Komplexität

30. Januar 2020

Definition der Komplexitätsklassen

Sei $f, g : \mathbb{N} \to \mathbb{R}_+$ zwei Funktionen. Dann ist

Sei $f, g : \mathbb{N} \to \mathbb{R}_+$ zwei Funktionen. Dann ist

- $g(n) = \Omega(f(n))$, wenn es Konstanten $c, N > 0$ gibt, sodass

$$g(n) \geq c\, f(n) \qquad \text{für alle } n \geq N\,.$$

Sei $f, g : \mathbb{N} \to \mathbb{R}_+$ zwei Funktionen. Dann ist

- $g(n) = \Omega(f(n))$, wenn es Konstanten $c, N > 0$ gibt, sodass
$$g(n) \geq c\, f(n) \qquad \text{für alle } n \geq N\,.$$

- $g(n) = \mathcal{O}(f(n))$, wenn es Konstanten $c, N > 0$ gibt, sodass
$$g(n) \leq c\, f(n) \qquad \text{für alle } n \geq N\,.$$

Sei $f, g : \mathbb{N} \to \mathbb{R}_+$ zwei Funktionen. Dann ist

- $g(n) = \Omega(f(n))$, wenn es Konstanten $c, N > 0$ gibt, sodass
$$g(n) \geq c\, f(n) \qquad \text{für alle } n \geq N\,.$$
- $g(n) = \mathcal{O}(f(n))$, wenn es Konstanten $c, N > 0$ gibt, sodass
$$g(n) \leq c\, f(n) \qquad \text{für alle } n \geq N\,.$$
- $g(n) = \Theta(f(n))$, wenn
$$g(n) = \Omega(f(n)) \quad \text{und} \quad g(n) = \mathcal{O}(f(n))\,.$$

Bemerkungen

- Die Konstanten müssen von n unabhängig sein.
- Die wichtigste Klasse ist $\mathcal{O}(\cdot)$.
- $g(n)$ entspricht später der Laufzeit eines Algorithmus in Abhängigkeit der Eingabegröße n.
  Sagen dann: "Der Algorithmus wächst wie $f(n)$"

Beispiele

Sei $g(n) = 2n$.
Behauptung: $g(n) = \mathcal{O}(n)$.

Beispiele

Sei $g(n) = 2n$.
Behauptung: $g(n) = \mathcal{O}(n)$.

Beweis.

Müssen zeigen: $\exists C, N$, sd. $2n \leq Cn$ für alle $n \geq N$.

Beispiele

Sei $g(n) = 2n$.
Behauptung: $g(n) = \mathcal{O}(n)$.

Beweis.

Müssen zeigen: $\exists C, N$, sd. $2n \leq Cn$ für alle $n \geq N$.

Mit $C = 3$ und $N = 1$ klappt's (zum Beispiel). $\square$

Beispiele

Sei $g(n) = 2n$.
Behauptung: $g(n) = \mathcal{O}(n)$.

Beweis.

Müssen zeigen: $\exists C, N,$ sd. $2n \leq Cn$ für alle $n \geq N$.

Mit $C = 3$ und $N = 1$ klappt's (zum Beispiel). $\square$

Das funktioniert natürlich genau so mit $g(n) = 3n$, $g(n) = 4n$ etc. Also

$$\text{Für} \quad g(n) = kn \quad \text{gilt} \quad g(n) = \mathcal{O}(n).$$

In anderen Worten:

Der Koeffizient vor dem n spielt keine Rolle.

Beispiele

Sei $g(n) = n$.
Behauptung: $g(n) = \mathcal{O}(\frac{1}{3}n)$.

Beispiele

Sei $g(n) = n$.
Behauptung: $g(n) = \mathcal{O}(\frac{1}{3}n)$.

Beweis.

Müssen zeigen: $\exists C, N$, sd. $n \leq \frac{C}{3}n$ für alle $n \geq N$.

Beispiele

Sei $g(n) = n$.
Behauptung: $g(n) = \mathcal{O}(\frac{1}{3}n)$.

Beweis.

Müssen zeigen: $\exists C, N$, sd. $n \leq \frac{C}{3}n$ für alle $n \geq N$.

Mit $C = 4$ und $N = 1$ klappt's (zum Beispiel). $\square$

Beispiele

Sei $g(n) = n$.
Behauptung: $g(n) = \mathcal{O}(\frac{1}{3}n)$.

Beweis.

Müssen zeigen: $\exists C, N$, sd. $n \leq \frac{C}{3}n$ für alle $n \geq N$.

Mit $C = 4$ und $N = 1$ klappt's (zum Beispiel). $\square$

Frage: Gilt auch

$$g(n) = \mathcal{O}(r\,n), \quad r \text{ beliebig?}$$

Beispiele

Sei $g(n) = n$.
Behauptung: $g(n) = \mathcal{O}(\frac{1}{3}n)$.

Beweis.

Müssen zeigen: $\exists C, N$, sd. $n \leq \frac{C}{3}n$ für alle $n \geq N$.

Mit $C = 4$ und $N = 1$ klappt's (zum Beispiel). $\square$

Frage: Gilt auch

$$g(n) = \mathcal{O}(r\,n), \quad r \text{ beliebig?}$$

Haben also gezeigt:

$$\mathcal{O}(n) = \mathcal{O}(r\,n)$$

für $r \in \mathbb{R}$ beliebig.

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) \neq \mathcal{O}(n)$.

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) \neq \mathcal{O}(n)$.

Beweis.

Es müsste c existieren, sd. $n^2 \leq cn$, bzw. $n \leq c$.

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) \neq \mathcal{O}(n)$.

Beweis.

Es müsste c existieren, sd. $n^2 \leq cn$, bzw. $n \leq c$. Da c von n unabhängig sein muss, kann das nicht funktionieren (für festes c ist $n = c + 1 > c$). $\square$

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) \neq \mathcal{O}(n)$.

Beweis.

Es müsste c existieren, sd. $n^2 \leq cn$, bzw. $n \leq c$. Da c von n unabhängig sein muss, kann das nicht funktionieren (für festes c ist $n = c + 1 > c$). □

Allgemeiner gilt:

$$n^k \neq \mathcal{O}(n^l) \quad \text{für} \quad k > l\,.$$

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) = \mathcal{O}(n^3)$.

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) = \mathcal{O}(n^3)$.

Beweis.

Mit $c = 1$, $N = 1$ klappt's. $\square$

Beispiele

Sei $g(n) = n^2$.
Behauptung: $g(n) = \mathcal{O}(n^3)$.

Beweis.

Mit $c = 1$, $N = 1$ klappt's. $\square$

Allgemeiner gilt:

$$n^l = \mathcal{O}(n^k) \quad \text{für} \quad k > l\,.$$

Beispiele

Sei $g(n) = n^3 + 2n^2 + n$.
Behauptung: $g(n) = \mathcal{O}(n^3)$.

Beispiele

Sei $g(n) = n^3 + 2n^2 + n$.
Behauptung: $g(n) = \mathcal{O}(n^3)$.

Beweis.

Mit $c = 4$, $N = 1$ klappt's (nachrechnen). $\square$

Beispiele

Sei $g(n) = n^3 + 2n^2 + n$.
Behauptung: $g(n) = \mathcal{O}(n^3)$.

Beweis.

Mit $c = 4$, $N = 1$ klappt's (nachrechnen). $\square$

Allgemein: Ist $g(n) = \sum_{i=0}^{m} a_i n^i$ ein Polynom, dann gilt

$$g(n) = \mathcal{O}(n^m),$$

das heißt für Polynome ist nur der Grad entscheidend.

Einfache Komplexitätsanalysen

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
int sum(int a[], int n) {
  int res = 0;
  for (int i=0; i < n; i++) {
    res = res + a[i];
  }
  return res;
}
```

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
int sum(int a[], int n) {
  int res = 0;
  for (int i=0; i < n; i++) {
    res = res + a[i];
  }
  return res;
}
```

Müssen also zunächst herausfinden, wie oft die Schleife ausgeführt wird und dann, wie groß die Komplexität der Befehle innerhalb der Schleife ist.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
int sum(int a[], int n) {
  int res = 0;
  for (int i=0; i < n; i++) {
    res = res + a[i];
  }
  return res;
}
```

Müssen also zunächst herausfinden, wie oft die Schleife ausgeführt wird und dann, wie groß die Komplexität der Befehle innerhalb der Schleife ist.

- Befehl in der Schleife wird n-mal ausgeführt.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
1 int sum(int a[], int n) {
2   int res = 0;
3   for (int i=0; i < n; i++) {
4     res = res + a[i];
5   }
6   return res;
7 }
```

Müssen also zunächst herausfinden, wie oft die Schleife ausgeführt wird und dann, wie groß die Komplexität der Befehle innerhalb der Schleife ist.

- Befehl in der Schleife wird n-mal ausgeführt.
- Befehl in der Schleife (Zeile 4) hängt nicht von n hab, d.h. Komplexität $\mathcal{O}(1)$.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
1 int sum(int a[], int n) {
2   int res = 0;
3   for (int i=0; i < n; i++) {
4     res = res + a[i];
5   }
6   return res;
7 }
```

Müssen also zunächst herausfinden, wie oft die Schleife ausgeführt wird und dann, wie groß die Komplexität der Befehle innerhalb der Schleife ist.

- Befehl in der Schleife wird n-mal ausgeführt.
- Befehl in der Schleife (Zeile 4) hängt nicht von n hab, d.h. Komplexität $\mathcal{O}(1)$.

Insgesamt also n mal $\mathcal{O}(1)$, also $\mathcal{O}(n)$.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
int sum(Matrix mat) {
  int n = mat.size();
  int res = 0;

  for (int i=0; i < n; i++)
    for (int j=0; j < n; j++)
      res += mat[i][j];

  return res;
}
```

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
int sum(Matrix mat) {
  int n = mat.size();
  int res = 0;

  for (int i=0; i < n; i++)
    for (int j=0; j < n; j++)
      res += mat[i][j];

  return res;
}
```

- Äußere Schleife läuft n mal durch.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
int sum(Matrix mat) {
  int n = mat.size();
  int res = 0;

  for (int i=0; i < n; i++)
    for (int j=0; j < n; j++)
      res += mat[i][j];

  return res;
}
```

- Äußere Schleife läuft n mal durch.
- Innere Schleife läuft n mal durch.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
1 int sum(Matrix mat) {
2   int n = mat.size();
3   int res = 0;
4 
5   for (int i=0; i < n; i++)
6     for (int j=0; j < n; j++)
7       res += mat[i][j];
8 
9   return res;
10 }
```

- Äußere Schleife läuft n mal durch.
- Innere Schleife läuft n mal durch.
- Befehl in der Schleife (Zeile 7) hat konstante Komplexität.

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
1  int sum(Matrix mat) {
2    int n = mat.size();
3    int res = 0;
4  
5    for (int i=0; i < n; i++)
6      for (int j=0; j < n; j++)
7        res += mat[i][j];
8  
9    return res;
10 }
```

- Äußere Schleife läuft n mal durch.
- Innere Schleife läuft n mal durch.
- Befehl in der Schleife (Zeile 7) hat konstante Komplexität.

Insgesamt also $\mathcal{O}(n^2)$.

In welcher Komplexitätsklasse liegt der folgende Algorithmus?

```
int func(int n) {
  int x = 2;

  for (int i=0; i < n / 2; i++)
    x = x * 2;

  return res;
}
```

In welcher Komplexitätsklasse liegt der folgende Algorithmus?

```
int func(int n) {
  int x = 2;

  for (int i=0; i < n / 2; i++)
    x = x * 2;

  return res;
}
```

Der Algorithmus liegt in

$$\mathcal{O}(\frac{n}{2}).$$

In welcher Komplexitätsklasse liegt der folgende Algorithmus?

```
int func(int n) {
  int x = 2;

  for (int i=0; i < n / 2; i++)
    x = x * 2;

  return res;
}
```

Der Algorithmus liegt in

$$\mathcal{O}(\frac{n}{2}).$$

Haben vorhin gezeigt: $\mathcal{O}(n/2) = \mathcal{O}(n)$, d.h. der Algorithmus liegt in $\mathcal{O}(n)$.

Kompliziertere Komplexitätsanalysen

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```cpp
void selectionsort(std::vector<double> &list) {
  int n = list.size();
  for (int i=0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    std::swap(list[i], list[min]);
  }
}
```

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
void selectionsort(std::vector<double> &list) {
  int n = list.size();
  for (int i=0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    std::swap(list[i], list[min]);
  }
}
```

Behauptung. Selection-Sort liegt in $\mathcal{O}(n^2)$.

Beweis.

Tafel. □

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
template <class T>
void gausian_elimination(Matrix<T> &mat) {
  for (int i=0; i < mat.rows() - 1; i++) {
    for (int k=i+1; k < mat.rows(); k++) {
      double t = mat[k][i] / mat[i][i];
      for (int j=0; j < mat.cols(); j++) {
        mat[k][j] = mat[k][j] - t * mat[i][j];
      }
    }
  }
}
```

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
template <class T>
void gausian_elimination(Matrix<T> &mat) {
  for (int i=0; i < mat.rows() - 1; i++) {
    for (int k=i+1; k < mat.rows(); k++) {
      double t = mat[k][i] / mat[i][i];
      for (int j=0; j < mat.cols(); j++) {
        mat[k][j] = mat[k][j] - t * mat[i][j];
      }
    }
  }
}
```

Behauptung. Gauß-Elimination liegt in $\mathcal{O}(n^3)$

Beweis.

Übung. □

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
...
int k = 0;
for (int i=n/2; i <= n; i++) {
  for (int j=2; j <= n; j = j*2) {
    k = k + n / 2;
  }
}
...
```

Wollen die Komplexität des folgenden Algorithmus bestimmen.

```
...
int k = 0;
for (int i=n/2; i <= n; i++) {
  for (int j=2; j <= n; j = j*2) {
    k = k + n / 2;
  }
}
...
```

Behauptung. Der Algorithmus liegt in $\mathcal{O}(n \log n)$.

Beweis.

Übung.

- Überlege zuerst wie oft die innere Schleife durchläuft (j wird jedes mal verdoppelt).
- Überlege dann, wie oft die äußere Schleife durchläuft.

□

Wie oft wird *IPI* ausgegeben?

```
void fun(int n)
{
  for (int i=1; i<=n; i++)
    for (int j=1; j<=log(i); j++)
      std::cout << "IPI";
}
```

Wie oft wird *IPI* ausgegeben?

```
void fun(int n)
{
  for (int i=1; i<=n; i++)
    for (int j=1; j<=log(i); j++)
      std::cout << "IPI";
}
```

Übung.

Hinweise:

- $\log(a_1) + \log(a_2) + \ldots + \log(a_n) = log(a_1 \cdot a_2 \cdot \ldots \cdot a_n)$
- $\mathcal{O}(\log n!) = \mathcal{O}(n \log n)$ (Stirling Formel)