Einheit 3 Objektorientierung I

Objektorientierung Programmierung

Lukas Wais

CODERS.BAY

Version: 8. Oktober 2023

Inhaltsverzeichnis I

Inhaltsverzeichnis II

Objektorientierung

Der Employee

Abbildung: Foto von Emmanuel Ikwuegbu auf Unsplash

Employee Klasse

```
public class Employee {
    String firstName; String lastName; int salary;

    void printName() {
      System.out.println("Name: " + firstName + " " + lastName);
    }

    void printSalary() {
      System.out.println("Salary of " + firstName +
        " " + lastName + " is: "  + salary + "$");
    }
}
```

Eigenschaften eines Autos

- Marke
- Modell
- Farbe
- Motor
- ...

Employee Klasse

```
public class Car {
  String brand;
  String model;
  Color color;
  Motor motor;

  public Car(String brand, String model, Color color, Motor motor) {
    this.brand = brand;
    this.modell = modell;
    this.color = color;
    this.motor = motor;
  }
}
```

Der Konstruktor

Gibt an welche Felder bei einer Instanzierung gesetzt werden **müssen**.
Jede Klasse hat einen leeren default Konstruktor `public Car() {}`.

```
public Car(String brand, String modell, Color color, Motor motor) {
  this.brand = brand;
  this.modell = modell;
  this.color = color;
  this.motor = motor;
}
```

Das Schlüsselwort `this` verweist auf die aktuelle Klasse.

Static

- **Statische Variablen** auch genannt Klassenvariablen existieren nur einmal für alle Instanzen.
  - Beispiel: eine globale statische Variable kann verwendet werden, um die Instanzen der Klasse zu zählen.
- **Statische Methoden** genannt Funktionen, sind nicht Instanzgebunden und gehören wie statische Variablen zur Klasse selbst.

Beispiel Instanzzähler

```
public class MyClass {
  static int counter = 0;

  public MyClass() {
    counter++;
  }
}
```

Objekte Vergleichen

Objekte Ausgeben

Sichtbarkeit

- `_` ... package private
- `private` ...
- `protected` ...
- `public` ...

Was ist Datenkapselung?

Getter

Setter

Properties in C#

Zeichenketten

Was sind Zeichenketten?

Konzeptuell sind Strings in Java nichts anderes als eine Sequenz von Unicode Characters. Als Beispiel, der String "cb\u2122"besteht aus den drei Unicode Characters c, b und ™.

Was sind Zeichenketten?

Konzeptuell sind Strings in Java nichts anderes als eine Sequenz von Unicode Characters. Als Beispiel, der String "cb\u2122"besteht aus den drei Unicode Characters c, b und ™.

Java hat **keinen** built-in String-Datentyp.

Stattdessen enthält die Java-Standardbibliothek eine Klasse namens String. Jede Zeichenkette ist eine Instanz dieser Klasse.

Einige Informationen zum Unicode findet unter `https://home.unicode.org/`.

Erstellung von Strings

```
String s = "";                  // an empty string
String name = "John Doe";
```

Substrings

Es ist möglich, eine Teilzeichenkette aus einer größeren Zeichenkette zu extrahieren. In dem folgenden Beispiel hat die Variable *s* den Wert "Hel".

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

Der zweite Parameter von substring ist die erste Position, die man nicht mehr kopieren will. Im Beispiel kopieren wir die Positionen 0, 1 und 2 (von Position 0 bis einschließlich Position 2).

Vorteile dieser Zählweise

Die Berechnung der Länge einer Teilzeichenkette ist einfach. Die Zeichenkette `s.substring(a,b)` hat die Länge $b - a$. Wenn wir das Beispiel von vorhin nehmen, hat "Hel"die Länge $3 - 0 = 3$.

Konkatenation = Verkettung

Wie das in den meisten Programmiersprachen üblich ist, wird dazu der + Operator verwendet.

```
String hello = "Hello";
String world = "World";

String helloWorld = hello + " " + world;

// helloWorld has the value "Hello World"
```

Verkettung mit nicht Strings

Wenn man Strings mit nicht Strings verkettet, so wird der nicht String automatisch konvertiert. Das wird oft bei Outputs verwendet.

```
int x = 3;
int y = 4;
int result = x + y;

System.out.println("The result is: " + result);
```

Delimiter

Für Trennzeichen verwendet man am besten die statische *join* Methode.

```
String sizes = String.join("/", "S", "M", "L", "XL");

// sizes has the value "S/M/L/XL"
```

Wiederholen

Seit Java 11 gibt es auch eine Methode für das Wiederholen

```
String repeated = "CB".repeat(3);

// repeated has the value "CBCBCB"
```

Strings sind unveränderlich

Die String-Klasse hat keine Methode um einen Character in einem existierenden String zu verändern. Wenn aus der Zeichenkette *greetings* "Hello", "Help!"werden soll, kann man nicht einfach die letzten beiden Stellen überschreiben.

Was ist also die Lösung?

Strings sind unveränderlich

Die String-Klasse hat keine Methode um einen Character in einem existierenden String zu verändern. Wenn aus der Zeichenkette *greetings* "Hello", "Help!"werden soll, kann man nicht einfach die letzten beiden Stellen überschreiben.

Was ist also die Lösung?

Wir konkatenieren den Substring, welchen wir behalten wollen.

```
String greeting = "Hello";
greeting = greeting.substring(0, 3) + "p!";
```

Unveränderlichkeit

Wichtig

Die Dokumentation beschreibt Objekte der Klasse String als unveränderlich. So wie die Zahl 3 immer 3 ist, enthält die Zeichenkette "*Hello*" immer die Code-Unit-Folge für die Zeichen H, e, l, l, o. Man kann diese Werte nicht ändern. Man kann jedoch den Inhalt der String-Variablen "*Hello*" ändern und sie auf eine andere Zeichenkette verweisen lassen, was man auch mit numerischen Variablen tun kann, die derzeit den Wert 3, enthalten und dann den Wert 4.

Ist das effizient?

Ist das effizient?

Es scheint simple und effizienter zu sein, wenn man nicht jedes Mal einen neuen String erstellen muss. Gut, es ist nicht effizient einen String zu erstellen der die Konkatenation von "*Hel*" und "*p*!" enthält.

Aber

Unveränderliche Zeichenketten haben einen sehr großen Vorteil: Der Compiler betrachtet die Strings als shared.

Wie funktioniert das? Stell dir vor, alle Strings sind in einem Behälter (Pool). String-Variablen zeigen dann auf die Position in diesem Behälter. Kopiert man nun eine String-Variable, teilen sich die originale Variable und die Kopie die gleichen Chars.

Zitat von Cay S. Horstmann aus seinem Buch Core Java

The language designers of Java decided that the efficiency of sharing outweighs the inefficiency of string editing by extracting substrings and concatenating. Most of the time you do not change strings, you just compare them.

Strings vergleichen

Zeichenkette **immer** mit der *equals* Methode vergleichen. Sie gibt true zurück, wenn sie gleich sind und false wenn nicht.

```
myString.equals(otherString);
"Hello".equals(greetings);
// if upper/lower case does not matter
"Hello".equalsIgnoreCase("hello");
```

Verwende **niemals** den Operator ==, um zwei Zeichenketten zu vergleichen, da er nur feststellt, ob zwei Zeichenketten am selben Ort gespeichert sind oder nicht. Natürlich müssen sie gleich sein, wenn sie am selben Ort gespeichert sind, aber es ist auch möglich, dass mehrere Kopien derselben Zeichenkette an verschiedenen Orten gespeichert sind.

Empty und null-Checks

- Um zu prüfen ob ein String leer ist:

```
if (str.length() == 0)
```

oder

```
if (str.equals(""))
```

- Für den null-Check

```
if (str == null)
```

StringBuilder

Gelegentlich muss man Strings aus kürzeren Strings zusammensetzen. Zum Beispiel in Schleifen oder bei Tastatureingaben. Du weißt bereits, dass bei der Verkettung einer Zeichenkette ein neues String-Objekt erstellt wird, was sehr ineffizient ist. Es ist zeitaufwendig und verschwendet Speicher. Die Verwendung eines StringBuilders vermeidet dieses Problem.

$$String \neq StringBuilder$$

Eine Umwandlung vom StringBuilder in String ist notwendig.

Beispiel StringBuilder

```
// first instance it
StringBuilder builder = new StringBuilder();

// then use it
builder.append(number); // appends a single int
builder.append(str);    //  appends a string
// …                   // you can append every data type

// if you are done create a string of it
String completed = builder.toString();
```

Mehrzeilige Strings

Seit Java 15 kann man Textblöcke verwenden, die die Formatierung beibehalten. Beachte dabei die Tabulatoren und Leerzeichen, sie sind unterschiedlich, sehen aber gleich aus. Vor allem beim Vergleichen macht das einen Unterschied.

```
String helloWorld = """
Hello
        formatted
World!
""";
```

Escape-Zeichen in Textblöcken

Es gibt ein Escape-Zeichen, das nur in Textblöcken funktioniert. Ein \ direkt vor dem Ende einer Zeile verbindet diese und die nächste Zeile.

```
"""
Hello, my name is John. \
Please enter your name: """

// is the same as

"Hello, my name is John. Please enter your name: "
```

Algorithmen und Komplexität

Was ist ein Algorithmus?

Was ist ein Algorithmus?

Definition laut Duden

Verfahren zur schrittweisen Umformung von Zeichenreihen; Rechenvorgang nach einem bestimmten [sich wiederholenden] Schema.

Formale Definition

Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.

Beispiele:

- Kochrezepte, Bedienungsanleitungen
- Programme welche Computer ausführen kann

Arten von Komplexitäten

1. **Zeitkomplexität:** Anzahl der Rechenschritte in Abhängigkeit von der Länge der Eingabe. Sie wird oft asymptotische Laufzeitkomplexität genannt.
2. **Platzkomplexität:** Minimaler Speicherbedarf in Abhängigkeit der Eingabelänge.

Beide Arten sind wichtige Merkmale für die Skalierbarkeit von Algorithmen.

Laundau-Symbole oder Big O - Notation

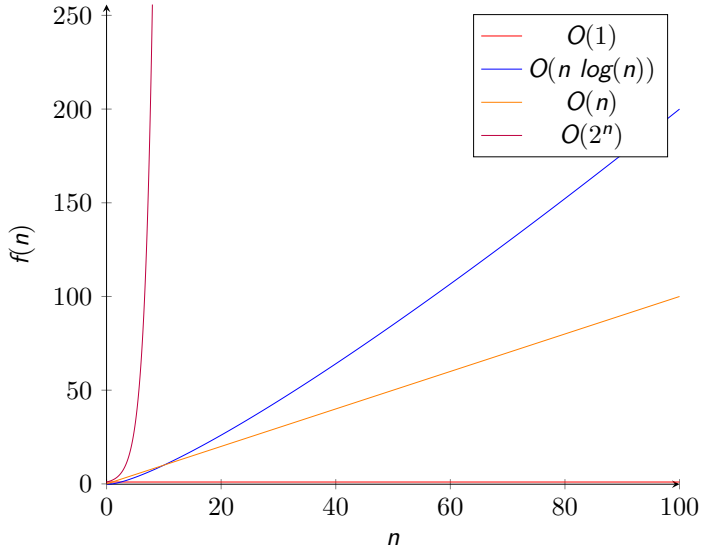
ein Auszug

Diese Symbole werden verwendet um das asymptotische Verhalten von Funktionen (und auch Folgen) zu beschreiben.

$f(n) = O(g(n))$	obere asymptotische Schranke	f wächst höchstens so schnell wie g
$f(n) = \Omega(g(n))$	untere asymptotische Schranke	f wächst mindestens so schnell wie g
$f(n) = \Theta(g(n))$	exakte asymptotische Schranke	f wächst genauso so schnell wie g

Als Kurzschreibweise kommt auch oft $f \in O(g)$ vor.

Beispiel



Erklärung Beispiel

- $O(1)$ f ist beschränkt und überschreitet einen konstanten Wert nicht.
  Beispiel: Arrayzugriff über den Index.
- $O(n\ log(n))$ super-lineares Wachstum. Vergleichbasierte Algorithmen zum Sortieren von n Zahlen.
- $O(n)$ lineares Wachstum.
- $O(2^n)$ exponentielles Wachstum. f wächst ungefähr auf das Doppelte, wenn sich n um 1 erhöht.

Erklärung Beispiel

- $O(1)$ *f* ist beschränkt und überschreitet einen konstanten Wert nicht.
  Beispiel: Arrayzugriff über den Index.
- $O(n\ log(n))$ super-lineares Wachstum. Vergleichbasierte Algorithmen zum Sortieren von *n* Zahlen.
- $O(n)$ lineares Wachstum.
- $O(2^n)$ exponentielles Wachstum. *f* wächst ungefähr auf das Doppelte, wenn sich *n* um 1 erhöht.

Achtung: Für den roten Algorithmus $rot(n)$ gilt auch $rot(n) = O(2^n)$, weil O eine obere Schranke ist.

Laufzeitkomplexität für imperative Konstrukte

```
for(int i = 0; i < n; i++) {          // O(n)
    a[i] = 0;                         // O(1)
}
```

Daraus folgt eine asymptotische Laufzeitkomplexität von $O(1) \cdot O(n) = O(n)$.

Eine Methode zur Feststellung der Laufzeitklasse für rekursive Algorithmen ist das Master-Theorem.

Sortieren

Strategie für den Bubble Sort

Annahme: Sortieren von Spielkarten

- Bubble Sort:
  - Aufnehmen aller Karten vom Tisch
  - Wenn nötig vertausche benachbarte Karten, bis die Reihenfolge korrekt ist

- The Simple Informatics
- Illustration als Tanz

Erstes Händische Beispiel

Bubble Sort mit folgenden Zahlen:

$$8 \quad 2 \quad 4 \quad 1 \quad 3$$

Jeder Durchgang soll einzeln aufgeschrieben werden und in rot sind die jeweils vertauschten Zahlen.

$$\begin{array}{ccccc} 2 & 8 & 4 & 1 & 3 \\ 2 & 4 & 8 & 1 & 3 \\ 2 & 4 & 1 & 8 & 3 \\ 2 & 4 & 1 & 3 & 8 \end{array}$$

Die größte Zahl ist jetzt bis zum Ende aufgestiegen; Sie ist nach hinten gebubbelt.

Aufgabe

Schreibe wieder jeden Schritt einzeln auf und markiere Vertauschungen rot.

$$3 \quad 9 \quad 0 \quad 8 \quad 5 \quad 7$$

Lösung

3	9	0	8	5	7
3	0	9	8	5	7
3	0	8	9	5	7
3	0	8	5	9	7
3	0	8	5	7	9
0	3	8	5	7	9
0	3	5	8	7	9
0	3	5	7	8	9
0	3	5	7	8	9

Javacode

```
static void sort(int[] array) {
  boolean sorted;
  do {
    sorted = true;
    for (int i = 1; i < array.length; i++) {
      if (array[i - 1] > array[i]) {
        int tmp = array[i - 1];
        array[i - 1] = array[i];
        array[i] = tmp;
        sorted = false;
      }
    }
  } while (!sorted);
}
```

Laufzeit

- Worst case, das Array ist umgekehrt sortiert. $n, n-1, \ldots, 2, 1$
  - $n-1$ Vertauschungen im ersten Durchlauf (n von Position 1 nach n)
  - $n-2$ Vertauschungen im ersten Durchlauf ($n-1$ von Position 1 nach $n-1$)
  - ...
  - Eine Vertauschung im n-ten Durchlauf, 2 von Position 1 nach 2.

  Daraus folgt: $\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \in O(n^2)$

- Best case, das Array ist sortiert, ein Durchlauf, deshalb $O(n)$
- Average case $O(n^2)$

Eigenschaften

- iterativ
- in-place $\rightarrow$ kein zusätzlicher Speicherbedarf notwendig
- einfach

Aufgabe

- Sortiere ein int Array deiner Wahl mit einem Bubble Sort mit 2 for-Schleifen.
- Nenne die Funktion `private static int[] bubbleSort(int[] numbers)`
- Ab welcher Größe vom Array kommt es zu spürbaren Wartezeiten?
- Wie könnte man das Problem lösen, dass große Elemente am Anfang schnell nach hinten getauscht werden, aber kleine Elemente am Ende nur langsam nach vorne?

Aufgabe

- Sortiere ein int Array deiner Wahl mit einem Bubble Sort mit 2 for-Schleifen.
- Nenne die Funktion `private static int[] bubbleSort(int[] numbers)`
- Ab welcher Größe vom Array kommt es zu spürbaren Wartezeiten?
- Wie könnte man das Problem lösen, dass große Elemente am Anfang schnell nach hinten getauscht werden, aber kleine Elemente am Ende nur langsam nach vorne?

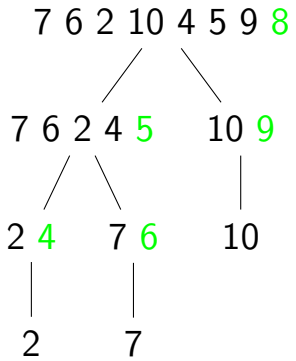
- Zum Beispiel Shaker-Sort; Array wird abwechselnd von oben und von unten durchlaufen.

Strategie für den Quick Sort

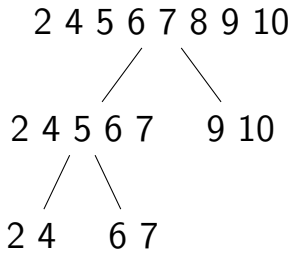
- Teile-und-Herrsche-Ansatz Engl.: divide and conquer.
  - aufwendiger Teile-Schritt, einfacher Merge-Schritt.
- Arrays mit einem oder keinem Element sind sortiert, sonst:
  - Aufteilung des Problems in zwei Teilmengen, wobei alle Elemente einer Teilmenge kleiner als alle Elemente der anderen Teilmenge sind.
  - rekursiver Aufruf des Algorithmus für beide Teilmengen.
- Verbindung der beiden Teilmengen, impliziert gegeben, da die Sortierung in-place geschieht.
- Aufteilung:
  - Wahl des Pivotelements p (Schlüssel).
  - Elemente $< p$, werden der linken Teilmenge zugeordnet.
  - Elemente $> p$, der rechten.

Quick Sort Baum Divide ↓

Grün ist das Pivotelement



Quick Sort Baum Conquer ↑



Quick Sort Methode

- Betrachte die Elemente $a[1], \ldots, a[N]$ als Folge S.
- Teile die Folge $S = a[1], \ldots, a[N]$ in $S1$ und $S2$, so dass:
  - für jeden Schlüssel k_{i_1} von S_1 und jeder Schlüssel k_{i_2} von S_2 gilt $k_{i_1} < k_{i_2}$
- Führe diesen Teilungsvorgang für $S1$ und $S2$ erneut durch.
- Die Methode endet bei einer 1-elementigen-Teilsequenz.
- S ist nach Beendigung des Verfahrens sortiert

In-Place Quick Sort Divide Step I

l durchläuft die Sequenz von links nach rechts, *r* vice versa

$$\underbrace{85}_{l} \quad 24 \quad 63 \quad 45 \quad 17 \quad 31 \quad \underbrace{96}_{r} \quad \underbrace{50}_{Pivot}$$

wenn *l* auf ein größeres Element als *r* zeigt $\rightarrow$ SWAP

$$\underbrace{\underbrace{85}_{l} \quad 24 \quad 63 \quad 45 \quad 17 \quad \underbrace{31}_{r}}_{swap} \quad 96 \quad \underbrace{50}_{Pivot}$$

$$31 \quad 24 \quad 63 \quad 45 \quad 17 \quad 85 \quad 96 \quad 50$$

$$\ldots$$

In-Place Quick Sort Divide Step II

$$31 \quad 24 \quad 17 \quad \underbrace{45}_{r} \quad \underbrace{63}_{l} \quad 85 \quad 96 \quad 50$$

Im letzten Schritt: SWAP mit dem Pivot Element

$$31 \quad 24 \quad 17 \quad \underbrace{45}_{r} \quad \underbrace{50}_{l} \quad 85 \quad 96 \quad 63$$

neues Pivotelement, entsprechend l,r; wiederhole bis 1-elementige Teilmengen.

Weitere Visualisierungen

- Video (englisch) Tree
- Video (deutsch) In-place
- visualgo

Implementierung in Java

rekursiver Aufruf

```java
public static void quickSort(int numbers[], int begin, int end) {
  if (begin < end) {
    int partitionIndex = partition(numbers, begin, end);

    quickSort(numbers, begin, partitionIndex - 1);
    quickSort(numbers, partitionIndex + 1, end);
  }
}
```

Implementierung in Java

Divide

```
private static int partition(int numbers[], int begin, int end) {
  int pivot = numbers[end]; int i = (begin - 1);

  for (int j = begin; j < end; j++) {
    if (numbers[j] <= pivot) {
      i++;
      int swapTemp = numbers[i];
      numbers[i] = numbers[j];
      numbers[j] = swapTemp;
    }
  }

  int swapTemp = numbers[i + 1];
  numbers[i + 1] = numbers[end];
  numbers[end] = swapTemp;

  return i + 1;
}
```

Laufzeit

- Wort case $O(n^2)$
- Best case $O(n\ log(n))$. Durch das Master Theorem $T(n) \in \Theta(n\ log(n))$
- Average case $O(n\ log(n))$

Eigenschaften

- rekursiv
- in-place
- optimale und mittlere Laufzeit von $O(n\ log(n))$; bester Wert bei vergleichsbasierten Suchalgorithmen.
- dafür ungünstige schlechteste Laufzeit von $O(n^2)$.
- nicht stabil (Laufzeiten unterscheiden sich)

Eigenschaften

- rekursiv
- in-place
- optimale und mittlere Laufzeit von $O(n\ log(n))$; bester Wert bei vergleichsbasierten Suchalgorithmen.
- dafür ungünstige schlechteste Laufzeit von $O(n^2)$.
- nicht stabil (Laufzeiten unterscheiden sich)

Wie kann man den Algorithmus verbessern?

Eigenschaften

- rekursiv
- in-place
- optimale und mittlere Laufzeit von $O(n\ log(n))$; bester Wert bei vergleichsbasierten Suchalgorithmen.
- dafür ungünstige schlechteste Laufzeit von $O(n^2)$.
- nicht stabil (Laufzeiten unterscheiden sich)

Wie kann man den Algorithmus verbessern?

- Vermeiden eines schlechten Pivotelements. Suche nach mittleren Element ist nicht zu empfehlen, weil nicht konstant. Darum:
  - mittleres Element vom ersten, mittleren Schlüssel und letztem Schlüssel.
  - randomisierte Wahl des Pivotelements. Hohe Wahrscheinlichkeit, dass schlechtes Pivotelement vermieden wird.

Laufzeiten Sortieralgorithmen auf Vergleichsbasis

Es gibt noch weitere Algorithmen

Algorithmus	Worst case	Best case	Average case
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n\ log(n))$	$O(n\ log(n))$	$O(n\ log(n))$
Merge Sort	$O(n\ log(n))$	$O(n\ log(n))$	$O(n\ log(n))$
Quick Sort	$O(n^2)$	$O(n\ log(n))$	$O(n\ log(n))$

Aufgabe

Suche dir einen Algorithmus aus der vorherigen Tabelle aus (ausgenommen Bubble Sort und Quick Sort), implementiere ihn in Java. Anschließend stelle ihn deinen Kollegen vor. Das soll auch grafisch geschehen.

Suchen

Das Suchproblem

- **Eingabe:** Folge von Elementen $< a_1, a_2, \ldots, a_n >$ und Suchelement.
- **Ausgabe:** Suche erfolgreich (ja/nein).
  - Erfolgreich: Index des gefundenen Elements a, dessen Wert mit dem Suchelement übereinstimmt. Eine Definition falls mehrere Elemente gefunden werden ist notwendig.
  - Nicht Erfolgreich: Entsprechende Ausgabe. Wird nach positiven Zahlen in Arrays gesucht ist das oft -1, als Rückgabewert.

Brute Force Methode

auch gennant lineare oder sequentielle Suche

Beginne beim ersten Element und durchsuche jede Stelle im Array sequentiell.

```java
public static int linearSearch(int[] data, int element) {
  // loop through array sequentially
  for (int i = 0; i < data.length; i++) {
    if (data[i] == element) {
      return i; // return index of the found element
    }
  }
  return -1; // element was not found
}
```

Brute Force Methode

auch gennant lineare oder sequentielle Suche

Beginne beim ersten Element und durchsuche jede Stelle im Array sequentiell.

```java
public static int linearSearch(int[] data, int element) {
  // loop through array sequentially
  for (int i = 0; i < data.length; i++) {
    if (data[i] == element) {
      return i; // return index of the found element
    }
  }
  return -1; // element was not found
}
```

Vorteil: einer der wenigen Algorithmen wo die Folge nicht sortiert sein muss.
Laufzeit:

Brute Force Methode

auch gennant lineare oder sequentielle Suche

Beginne beim ersten Element und durchsuche jede Stelle im Array sequentiell.

```java
public static int linearSearch(int[] data, int element) {
  // loop through array sequentially
  for (int i = 0; i < data.length; i++) {
    if (data[i] == element) {
      return i; // return index of the found element
    }
  }
  return -1; // element was not found
}
```

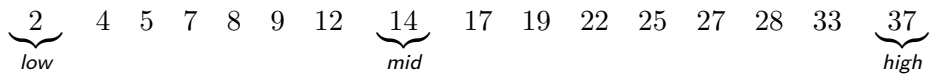
Vorteil: einer der wenigen Algorithmen wo die Folge nicht sortiert sein muss.
Laufzeit: $O(n)$.

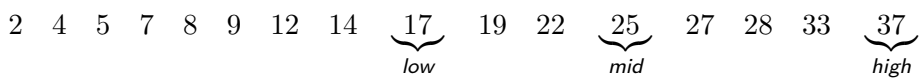
Strategie für die Binäre Suche

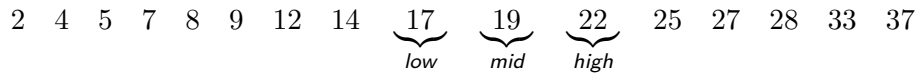
- Suche in einer sortieren Liste
- divide and conquer
- Suche nach dem Schlüssel k in einer Liste mit aufsteigend sortierten Schlüsseln:
  1. Wenn die Liste leer ist, wird die Suche erfolglos beendet.
     Andernfalls: Element $list[m]$ an mittlerer Position m prüfen.
  2. wenn $k = list[m].key$
     dann ist das gesuchte Element gefunden.
  3. wenn $k < list[m].key$,
     dann Suche in der linken Teilliste von Position 1 bis $m - 1$ nach demselben Verfahren.
  4. Andernfalls: $k > list[m].key$
     dann wird die rechte Teilliste ab Position $m + 1$ bis zum Ende der Liste nach demselben Verfahren durchsucht.

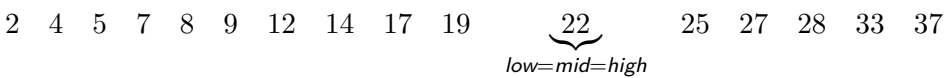
Beispiel

Suche 22









Binäre Suche rekursiv in Java

```
public static int binarySearch(int[] sortedArray, int key,
                               int low, int high) {
  int middle = low + ((high - low) / 2);

  if (high < low)
    return -1;

  if (key == sortedArray[middle])
    return middle;
  else if (key < sortedArray[middle])
    return binarySearch(sortedArray, key, low, middle - 1);
  else
    return binarySearch(sortedArray, key, middle + 1, high);
}
```

Laufzeit

- Wort case $O(log(n))$
- Best case $O(1)$
- Average case $O(log(n))$

Aufgabe

Definiere ein großes int-Array und Suche nach einer beliebigen Zahl.

- Suche linear nach dem Element.
- Verwenden einen von dir implementierten Sortieralgorithmus und suche anschließend binär.
- Ab welcher Größe vom Array merkst du einen zeitlichen Unterschied zwischen den beiden Algorithmen.

Aufgabe

Definiere ein großes int-Array und Suche nach einer beliebigen Zahl.

- Suche linear nach dem Element.
- Verwenden einen von dir implementierten Sortieralgorithmus und suche anschließend binär.
- Ab welcher Größe vom Array merkst du einen zeitlichen Unterschied zwischen den beiden Algorithmen.
- Suche eine weitere Art zu suchen und erkläre die Funktionsweise des Algorithmus. Er muss nicht implementiert werden.

Mustervergleich

Was ist ein String-Matching-Algorithmus

Finde Textsegmente in einer gegeben Zeichenkette (String).
Beispiel: strg+f Suche; Suche in PDFs oder anderen Dokumenten.

Naiver Algorithmus

Verschiebe eine Suchmaske über die Zeichenkette.

Eingabe: Strings $T = T_1 \dots T_n$ und $P = P_1 \dots P_m$

Ausgabe: $q =$ Stellen an denen P in T vorkommt.

T steht für Text und P für Pattern.

Visualisierung

- Video zur Visualisierung

Naive Pattern Matching in Java

Achtung: diese Implementierung bricht nach dem ersten Fund ab

```java
public static void search(String text, String pattern) {
  int n = text.length();
  int m = pattern.length();

  for (int s = 0; s <= n - m; s++) {
    for (int j = 0; j < m; j++) {
      if (text.charAt(s + j) != pattern.charAt(j)) break;
    }
    if (j == m)
      System.out.println("Pattern occurs at index " + s);
  }
}
```

Laufzeit

Keine Vorbereitung notwendig und die Suchzeit beträgt $\Theta((n - m + 1) \cdot m)$.

Alternative Algorithmen

- Deterministischer endlicher Automat
  - Automaten sind elementare Modelle in der Informatik. Sind finden fast überall Anwendung, vom Compilerbau bis hin zum Hardwareentwurf. Auch in der Netzwerktechnik finden sie Anwendung.
- Rabin-Karp Algorithmus
- Boyer-Moore Algorithmus
- Knuth-Morris-Pratt Algorithmus

Aufgabe

Implementiere eine Patternsuche in einem String. Es soll auch die Anzahl der gefundenen Elemente gezählt werden. Einen Lorem Ipsum Text findest du hier. Es kann hilfreich sein von mehrzeiligen Strings Gebrauch zu machen.

```
String text = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
in reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur. Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.
""";
```