produkt[2] der i-ten Zeile von A mit der j-ten Spalte von B. Deshalb ist es notwendig, dass die „Breite“ von A genau der „Höhe“ von B entspricht. Das Produkt $C = AB$ wird die „Höhe“ von A und die „Breite“ von B haben. Formal definiert man die Einträge von C wie folgt:

$$c_{i,j} := \sum_{k=1}^{n} a_{i,k} b_{k,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \ldots + a_{i,n} b_{n,j}, \tag{4}$$

für $(i = 1, 2, \ldots, m \quad j = 1, 2, \ldots, p)$.

[2] Dieses enstpricht dem standard Skalarprodukt, wenn sowohl die i-te Zeile von A als auch die j-te Spalte von C jeweils als Spaltenvektoren aufgefasst werden.

Matrix-Matrix-Multiplikation

```
def mmm(A, B, num_rows, num_cols):
    C = [0] * (num_rows * num_cols)

    for i in range(num_rows):
        for j in range(num_cols):
            for k in range(num_cols):
                C[i * num_cols + j] += A[i * num_cols + k] * B[k * num_cols + j]

    return C
```

Programm: `mmm.py`



FreeFlower

Laufzeitkomplexität von `mmm`

Aufgabe 0.17

Was ist die worst-case Laufzeit von `mmm`?



FreeFlower

✔ Lösungsvorschlag zu Aufgabe 0.17

Wir müssen jeden der $m \cdot p$ Einträge der Matrix C berechnen. Für jeden Eintrag $c_{i,j}$ ist eine Summe von n Produkten zu bestimmen. Für jeden Eintrag der Matrix C sind also n Multiplikationen und $(n-1)$ Additionen notwendig. Insgesamt sind also $m \cdot p \cdot (n + (n-1))$ Operationen notwendig. Die asymptotische Laufzeit von `mvm` ist somit $\Theta(m \cdot p \cdot n)$. Falls A und B jeweils quadratische $n \times n$ Matrizen sind, dann ist die Laufzeit gegeben durch $\Theta(n^3)$ (kubische Laufzeit).



Die folgende schematische Darstellung soll nochmals die Situation der Matrixdimensionen in dem Produkt $C = AB$ verdeutlichen. In den letzten beiden Beispielen sind A und B jeweils Vektoren derselben Länge.



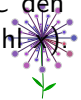
$$A \cdot B = C$$

$$A \cdot B = C$$

$$A \cdot B = C \in \mathbb{R}$$

$$A \cdot B = C$$

Die Matrix C ist so „hoch“ wie A und so „breit“ wie B. Das Produkt AB ist nur definiert, wenn B so „hoch“ ist, wie A „breit“ ist. Hat eine reelle 1×1-Matrix C den Eintrag x, so identifizieren wir die Matrix C mit ihrem Eintrag x (der reellen Zahl



Wunsch nach Klassifizierung von Berechnungsproblemen

- Bis jetzt haben wir die Zeitkomplexität von Algorithmen definiert.



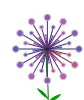
Wunsch nach Klassifizierung von Berechnungsproblemen

- Bis jetzt haben wir die Zeitkomplexität von Algorithmen definiert.
- Was uns aber primär interessiert, ist die Komplexität von Berechnungsproblemen, welche wir gerne nach dieser Komplexität klassifizieren würden.



Wunsch nach Klassifizierung von Berechnungsproblemen

- Bis jetzt haben wir die Zeitkomplexität von Algorithmen definiert.
- Was uns aber primär interessiert, ist die Komplexität von Berechnungsproblemen, welche wir gerne nach dieser Komplexität klassifizieren würden.
- Intuitiv würde man gerne sagen, dass die Zeitkomplexität eines Problems P der Zeitkomplexität eines asymptotisch optimalen Algorithmus für P ist.



Wunsch nach Klassifizierung von Berechnungsproblemen

- Unter asymptotischen Optimalität eines Algorithmus A für das Problem P versteht man, dass keine Algorithmus B für P existiert, welcher asymptotisch besser (schneller) ist als A.



Wunsch nach Klassifizierung von Berechnungsproblemen

- Unter asymptotischen Optimalität eines Algorithmus A für das Problem P versteht man, dass keine Algorithmus B für P existiert, welcher asymptotisch besser (schneller) ist als A.
- Die Idee wäre also, die Komplexität eines Berechnungsproblems als die Komplexität eines „optimalen“ Algorithmus für dieses Problem zu betrachten.



Wunsch nach Klassifizierung von Berechnungsproblemen

- Unter asymptotischen Optimalität eines Algorithmus A für das Problem P versteht man, dass keine Algorithmus B für P existiert, welcher asymptotisch besser (schneller) ist als A.
- Die Idee wäre also, die Komplexität eines Berechnungsproblems als die Komplexität eines „optimalen“ Algorithmus für dieses Problem zu betrachten.
- Diese Idee scheint sehr natürlich und vernünftig zu sein.



Wunsch nach Klassifizierung von Berechnungsproblemen

- Unter asymptotischen Optimalität eines Algorithmus A für das Problem P versteht man, dass keine Algorithmus B für P existiert, welcher asymptotisch besser (schneller) ist als A.
- Die Idee wäre also, die Komplexität eines Berechnungsproblems als die Komplexität eines „optimalen“ Algorithmus für dieses Problem zu betrachten.
- Diese Idee scheint sehr natürlich und vernünftig zu sein.
- Erstaunlicherweise gibt es aber Berechnungsprobleme, für welche beweisbar kein optimaler Algorithmus im obigen Sinne exisitert.



Blum Speed-Up-Theorem

Theorem (Blum-Speed-Up-Theorem, 1967)

Es existiert ein algorithmisches Entscheidungsproblem E (dies ist insbesondere ein Berechnungsproblem), sodass für jeden Algorithmus A^0, welcher E entscheidet, ein Algorithmus A^1 existiert, welcher ebenfalls E entscheidet, und für den gilt

$$T_{A^1}(n) \leq \log_2 (T_{A^0}(n))$$

für unendlich viele Grössen von Probleminstanzen $n \in \mathbb{N}$.



Blum Speed-Up-Theorem

Das Blum Speed-Up-Theorem besagt, dass es Probleme gibt, für die man jeden gegebenen Algorithmus für dieses Problem wesentlich verbessern kann. Dies bedeutet, dass für solche Probleme keine optimalen Algorithmen existieren kann. Damit lässt sich für diese Probleme die Komplexität nicht in dem oben beschriebenen Sinne (durch einen optimalen Algorithmus) definieren.



obere und untere Schranke

Deswegen spricht man in der Komplexitätstheorie nur über obere und untere Schranken für die Komplexität eines Problems im Sinne der folgenden Definition:

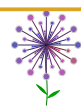
Definition (Obere- und untere Schranke, Optimalität)

Sei P ein Berechnungsproblem und $g(n)$ eine Funktion.

Obere Schranke für P: Wir nennen $O(g(n))$ eine obere Schranke für die Komplexität von P, wenn es einen Algorithmus A gibt, der P löst und dessen Worst-Case Laufzeit $T_A(n) \in O(g(n))$ erfüllt.

Untere Schranke für P: Wir nennen $\Omega(g(n))$ eine untere Schranke für die Komplexität von P, wenn für **jeden** Algorithmus A, der P löst, gilt: $T_A(n) \in \Omega(g(n))$.

Optimalität: Ein Algorithmus A heisst *optimal* für P, wenn $\Omega(g(n))$ eine untere Schranke für die Komplexität von P ist und zugleich $T_A(n) \in O(g(n))$ gilt.



FreeFlower