„Complexity“

Komplexität: in der Programmierung bezieht sich normalerweise auf die **Zeit- und Speicherkomplexität eines Algorithmus.**

- Betrachtet wird heutzutage nur die **Zeitkomplexität**, da unsere Rechner sehr große Speicherplatz hat

Zeitkomplexität hat drei Notation:

1. best case => Omega notation
2. average case => theta notation
3. worst case => Big-O-notation

Hinweis: Die Big-O-Notation wird verwendet, um die **Schrittanzahl** (oder Operationen) eines Algorithmus im Verhältnis zur Größe der Eingabe zu beschreiben, **nicht** die tatsächliche Zeit in Sekunden.

Hinweis: In der Tat gibt es für viele Algorithmen verschiedene Fälle, die als Best Case, Average Case und Worst Case bezeichnet werde

Beispiele:

1. Linear Search (Lineare Suche)

- **Best Case:** Der Zielwert (Target) befindet sich am Anfang des Arrays (Index 0). In diesem Fall wird der Zielwert in konstanter Zeit gefunden, also O(1).
- **Average Case:** Der Zielwert befindet sich irgendwo in der Mitte des Arrays. Hierdurch muss der Algorithmus im Durchschnitt die Hälfte der Elemente durchsuchen, was eine durchschnittliche Komplexität von O(n) ergibt.
- **Worst Case:** Der Zielwert befindet sich am Ende des Arrays oder ist gar nicht im Array enthalten. Der Algorithmus muss in diesem Fall alle Elemente durchsuchen, was zu einer linearen Komplexität von O(n) führt.

2. Binary Search (Binäre Suche)

- **Best Case:** Der Zielwert befindet sich genau in der Mitte des Arrays beim ersten Vergleich. In diesem Fall beträgt die Komplexität O(1).
- **Average Case:** Der Zielwert ist irgendwo im Array und die Anzahl der Schritte, um das Element zu finden, ist logarithmisch zur Größe des Arrays. Die durchschnittliche Komplexität beträgt hier O(log n).
- **Worst Case:** Der Zielwert befindet sich entweder am Anfang oder am Ende des Arrays nach mehreren Teilungen. Auch in diesem Fall bleibt die Komplexität O(log n), weil der Algorithmus das Array immer wieder halbiert.

- Betrachtet wird nur die **Big-O-Notation (O = Order):**

Die Big-O-Notation wird verwendet, um die Komplexität eines Algorithmus zu beschreiben. Sie gibt das schlimmste Szenario an, wie sich der Ressourcenbedarf (meistens die Zeit) verhält, wenn die Eingabegröße n wächst.

Einige übliche Big-O-Komplexitäten sind:

- **O(1):** Konstante Zeit, unabhängig von der Eingabegröße.
- **O(log n):** Logarithmische Zeit, steigt langsam an, selbst bei großen n.
- **O(n):** Lineare Zeit, proportional zur Eingabegröße.
- **O(n log n):** Logarithmisch-linear, eine Kombination aus linear und logarithmisch.
- **O(n^2):** Quadratische Zeit, wächst schnell mit der Eingabegröße.
- **O(2^n):** Exponentielle Zeit, wächst extrem schnell mit der Eingabegröße.

O(1): Konstante Zeit, unabhängig von der Eingabegröße

- Zugriff auf ein Element in einem Array

```
public int getElementAtIndex(int[] arr, int index) {
    return arr[index];
}
```

- Überprüfen, ob eine Zahl gerade ist

```
public boolean isEven(int number) {
    return number % 2 == 0;
}
```

O(log n): also zu verstehen ist, dass Z.B die Schleifenvariable oder der Suchraum in jeder Iteration nicht einfach nur um 1 erhöht wird (wie bei `i++`), sondern stattdessen in größeren Schritten verändert wird. Dies kann durch Multiplikation, Division oder andere Operationen erfolgen, die den Suchraum in jedem Schritt drastisch verkleinern oder vergrößern.

Beispiel:

```java
int left = 0, right = n - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) {
        // Element gefunden
        break;
    }
    if (arr[mid] < target) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
```

Erklärung: Hier wird der Bereich, in dem das Zielobjekt liegen könnte, in jeder Iteration halbiert. Dies führt zu einer logarithmischen Reduzierung der Anzahl möglicher Positionen.

O(n): hier auch zu Verstehen ist, dass die Laufzeit des Algorithmus direkt proportional zur Größe der Eingabe (`n`) ist. Wenn das Array oder die Liste größer wird, wächst die Laufzeit des Algorithmus linear mit der Eingabegröße.

Beispiel 1: Durchlaufen eines Arrays und Summieren der Elemente

```java
public int sumArray(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Erklärung: Die Schleife läuft von 0 bis zur Länge des Arrays (`arr.length`). Da jeder Schritt konstant ist und die Schleife genau `n` Mal durchlaufen wird (wobei `n` die Länge des Arrays ist), ist die Zeitkomplexität O(n).

O(n log n): Beispiel ist zu Verstehen

```
fort (int i =0; i < n;i++){
        for (int j = 0; j < n; j = j/2){
                sysout(„“);
        }
}
```

Erklärung: die eine Kombination aus einer linearen und einer logarithmischen Laufzeit beinhalten.

Noch weitere Infos:

Hier werden die Schritte gezählt!

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Grafisches Diagramm: