Wie schnell ist dein Code? – Einführung in die Laufzeitanalyse

Algorithmen, O-Notation und warum Sekunden keine gute Maßeinheit sind.

Eine Einführung für die gymnasiale Oberstufe

Das Problem: "Bei mir läuft es schnell..."

Supercomputer (Schlechter Code)

Alter Laptop (Guter Code)

Warum wir Laufzeit nicht einfach in Sekunden messen können (System.nanoTime()):

- **Hardware-Abhängigkeit**: Ein Supercomputer ist auch bei schlechtem Code schnell.
- **Auslastung**: Laufen im Hintergrund Updates, verfälscht das die Messung.
- **Eingabedaten**: Wir können nicht jede mögliche Eingabe testen.

Wir brauchen ein Maß, das unabhängig von der Hardware ist.
Wir zählen Schritte, nicht Sekunden.

Was ist Big-O? (Wachstum > Zeit)

Die O-Notation (Landau-Symbol) beschreibt nicht die exakte Dauer, sondern wie der Aufwand wächst, wenn die Eingabemenge n größer wird.

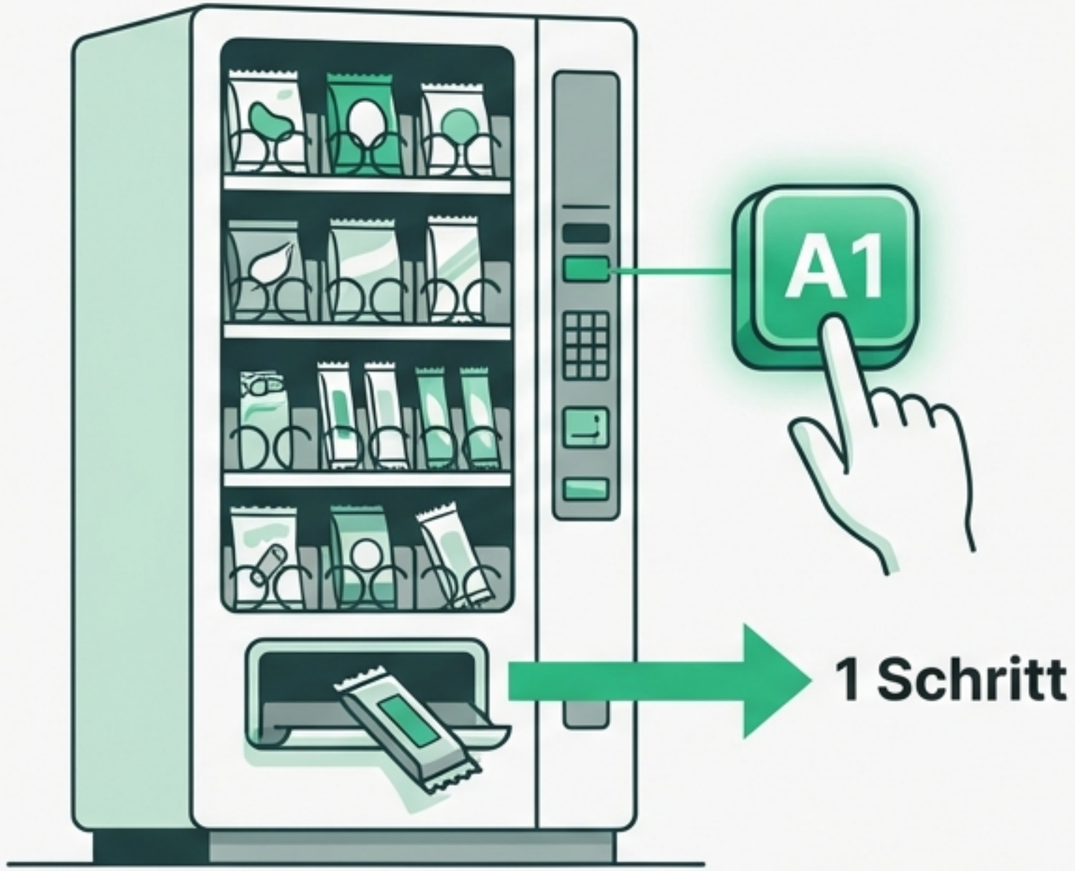
n = Die Größe der Eingabe (z. B. **Anzahl der Elemente in einer Liste).**
T(n) = **Die Anzahl der Schritte, die der Algorithmus benötigt.**

Linear: Ein Buch lesen.
10 Seiten dauern t.
20 Seiten dauern 2t.

Quadratisch:
Händeschütteln.
Bei 5 Personen wenige Handschläge, bei 100 Personen extrem viele.

O(1) – Konstanter Aufwand

"Der Instant-Zugriff"



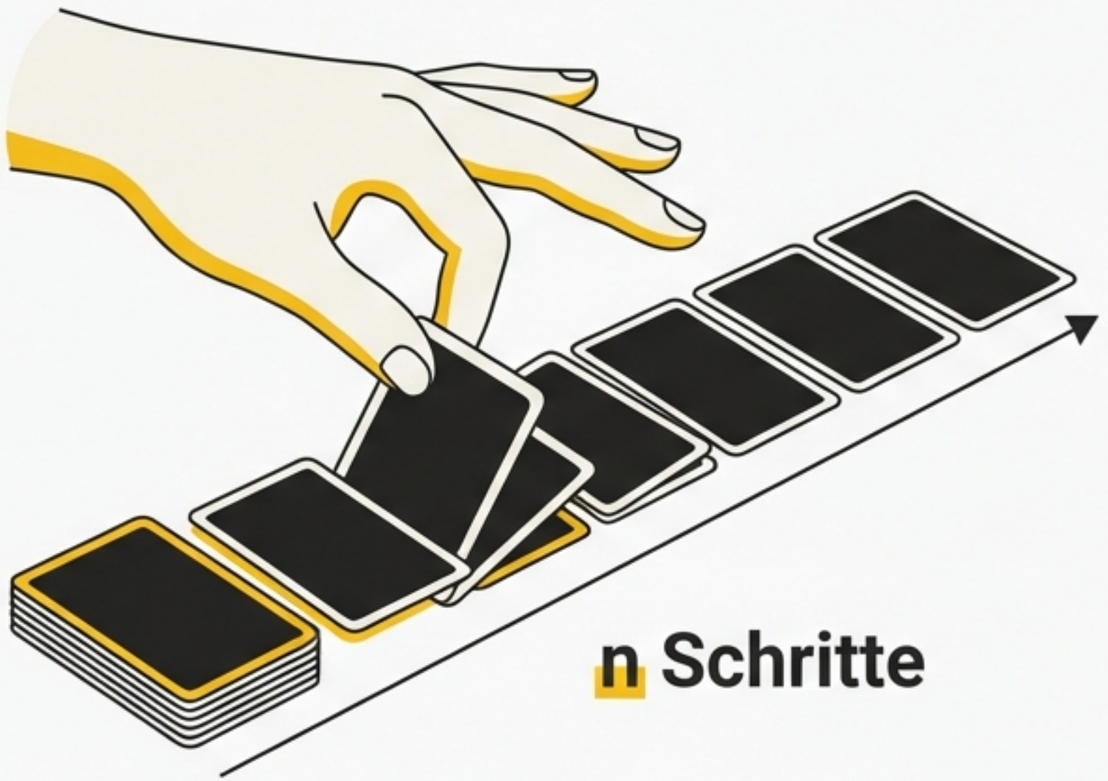
Die Laufzeit ist unabhängig von n. Egal wie viele Daten vorhanden sind, es dauert immer gleich lang.

```
// Zugriff auf ein Array-Element
int getElement(int[] array) {
    return array[0]; // Immer 1 Schritt
}
```

Beispiele: Zugriff auf einen Index arr[5], Prüfen ob eine Zahl gerade ist.

O(n) – Linearer Aufwand

"Der Seiten-Blätterer"

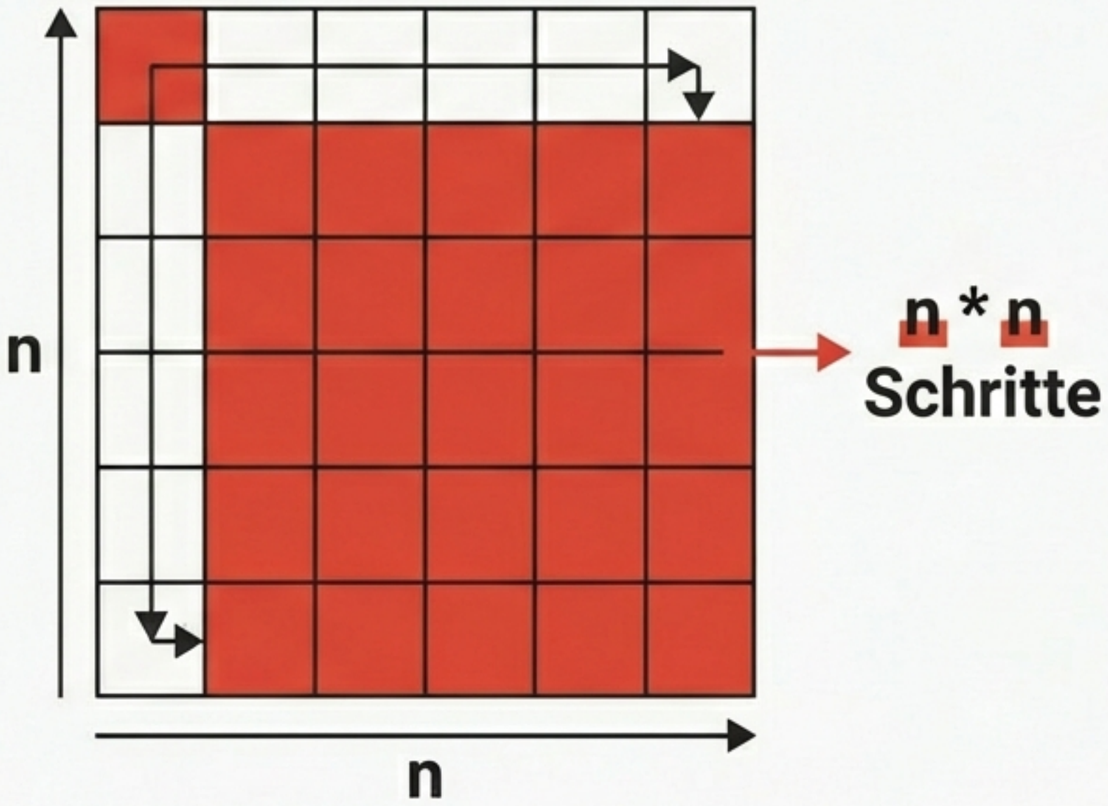


Verdoppelt sich die Eingabe n, verdoppelt sich auch die Laufzeit. Der Algorithmus muss jedes Element einmal 'anfassen'.

```
// Summe aller Zahlen
double sum(double[] x) {
    double s = 0;
    for (int i = 0; i < x.length; i++) { // Schleife läuft n-mal
        s = s + x[i];
    }
    return s;
}
```

O(n²) – Quadratischer Aufwand

"Die Bremse"

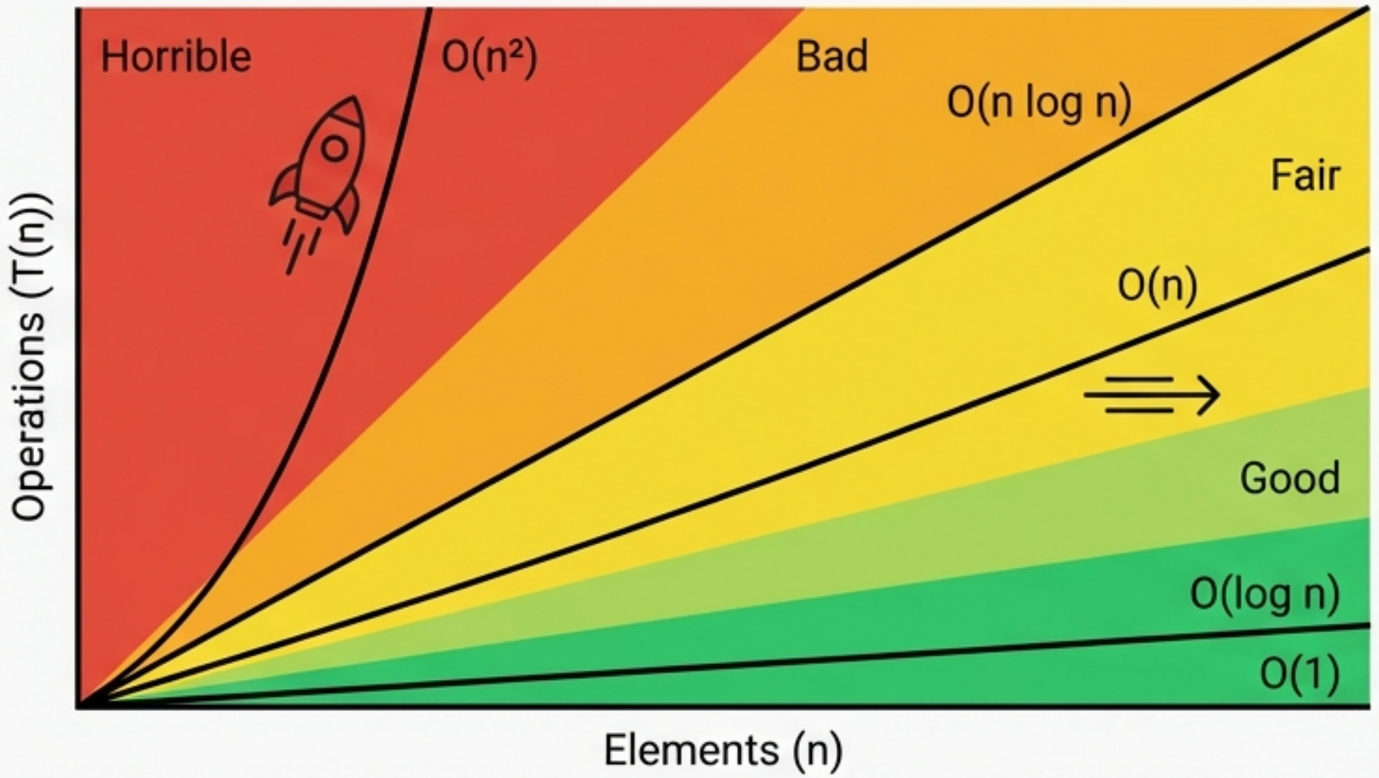


Eine Schleife in einer Schleife. Wenn sich n verdoppelt, vervierfacht sich der Aufwand ($2^2 = 4$).

```
// Verschachtelte Schleifen (z.B. Duplikate finden)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        check(i, j); // Wird n * n mal ausgeführt
    }
}
```

Real-World: Bubble Sort, oder jeden Schüler einer Schule mit jedem anderen vergleichen.

Der direkte Vergleich: Warum O(n²) gefährlich ist



- Für kleine n (z. B. 10 Elemente) sind die Unterschiede kaum spürbar.
- Für große n (z. B. 1.000.000) explodiert die Kurve von O(n²), während O(n) flach bleibt.

Skalierbarkeit bedeutet, dass dein Code auch bei großen Datenmengen nicht abstürzt.

O(log n) – Logarithmischer Aufwand

"Teile und Herrsche"

Suche im Telefonbuch

Suche ‘Müller’.
Mitte aufschlagen -> ‘M’ ist danach?
-> Linke Hälfte wegwerfen.
-> Wiederholen.

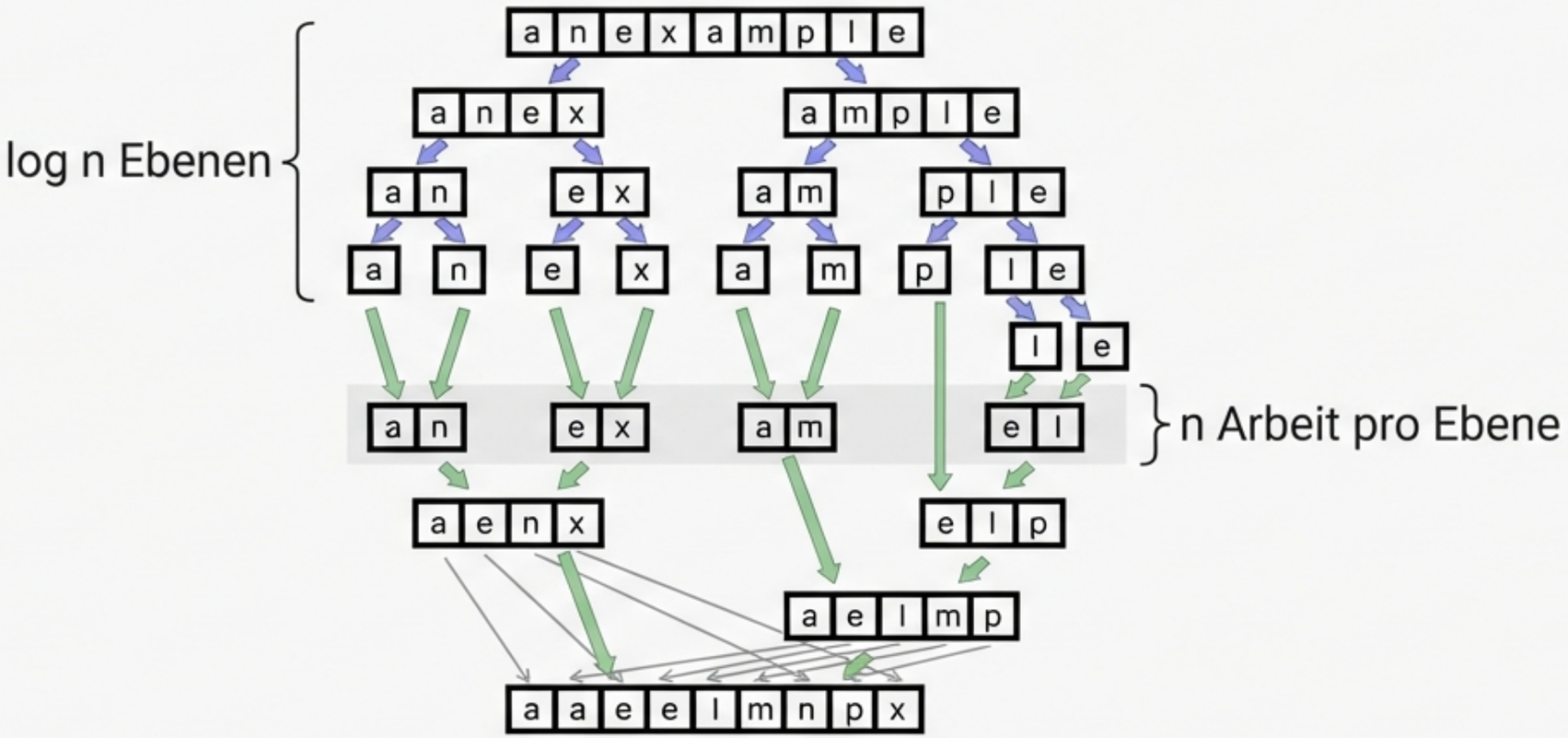
Bei 1.000.000 Einträgen nur ca. 20 Schritte!

$$2^x = n \rightarrow x = \log_2 n$$

Beispiel: Binäre Suche (Binary Search).
Voraussetzung: Daten müssen sortiert sein.

O(n log n) – Quasi-Linear

"Smartes Sortieren"



Besser als O(n^2), aber langsamer als O(n). Der Standard für gute Sortieralgorithmen (z. B. Mergesort, Heapsort).

Die Hierarchie der Komplexität

O(1) – Sofort (Array Index)

O(log n) – Sehr schnell (Binäre Suche)

O(n) – Okay (Lineare Suche)

O(n log n) – Gut für Sortieren (Mergesort)

O(n²) – Langsam (Bubble Sort)

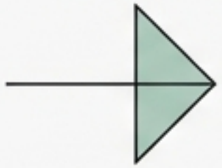
O(2^n) – Exponentiell (Passwort knacken)

O(n!) – Faktoriell (Traveling Salesman)

Warnung: Algorithmen ab $O(n^2)$ sollten bei großen Datenmengen vermieden werden.

Wie bestimme ich Big-O? (Die Faustregeln)

Du brauchst keine komplexe Grenzwertberechnung. Es reicht das “Eye-balling”.



Regel 1: Konstanten ignorieren

```
2n -> O(n)
500n -> O(n)
```

Bei n –> ∞ spielt der Faktor keine Rolle.

Regel 2: Dominanten Term finden

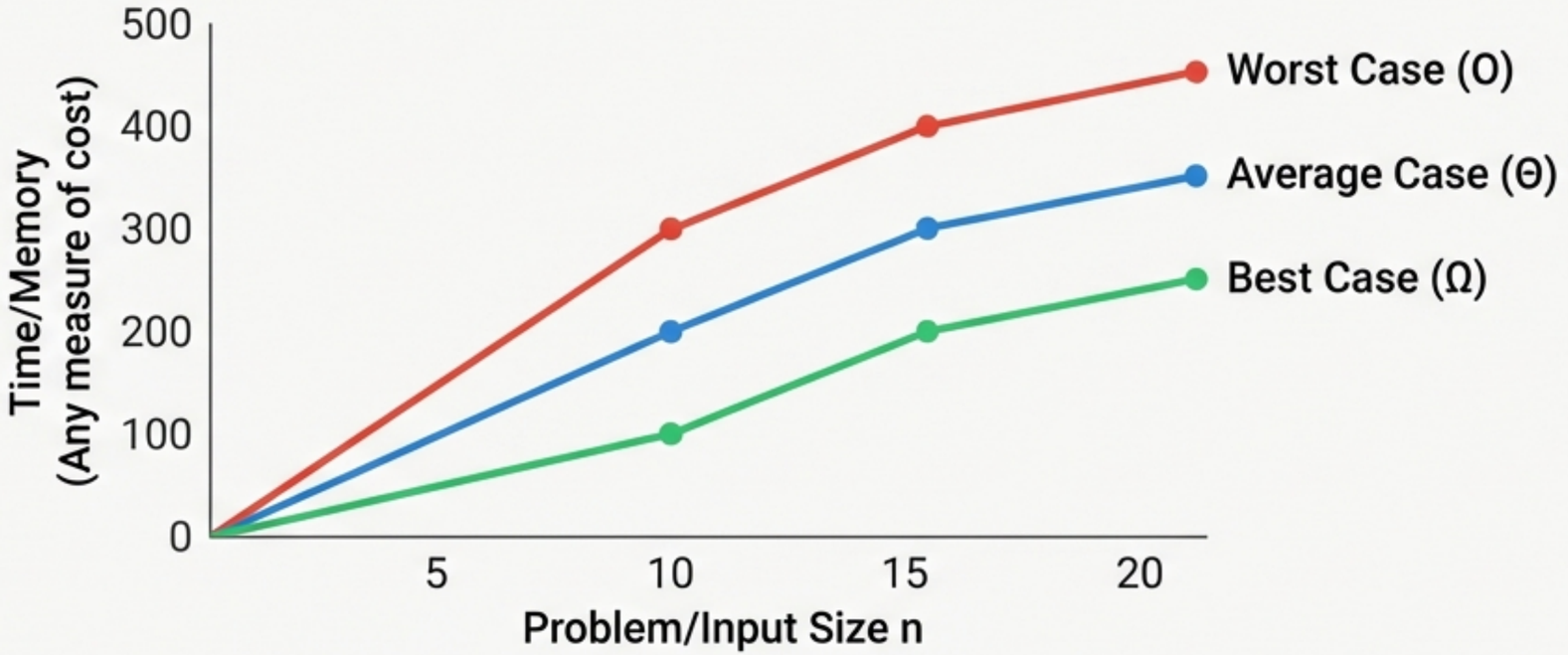
$n^2 + n + 1000 \rightarrow O(n^2)$

Der am stärksten wachsende Term “gewinnt”.

Übung: $f(n) = 3n^2 + 10n + 5$

Lösung: $O(n^2)$

Best Case, Worst Case & Average Case



Szenario: Lineare Suche nach einer Zahl

- Best Case (Ω): Die Zahl steht am Anfang (1 Schritt).
- Worst Case (O): Die Zahl steht am Ende oder fehlt (n Schritte).
- Average Case (Θ): Die Zahl steht in der Mitte (~n/2 Schritte).

Wichtig: In der Informatik gehen wir meistens vom Worst Case (O) aus ('Pessimistische Schätzung').

Realitäts-Check: Sekunden vs. Jahre

Vergleich von Sortieralgorithmen bei 1 Milliarde Elementen.

Quicksort / Mergesort (O(n log n))	Dauer: ca. 90 Sekunden
Insertion Sort / Bubble Sort ($O(n^2)$)	Dauer: ca. 85 Millionen Sekunden Das sind 2 Jahre und 8 Monate!

Ein besserer Algorithmus ist mehr wert als ein schnellerer Computer.

Analyse-Beispiel: Matrix-Multiplikation

```
// Matrix Multiplikation
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        c[i][j] = 0;
        for (k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

n mal

n mal

1. Äußere Schleife: n
2. Mittlere Schleife: n
3. Innere Schleife: n

Gesamt: $n \cdot n \cdot n = n^3$

Laufzeitkomplexität $O(n^3)$

Zusammenfassung: Was du dir merken solltest

- **Schritte > Sekunden**: Hardware ändert sich, Komplexität bleibt.
- **Ignoriere Konstanten**: Uns interessiert das grobe Wachstum.
- **Vorsicht bei Loops**: Verschachtelte Schleifen können Programme extrem verlangsamen ($O(n^2)$).
- **Teile und Herrsche**: Algorithmen wie Binäre Suche ($O(\log n)$) sind mächtige Werkzeuge.