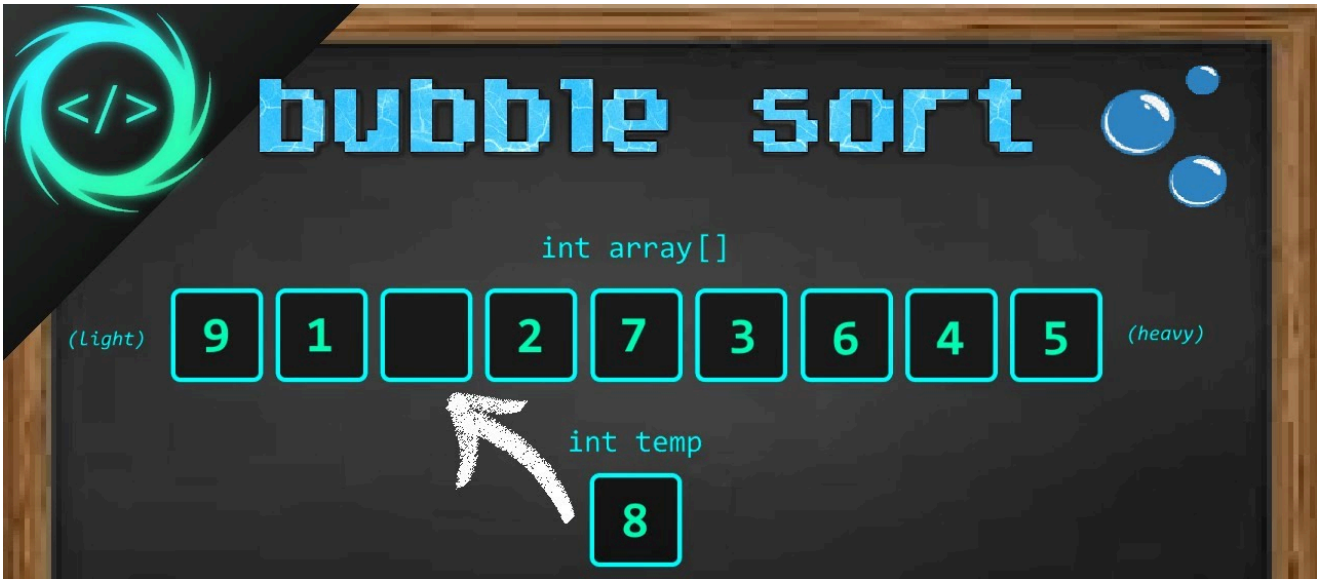
Sortier-Algorithmen

Bubble Sort



Namensgebung

Elemente werden mit einer Luftblase umschlossen. Luftblase = **Bubble**. Schwere Objekte sinken im Wasser nach unten, während leichte Objekte mit der **Luftblase** nach oben steigen.

Beschreibung

Ein Paar aus Elementen werden immer verglichen. Wenn ein Element größer ist tauschen sie Plätze.

Vorteil

simpel und einfach zu implementieren und zu warten.

Nachteil

Bei großen Datenmengen unbrauchbar. Muss immer beide Schleifen voll durchlaufen, selbst wenn die Liste schon sortiert ist.

Merkmale

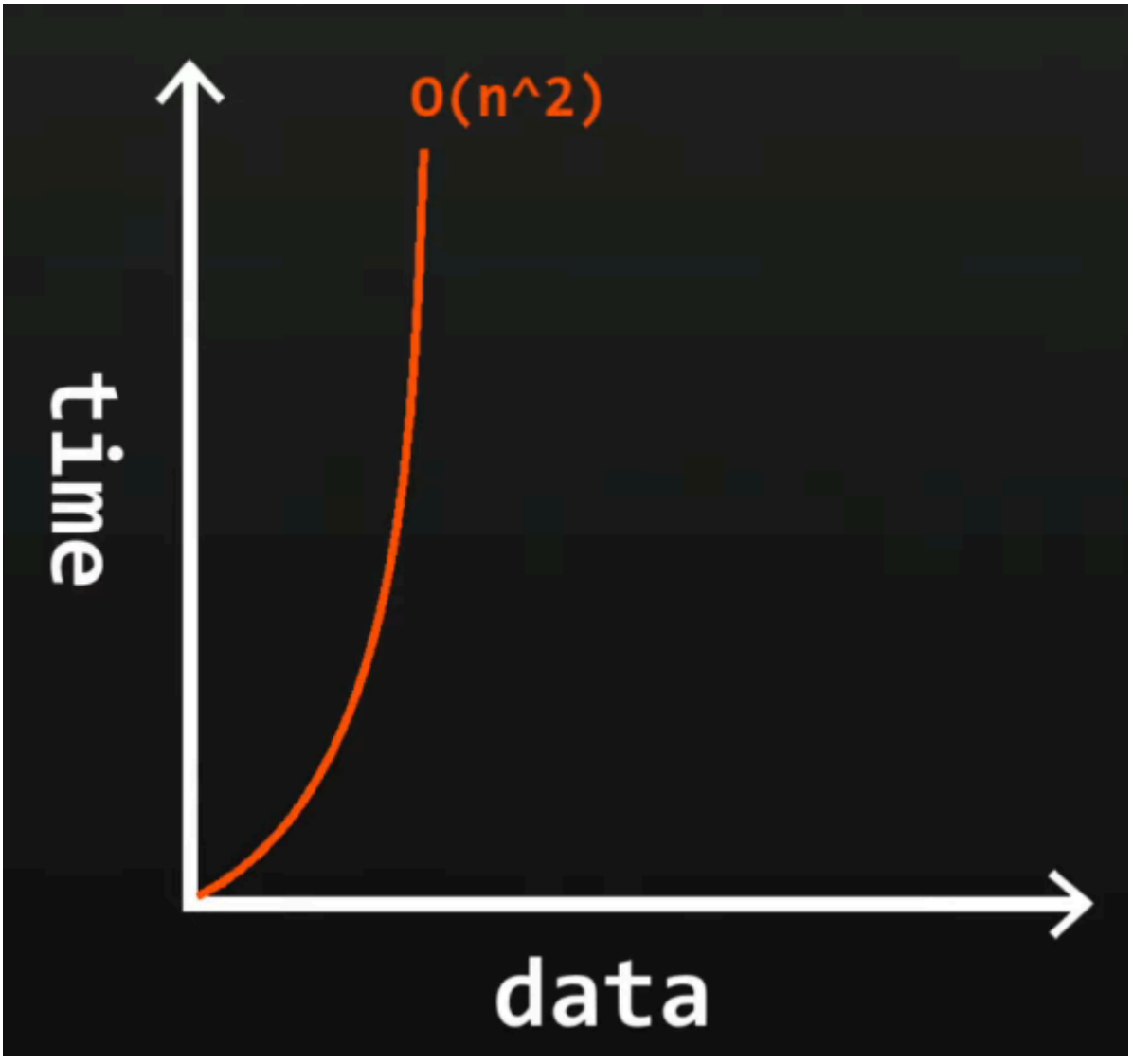
Zwei Schleifen die beide die Dauer des array.length haben.

Beispielcode

```
// ---------------------
// ---- BUBBLE-SORT ----
// https://www.youtube.com/watch?v=Dv4qLJcxus8

private static void bubbleSort(int[] array) {
```

```
    for (int k = 0; k <= array.length; k++) {
        for (int i = 1; i < array.length; i++) {
            int j = i - 1;
            int temp = array[j];
            if (array[i] < array[j]) {
                array[j] = array[i];
                array[i] = temp;
            }
        }
    }
}
```

Runtime Complexity / Big O



Insertion Sort

Namensgebung

Elemente werden verschoben wie bei einem **Schiebepuzzle**. Ist der richtige Platz durchs verschieben gefunden, wird das Element im Temporären Platz **inserted**.

Beschreibung

Ein Element aus der Liste wird in einem Temporären Platz gespeichert. Elemente werden nach links verschoben, so lange bis das Element im Temporären Speicher seinen Platz gefunden hat.

Vorteil

simpel und einfach wie Bubblesort, braucht aber weniger Schritte zum sortieren. Funktioniert gut für kleine und fast sortierte Sammlungen.

Nachteil

Ineffizient bei großen Sammlungen.

Merkmale

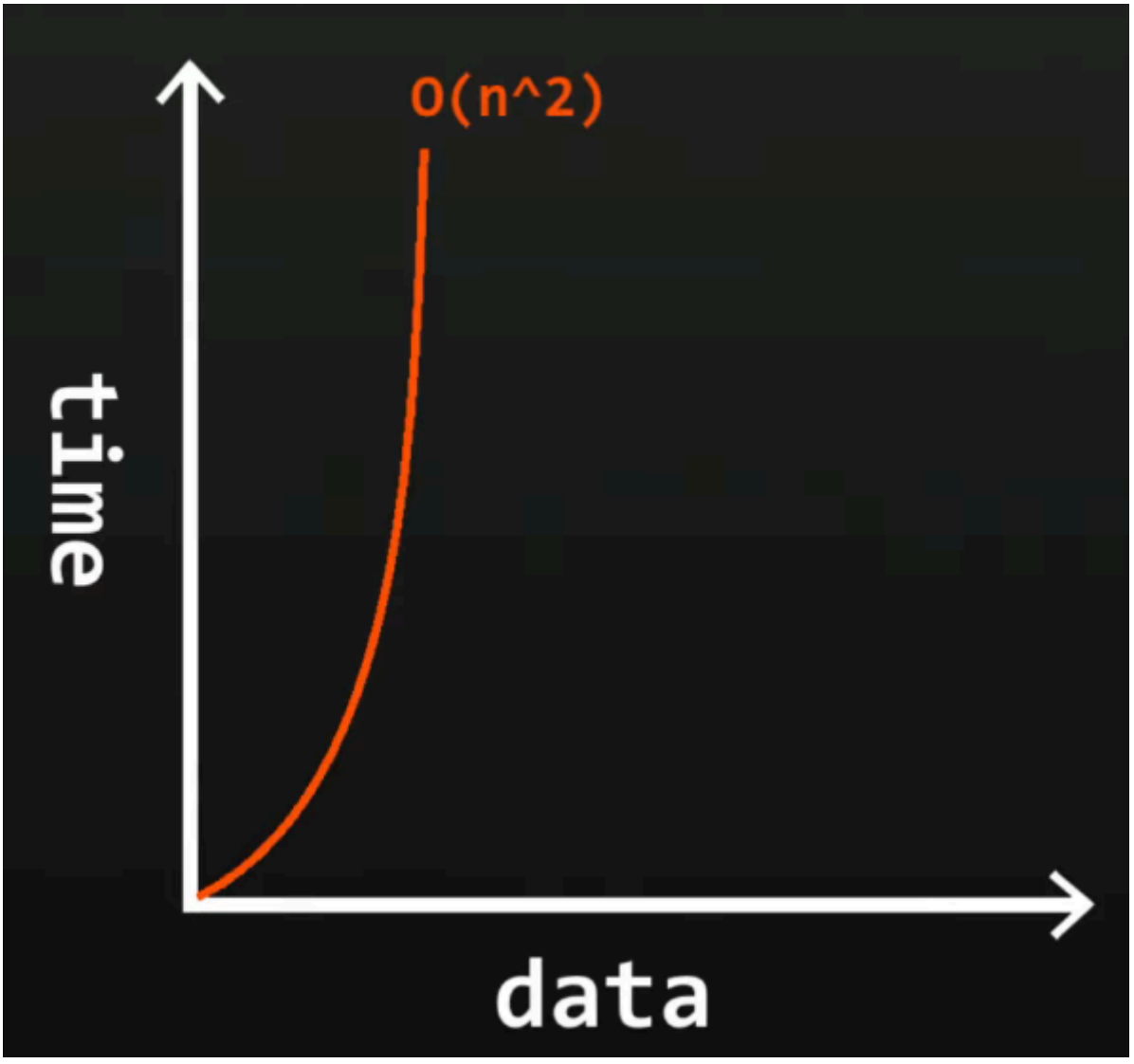
Erste Schleife fängt bei i=1 an, sprich auf Index 2. Die innere while-Schleife wird nur zum Verschieben größerer Elemente nach rechts benutzt.

Beispielcode

```
// ------------------------
// ---- INSERTION-SORT ----
// https://www.youtube.com/watch?v=8mJ-OhcfpYg

private static void insertionSort(int[] array) {

    for (int i = 1; i < array.length; i++) {
        int j = i - 1;
        int temp = array[i];
```

```
            while (j >= 0 && temp < array[j]) {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = temp;
        }
    }
```

Runtime Complexity / Big O



Selection Sort 📣🌞

Namensgebung

Mit einer **Taschenlampe** wird in jeden Indexplatz geschaut. Es ist sehr dunkel deswegen können wir uns nur eine Kiste gleichzeitig anschauen. Ein gefundenes Element das kleiner als der Minimumwert ist, wird dann **selected**.

Beschreibung

Der Index läuft mit jeder Iteration einmal komplett über die Sammlung. Dabei wird ein Element als "Minimum" Wert gespeichert. Jedes folgende Element wird daraufhin überprüft ob es noch kleiner als der aktuelle Minimumwert ist, falls ja wird ein neuer Minimumwert festgelegt. Nach einem ganzem Durchlauf wird der Minimumwert ganz links eingefügt, der Index springt einen platz weiter und beginnt von vorne.

Vorteil

Der Algorithmus hat die gleiche Laufzeit, unabhängig davon, ob die Liste bereits teilweise oder vollständig sortiert ist.

Nachteil

Je größer die Datenmenge, desto schlechter wird der Selectionsort.

Merkmale

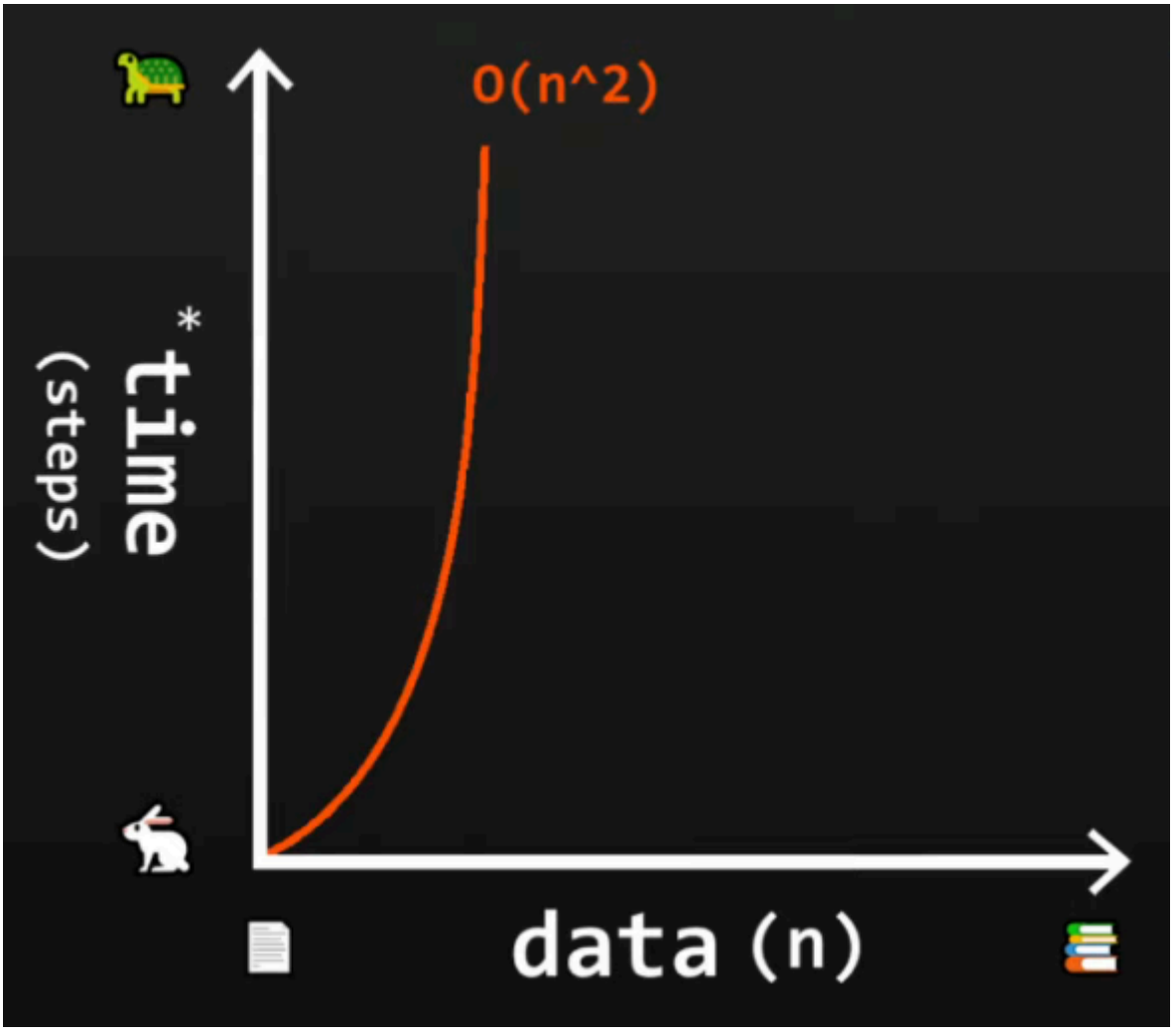
Zweifach verschachtelte Schleifen. Zwei Elemente werden miteinander in einer if-Bedingung verglichen um den neuen Minimumwert zu finden.

Beispielcode

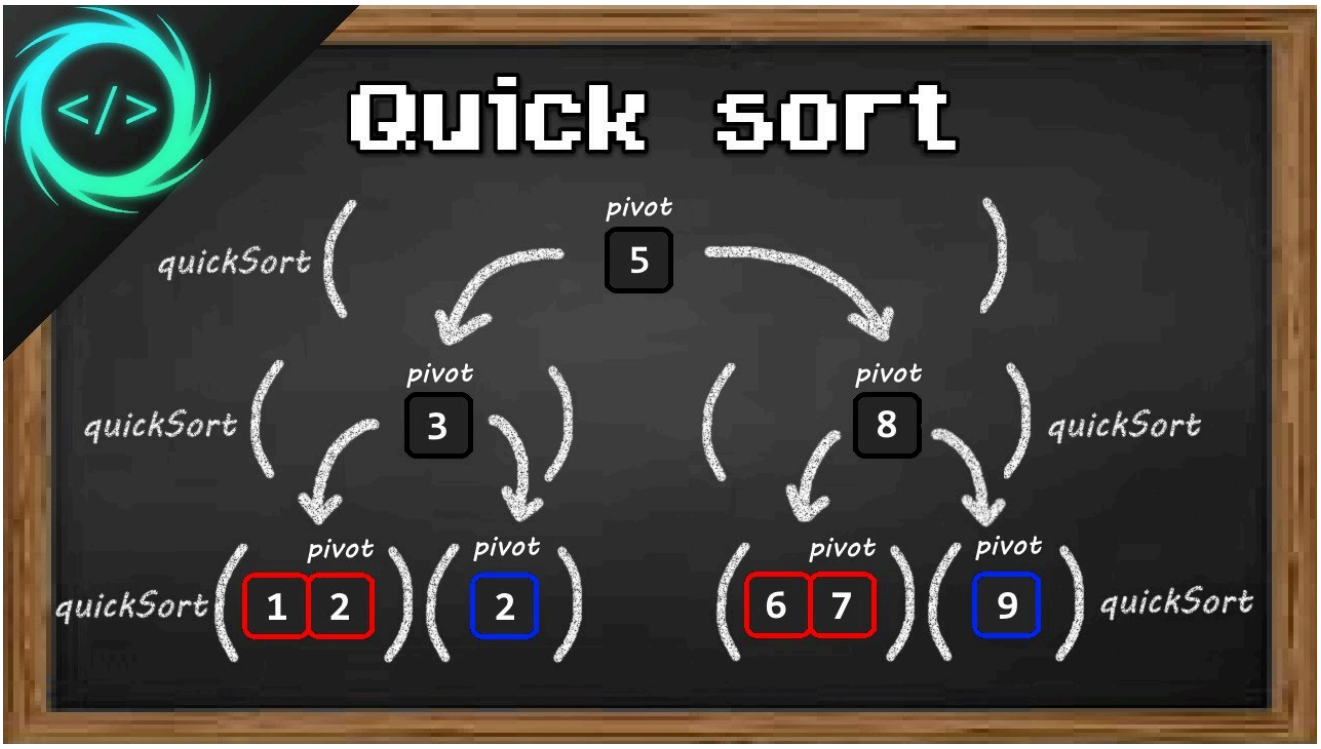
```
// ------------------------
// ---- SELECTION-SORT ----
// https://www.youtube.com/watch?v=EwjnF7rFLns

private static void selectionSort(int[] array) {

    for (int i = 0; i < array.length - 1; i++) {
```

```
        int min = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[min] > array[j]) {
                min = j;
            }
        }
        int temp = array[i];
        array[i] = array[min];
        array[min] = temp;
    }
}
```

Runtime Complexity / Big O



Quick Sort 🔨➗



Namensgebung

Der Quick Sort teilt die Sammlung immer in **zwei Hälften**. Mit einer **Axt** teilt er in jeder Iteration die Sammlung.

Beschreibung

Am Anfang des Quick Sort wird ein Pivot-Point festgelegt. Das ist meist das letzte Element in der Sammlung. Nun werden mit jeder Iteration die Elemente verglichen und Plätze getauscht. Kleinere Elemente werden nach links, größere nach rechts verschoben. Durch das Verhalten des Quick Sort wird der Pivot-Point seinen richtigen Platz nach der ersten Iteration einnehmen. Dies ist nun der Indexplatz an dem die Sammlung in zwei Hälften geteilt wird. Nun beginnt der Algorithmus bei beiden neuen Hälften zu sortierten usw.

Vorteil

Sehr gut bei großen, zufälligen Sammlungen. Ist im Durchschnitt schneller als Bubble oder Insertion Sort.

Nachteil

Worst case für einen Quick Sort ist eine vollständig sortierte Liste. Je sortierter die Sammlung für den Quick Sort ist, desto langsamer ist dieser Sortieralgorithmus.

Merkmale

Der Quick Sort braucht eine Hilfsmethode (im Beispiel "partition" genannt). Außerdem muss sich der Quick Sort rekursiv, selbst zweimal aufrufen. Ein Quick Sort braucht drei Übergabeparameter, die Sammlung, den Start und das Ende.

Beispielcode

```
// ------------------------
// ---- QUICK-SORT ----
// https://www.youtube.com/watch?v=Vtckgz38QHs&t=2s

private static void quickSort(int[] array, int start, int end) {

    if (end <= start) return;
    int pivot = partition(array, start, end);
    quickSort(array, start, pivot - 1);
    quickSort(array, pivot + 1, end);
}
private static int partition(int[] array, int start, int end){
    int pivot = array[end];
    int i = start - 1;
    for (int j = start; j <= end - 1; j++) {
        if (array[j] < pivot){
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    i++;
    int temp = array[i];
    array[i] = array[end];
    array[end] = temp;
    return i;
}
```

Runtime Complexity / Big O