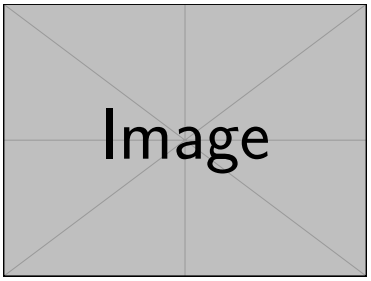
Notatki z wykładów

Matematyka – Analiza I



Twoje Imię i Nazwisko

Semestr zimowy 2025/2026

Hochschule Emden/Leer

Contents

1 ChapterExample **2**

1 Big O Notation 2

1.1 Grundidee 2

1.2 Häufige Komplexitätsklassen 2

1.3 Beispiele in C++ 3

1.4 Wachstumsvergleich der Funktionen 5

2 Algorithmen **6**

1 Sorting 6

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.

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