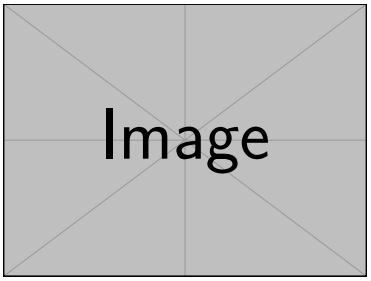
Notatki z wykładów

Matematyka – Analiza I



Twoje Imię i Nazwisko

Semestr zimowy 2025/2026

Hochschule Emden/Leer

Contents

Chapter 1

ChapterExample

1 Big O Notation

Die `Big O`-Notation beschreibt, wie stark die Laufzeit oder der Speicherverbrauch eines Algorithmus mit der Größe der Eingabe `n` wächst. Sie dient also zur Charakterisierung der Effizienz eines Algorithmus im **asymptotischen Grenzfall** – wenn `n` sehr groß wird.

1.1 Grundidee

Die Schreibweise `O(f(n))` bedeutet, dass die Laufzeit eines Algorithmus höchstens proportional zur Funktion `f(n)` wächst. Wenn also ein Algorithmus in `O(n)` arbeitet, wächst seine Ausführungszeit linear mit der Eingabegröße.

Hinweis

Die Big-O-Notation gibt keine exakte Laufzeit an, sondern das *Wachstumsverhalten*. Sie betrachtet nur den dominanten Term — also den Anteil, der für große `n` am stärksten wächst.

1.2 Häufige Komplexitätsklassen

- $O(1)$ – konstante Zeit (unabhängig von der Eingabegröße)
- $O(\log n)$ – logarithmisch (z. B. binäre Suche)
- $O(n)$ – linear (z. B. Schleife über ein Array)
- $O(n \log n)$ – n-log-n (z. B. effiziente Sortierverfahren)
- $O(n^2)$ – quadratisch (z. B. doppelt geschachtelte Schleifen)
- $O(2^n)$ – exponentiell (z. B. vollständige Kombinationssuche)

- $O(n!)$ – fakultativ (z. B. Permutationsprobleme)

1.3 Beispiele in C++

Beispiel 1 — konstante Komplexität `O(1)`

Diese Operation benötigt immer die gleiche Zeit, unabhängig von der Eingabegröße.

```cpp
int getFirstElement(const std::vector<int> &v) {
    return v[0]; // immer eine Operation
}
```

Beispiel 2 — logarithmische Komplexität $O(\log n)$

Die binäre Suche halbiert in jedem Schritt den Suchbereich.

```cpp
int binarySearch(const std::vector<int> &v, int target) {
    int left = 0, right = v.size() - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (v[mid] == target) return mid;
        else if (v[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Beispiel 3 — lineare Komplexität `O(n)`

Eine Schleife, die alle Elemente durchläuft, wächst linear mit der Eingabegröße.

```cpp
int sumElements(const std::vector<int> &v) {
    int sum = 0;
    for (int x : v) sum += x;
    return sum;
}
```

Beispiel 4 — n · log(n) Komplexität $O(n \log n)$

Sortieralgorithmen wie `std::sort()` oder MergeSort erreichen diese Effizienzklasse.

```cpp
void sortVector(std::vector<int> &v) {
    std::sort(v.begin(), v.end()); // O(n log n)
}
```

Beispiel 5 — quadratische Komplexität $O(n^2)$

Doppelte Schleifen über alle Elemente – z.B. einfacher Sortieralgorithmus.

```cpp
void bubbleSort(std::vector<int> &v) {
    for (size_t i = 0; i < v.size(); ++i)
        for (size_t j = 0; j < v.size() - 1; ++j)
            if (v[j] > v[j+1])
                std::swap(v[j], v[j+1]);
}
```

Beispiel 6 — exponentielle Komplexität $O(2^n)$

Eine rekursive Funktion, die alle Kombinationen prüft (z. B. Fibonacci ohne Memoization).

```cpp
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2); // O(2^n)
}
```

Beispiel 7 — Fakultätskomplexität $O(n!)$

Generierung aller Permutationen einer Liste — extrem ineffizient bei großen n.

```cpp
void permute(std::string s, int l, int r) {
    if (l == r) std::cout << s << std::endl;
    else {
        for (int i = l; i <= r; ++i) {
            std::swap(s[l], s[i]);
            permute(s, l + 1, r);
            std::swap(s[l], s[i]);
        }
    }
}
```

1.4 Wachstumsvergleich der Funktionen

Für große `n` spielen konstante Faktoren keine Rolle mehr. Entscheidend ist, wie schnell die jeweilige Funktion wächst:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$$

Fazit

In der Praxis sollten Algorithmen mit möglichst geringer Komplexität bevorzugt werden – idealerweise $O(1)$, $O(\log n)$ oder $O(n)$. Höhere Komplexitäten führen sehr schnell zu ineffizientem Verhalten bei großen Datenmengen.

2 Mastertheorem

Chodzi o porównanie szybkości wzrostu dwóch składników w równaniu rekurencyjnym. Jest to analiza dominacji w której sprawdzam, co bardziej wpływa na czas działania algorytmu. Pozwala to określic całkowity koszt $T(n)$, czyli króry składnik - rekurencja czy lokalna praca - dominuje.

Allgemeine Teilen und Herrschen

Folgende Gleichung heißt **Rekursive Gleichung**, weil T wieder durch T definiert wird (aber mit anderen Parametern).

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- $T(n) \rightarrow$ Gesamtlauf bei Eingabelänge na
- $a \rightarrow$ Anzahl der „Untergegebenen", $a \geq 1$
- $b \rightarrow$ Anteil der Eingabe pro Untergegebenem, $b > 1$
- $f(n) \rightarrow$ Zusatzaufwand bei Eingabelänge n

Die weiteren funktionieren durch Aufruf der $T(n)$ Funktion mit dem Argument $n = \frac{n}{b}$

$$T(\frac{n}{b}) = a \cdot T(\frac{n}{b^2}) + f(\frac{n}{b})$$

und so weiter

Hinweis

Die Untergebenen kriegen eine kurzere Aufgabe, somit sollen die nicht länger laufen als der Boss (weil die Angaben kürzer werden). Die untergebenen haben somit auf keinen Fall mehr zu tun als ihr Chef. Sie können aber insgesamt mehr oder weniger

machen. Das ist die Frage: wie viel arbeiten die Untergeordneten im vergleich zu Chef.

2.1 Fälle

Mastertheorem

Seien $a \geq 1, b > 1, f : \mathbb{N} \to \mathbb{N}$ und die Aufwandsfunktion $t(n)$ sei von der Form $t(n) = at(\frac{n}{b}) + f(n)$. Dann gilt (mit $c = \log_b a$ un $\epsilon > 0$)

Fall	Warunek	Co dominuje	wynik
1	$f(n) \in O(n^{c-\epsilon})$	rekurencja rośnie szybciej	$T(n) \in \Theta(n^c)$
2	$f(n) \in \Theta(n^c)$	obie części rosną tak samo	$T(n) \in \Theta(n^c \log n)$
3	$f(n) \in \Omega(n^{c+\epsilon})$ i $af(\frac{n}{b}) \leq c \cdot f(n)$	praca nierekurencyjna dominuje	$T(n) \in \Theta(f(n))$

Objaśnienie:

Fall 1 - rekurencja „zjada” większość czasu - podproblemów jest duo a lokalna praca ma mały wpływ.

Fall 2 - równowaga: praca w kadej warstwie kosztuje tyle samo czasu, więc sumuje się w dodatkowy $\log n$

Fall 3 - praca lokalna (np. scalanie, przetwarzanie) dominuje i decysuje o czasie działania

2.1.1 Fall 1

Definition

Falls $f(n) \in O(n^{c-\epsilon})$, dann ist $T(n) \in \Theta(n^c)$.

Für das Rekursionsschema

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

sei $c = \log_b a$. Fall 1 gilt, wenn $f(n)$ langsamer wächst als n^c, also

$$f(n) \in O(n^{c-\epsilon}) \quad \text{für ein } \epsilon > 0.$$

Im Beispiel

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

ist $a = 9$, $b = 3$, $c = \log_3 9 = 2$ und $f(n) = n$. Da $n = O(n^{2-\epsilon})$ für jedes $\epsilon < 1$, folgt nach dem Master-Theorem:

$$T(n) \in \Theta(n^2).$$

Podsumowanie

Wystarczy pokazać, że dla pewnego małego dodatniego ϵ (w praktyce: że $f(n)$ ma mniejszy wykładnik ni n^c) zachodzi

$$f(n) = O(n^{c-\epsilon})$$

2.1.2 Fall 2

Definition

Falls $f(n) \in O(n^c)$, dann ist $T(n) \in \Theta(n^c \log n)$.

Im Beispiel:

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

ist $a = 1$, $b = \frac{3}{2}$ und $f(n) = 1$.

$$n^c = n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

Daher gilt:

$$f(n) = 1 = \Theta(1)$$

und somit nach dem Master-Theorem:

$$T(n) \in \Theta(\log n)$$

Podsumowanie

W przypadku 2 funkcja $f(n)$ rośnie w tym samym tempie co n^c, czyli $f(n) \in \Theta(n^c)$. Wtedy całkowity czas wykonania zwiększa się o czynnik $\log n$:

$$T(n) \in \Theta(n^c \log n)$$

2.1.3 Fall 3

Definition

Falls $f(n) \in \Omega(n^{c+\epsilon})$ und $af(\frac{n}{b}) \leq cf(n)$, dann ist $T(n) \in \Theta(f(n))$.

Im Beispiel

$$T(n) = 3T(\frac{n}{4}) + n \lg n$$

ist $a = 3$, $b = 4$, $f(n) = n \lg n$, $\log_4 3 = 0.793$.

1. Kondition $f(n) \in \Omega(n^{c+\epsilon})$

$$f(n) = n \lg n = \Omega(n^{0.793+\epsilon}) \text{ mit } \epsilon > 0$$

dla $\epsilon = 0.2$ $n^{c+\epsilon} \approx n^1$. $n \lg n$ rośnie szybciej niż n^1, więc

$$f(n) = n \lg n = \Omega(n^1)$$

2. Kontition $af(\frac{n}{b}) \leq cf(n)$

$$3 \cdot \frac{n}{4} \lg \frac{n}{4} \leq c \cdot f(n) \; \forall n, c < 1$$

mit $c = \frac{3}{4}$

$$\frac{3}{4} n \cdot \lg \frac{n}{4} \leq c \cdot n \ln n \Rightarrow \frac{3}{4} n \cdot \log \frac{n}{4} \leq \frac{3}{4} n \ln n$$

Podsumowanie

W przypadku 3 funkcja $f(n)$ rośnie szybciej niż część rekurencyjna n^c, czyli $f(n) \in \Omega(n^{c+\epsilon})$ dla pewnego $\epsilon > 0$. Jeśli dodatkowo spełniony jest warunek regularności $af\left(\frac{n}{b}\right) \leq c \cdot f(n)$ dla pewnego $c < 1$, to dominującym składnikiem jest $f(n)$ i całkowity czas wykonania wynosi:

$$T(n) \in \Theta(f(n))$$

Chapter 2

Algorithmen

1 Sorting

`compareTo(T o)`

Słuy do porównywania obiektór tego samego typu. Zwraca:

- liczbę ujemną jesli `this < other`
- `0` jesli `this == other`
- liczbę dodatnią jesli `this > other`

Działa wewnątrz klasy i pochodzi z `Comparable`.

```
public class Student implements Comparable<Student> {
private String name;
private int age;

public Student(String name, int age) {
    this.name = name;
    this.age = age;
}

@Override
public int compareTo(Student other) {
    return Integer.compare(this.age, other.age);
}

@Override
public String toString() {
    return name + " (" + age + ")";
```

```
}



ic class Main {
public static void main(String[] args) {
    List<Student> students = Arrays.asList(
        new Student("Anna", 22),
        new Student("Kuba", 20),
        new Student("Ola", 25)
    );

    Collections.sort(students); // uses compareTo()
    System.out.println(students);
}
```

`comparator(T o)`

Świetne pytanie — Comparator robi to samo co Comparable, czyli porównuje obiekty, ale działa z zewnątrz, a nie wewnątrz klasy.

Cecha	Comparable	Comparator
Gdzie się definiuje	w klasie obiektu `implements Comparable`	w osobnej klasie lub lambdzie
Liczba moliwych porównań	Tylko jedno	dowolna liczba
Metoda	`compareTo(T other)`	`compare(T o1, T o2)`

Przykład

Załóżmy, że klasa `Student` nie implementuje `Comparable`:

```
public class Student {
    String name;
    int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
```

```
    public String toString() {
        return name + " (" + age + ")";
    }
}
```

Teraz tworzymy Comparator:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Anna", 22),
            new Student("Kuba", 20),
            new Student("Ola", 25)
        );

        // Comparator po wieku
        Comparator<Student> byAge = (s1, s2) -> Integer.compare(s1.age, s2
            .age);
        Collections.sort(students, byAge);

        System.out.println(students);
    }
}
```

Wynik to: `[Kuba (20), Anna (22), Ola (25)]`

Porównywanie obiektów w Java: `==` vs `equals()`

- `==` – porównuje **referencje obiektów**, czyli sprawdza, czy dwie zmienne wskazują na *ten sam obiekt w pamięci*.

$$a == b \Rightarrow \text{czy } a \text{ i } b \text{ to ten sam obiekt?}$$

- `equals()` – porównuje **zawartość obiektów**, czyli sprawdza, czy dane przechowywane w obiektach są takie same.

$$a.equals(b) \Rightarrow \text{czy } a \text{ i } b \text{ mają te same dane?}$$

- Domyślna implementacja `equals()` w klasie `Object` działa tak samo jak

`==` . Aby porównywać zawartość, należy ją **nadpisać**:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Student)) return false;
    Student s = (Student) o;
    return name.equals(s.name);
}
```

Podsumowanie:

Operator	Porównuje	Użycie
`==`	Referencję (adres pamięci)	Tożsamość obiektu
`equals()`	Zawartość (dane)	Równość logiczna