Wie denke ich wie ein Programmierer?

Eines der wichtigsten Fähigkeiten, die man als Programmierer lernt, ist nicht eine bestimmte Sprache zu meistern, oder mit irgendwelchen Tools fließend umzugehen, sondern richtig wie ein Programmierer zu denken. Es ist leider auch eines der Fähigkeiten, die an der Uni kaum beigebracht wird - einerseits weil es für jede Person unterschiedlich ist, andererseits weil man dies allgemein durch (viel) Üben ergreift. Dennoch gibt es einige Tipps, die ich euch geben möchte, um das Lernen bzw. eure zukünftige Karriere als Programmierer zu erleichtern. Zusätzlich wird es detaillierte Beispiele zu jedem Konzept geben, damit man es besser verinnerlicht.

Bemerkung: Dieses Dokument ist für Anfänger gedacht und enthält (teilweise) vereinfachte Erklärungen und viel Text. Schaut weg, StackOverflow-Mods.

- Das Problem vereinfachen / unterteilen
  - Beispiel: Minima und Maxima eines 2D-Arrays
- Das Problem generalisieren
  - Beispiel: Dreiecke aus Sternchen
- Schrittweise Debuggen
  - Beispiel: Invertieren einer Zahl
- Häufige Muster
  - Schleifen
  - Arrays

Das Problem vereinfachen / unterteilen

Es kann überwältigend sein, sich als Anfänger mit einem neuen, umfangreichen Problem auseinanderzusetzen, wo man evtl. mehrere Kontrolfluss-Statements und Hilfsfunktionen zusammenpuzzeln muss. Wo fängt man an? Was für Variablen und Hilfsfunktionen brauche ich eigentlich? Wie baue ich die Funktion(en) dann auf?

Genauso wie in der reelen Welt ist es einfacher, wenn man ein großes Problem so stark wie möglich zu unterteilen, um das Problem systematisch "vom Grund auf" zu konstruieren. Dafür reichen die bereits erworbenen Kenntnisse vollkommend aus.

Wenn du jemandem erklären würdest, wie man ein Auto startet, reicht "starte das Auto" nicht aus. Man bräuchte eine Liste an Instruktionen, die man nacheinander ausführt, um den gewünschten Effekt zu erzielen, z.B. öffne die Tür, steige in den Fahrersitz ein, setze den Schlüssel ins Zündschloss, drehe den Schlüssel usw. Dasselbe gilt fürs Programmieren - je mehr man ein Problem unterteilt, desto klarer wird es.

Beispiel: Minima und Maxima eines 2D-Arrays

Nehmen wir als Beispiel die erste Teilaufgabe von W03P02.

- **Eingabe**: ein 2D-Integer-Array
- **Ausgabe**: ein 2D-Array, wobei jedes Array nur aus dem Minimum und Maximum der jeweiligen Arrays aus dem 2D-Eingabearray besteht

Kern dieser Aufgabe ist es also, die kleinsten bzw. größten Werte aller Arrays in einem neuen 2D-Array abzuspeichern. Wie starten wir? Lassen wir erstens das 2D-Array beiseite und betrachten ein einfaches 1D-Array, welches nur `int`s enthält. Dies wäre Schritt 1 der Vereinfachung: von 2D auf 1D.

- **temp. Eingabe**: ein 1D-Integer-Array
- **temp. Ausgabe**: ein Array aus 2 Elementen, wobei das erste das Minimum und das zweite das Maximum des Arrays ist

Jetzt stellt sich die Frage, wie wir die Minima bzw. Maxima in einem 1D-Array finden und wiedergeben. Die Antwort auf das Letztere ist einfach: wir geben ein neues Array zurück, wobei das erste Element das Minimum und das zweite Element das Maximum ist (`return new int[]{min, max}`). Wir wissen jetzt, dass wir also **zwei Variablen** benötigen, um diese zwei Werte abzuspeichern.

- **temp. Eingabe**: ein 1D-Integer-Array
- **temp. Variable 1**: Minimum
- **temp. Variable 2**: Maximum

Wie finden wir aber die Minima bzw. Maxima? Schauen wir uns als Beispiel ein kleines Array von nur 3 Elementen an: `{2,1,3}`. Das Einzige, was wir mit einem Array machen können, ist es **mit einer `for`-Schleife zu traversieren**, standardmäßig von links nach rechts. Wie kommen wir somit auf den kleinsten bzw. größten Wert?

Wir brauchen einen Stützpunkt. Ein guter Anfangspunkt hierzu wäre einfach der erste Wert des Arrays, `2`. Setzen wir also sowohl `min` als auch `max` darauf (`int min = a[0], max = a[0]`). Jetzt vergleichen wir in jedem Iterationsschritt, ob es einen kleineren bzw. größeren Wert als den bisher gespeicherten Wert gibt. Wenn ja, haben wir offensichtlich ein neues Minimum bzw. Maximum, so dass wir die Variable entsprechend anpassen müssen.

0. Setze `min` und `max` auf `a[0]` (2)
1. **Iterationsschritt**:
    1. Minimum: ist `a[1]` (1) kleiner als `min` (2)? Ja → setze `min` auf `a[1]` (1)
    2. Maximum: ist `a[1]` (1) größer als `max` (2)? Nein → ändere `max` nicht
2. **Iterationsschritt**:
    1. Minimum: ist `a[2]` (3) kleiner als `min` (1)? Nein → ändere `min` nicht
    2. Maximum: ist `a[2]` (3) größer als `max` (2)? Ja → setze `max` auf `a[2]` (3)

3. `min = 1, max = 3`

Folglich sieht unsere Funktion generalisiert so aus (mit einem trivialen Edge-Case-Check am Anfang):

```
public static int[] minAndMax(int[] a) {
    // edge case, return null on empty array to prevent exception
    if (a.length == 0) {
        return null;
    }

    // init. min and max as first element in array
    int min = a[0];
    int max = a[0];

    // iterate through array to find new min and max
    for (int i = 1; i < a.length; i++) {
        // if current el. of array is smaller than stored min, replace min
with current el.
        if (a[i] < min) {
            min = a[i];
        }
        // if current el. of array is greater than stored max, replace max
with current el.
        if (a[i] > max) {
            max = a[i];
        }
    }

    // return array of min and max
    return new int[]{min, max};
}
```

Jetzt wissen wir, wie wir den Minimum und Maximum eines Arrays finden und zurückgeben können. Wie wenden wir das aber auf unser 2D-Array an?

Nehmen wir uns wieder ein kleines Beispiel. Für das 2D-Array `{{2,1,3,4}, {5,8,6,7}}` soll das Ergebnis `{{1,4}, {5,8}}` sein. Wir müssen also für den Rückgabewert ein neues 2D-Array initialisieren, aber mit welchen Werten? Na ja, wir müssen für jedem Array die Minima und Maxima finden, also hat unser 2D-Rückgabearray genauso viele Arrays wie unser Eingabearray. Was ist aber mit der Länge des jeweiligen Arrays innerhalb des Rückgabearrays? Da unsere Hilfsmethode ein Array der Länge 2 zurückgibt, haben wir eine konstante Länge von 2 für jedes innere Array. Folglich initialisieren wir unser Rückgabearray: `int[][] output = new int[a.length][2]`.

Jetzt müssen wir unser Array **mit Werten auffüllen**. Wie zuvor erwähnt ist das einzige, was wir machen können, eine `for`-Schleife zu verwenden, um auf jedes einzelne Array unsere Hilfsfunktion anzuwenden.

```
public static int[][] minAndMax2D(int[][] a) {
    // initialize output array with proper sizes
    // currently filled with n = a.length arrays that only contain {0,0}
    int[][] out = new int[a.length][2];

    // replace arrays with result of help method applied to each array of 2D
input
    for (int i = 0; i < a.length; i++) {
        out[i] = minAndMax(a[i]); // {0,0} -> {min, max} for each inner
array
    }
    return out;
}
```

Somit haben wir unser Problem gelöst, indem wir es stückweise vereinfacht haben und zuletzt alles zusammengebaut haben.

Das Problem generalisieren

Der Alptraum jedes Anfänger ist es, eine Aufgabenstellung zu lesen, die keinen offensichtlichen Hinweis auf den Lösungsweg gibt. Wie soll man jetzt wissen, was für Datentypen, Variablen oder Statements dann überhaupt verwenden soll?

In der Tat kann man jede Problemstellung quasi in Programmier-Lingo "umwandeln", m.a.W. kann man jede Teilaufgabe als eine Reihe von `if`s, `else`s und Schleifen generalisieren.

So kann man eigentlich jedes Phänomen der Welt erklären. Wenn es gerade Tageszeit ist und keine Wolken auf dem Himmel sind, so sollte alles momentan beleuchtet sein. Wenn man das "programmiermäßig" ausdrücken würde...

```
while(isDaytime){
    if(!overcast){
        isIlluminated = true;
    } else {
        isIlluminated = false;
    }
}
```

Beispiel: Dreiecke aus Sternchen

Schauen wir uns als Beispiel die dritte Teilaufgabe von W02P02 an. Man soll auf der Konsole ein Dreieck aus '*'-Charakteren mit einer bestimmten Seitenlänge ausgeben. Betrachten wir als Beispiel die Ausgabe für `sideLength = 3`:

```
***
**
*
```

Wie können wir dieses Problem generalisieren? Was schon gleich auffällig sein sollte ist die Tatsache, dass es **Wiederholungen** gibt: einerseits wird auf jeder Zeile wiederholt das Sternchen geprintet, andererseits erkennt man ein Muster bezüglich jeder einzelnen Zeile; in der darauffolgenden Zeile verringert sich immer die Anzahl der Sternchen um 1, bis wir letztendlich keine Sternchen mehr auf der Konsole ausgeben.

Wenn wir Wiederholungen sehen, bedeutet dies, dass wir **Schleifen** benötigen, um denselben Code-Segment mehrmals laufen zu lassen.

Von außen nach innen: wir benötigen eine Schleifeniteration für jede Zeile. Da `sideLength` sowohl der Anzahl der Sternchen, als auch der Anzahl der Zeilen entspricht, können wir dies als Abbruchbedingung in einem einfachen `for`-Loop einbauen, damit wir anfangen können, etwas mit jeder einzelnen Zeile zu machen:

```
for (int i = 0; i < sideLength; i++) {
    // ...
}
```

OK, was jetzt? Jetzt müssen wir sehen, was in jeder Zeile passiert. Es werden offensichtlich Sternchen auf der Konsole geprintet, aber wie viele? Schauen wir uns das Beispiel genauer an. In der ersten Zeile werden 3 Sternchen ausgegeben, in der zweiten 2 und in der letzten 1. Wie können wir das generalisieren? Schauen wir uns den Zusammenhang genauer an.

1. `sideLength = 3`, `i = 0` → `numOfAsterisks = 3` (3 - 0 = 3)
2. `sideLength = 3`, `i = 1` → `numOfAsterisks = 2` (3 - 1 = 2)
3. `sideLength = 3`, `i = 2` → `numOfAsterisks = 1` (3 - 2 = 1)

Wir erkennen einen Zusammenhang zwischen der Seitenlänge, dem Zähler und der Anzahl der Sternchen, was wir in einer weiteren `for`-Schleife einbauen können:

```
for(int j = 0; j < sideLength - i; j++) {
    System.out.print("*");
}
```

Zuletzt müssen wir zu einer neuen Zeile nach jeder Iteration springen. Unser Code sieht also so aus:

```java
public static void printTriangle(int sideLength) {
    // iterate through each line...
    for (int i = 0; i < sideLength; i++) {
        // on current line, print corresponding number of asterisks
        for (int j = 0; j < sideLength - i; j++) {
            System.out.print("*");
        }
        // jump to new line for each new iteration
        System.out.println();
    }
}
```

Somit haben wir erfolgreich unser Problem als Verschachtelung von Schleifen generalisiert, obwohl in der Aufgabenstellung nichts darüber erwähnt wurde.

TIPP: In der Informatik gibt es nur selten einen einzigen richtigen Lösungsweg. Man könnte dies auch mit einem Zähler lösen, so wie in der Musterlösung. Testet und schaut, ob das richtige rauskommt!

Schrittweise Debuggen

Manchmal kommt man nicht selber auf eine Idee und lässt sich aus anderen Quellen inspirieren - seien es Blogs, Foren, Freunde, AI... Code aus externen Quellen zu nehmen ist unter einer sehr wichtigen Bedingung akzeptabel: Sinn der Sache ist es, dass man den Code **versteht**.

Was passiert jedoch, wenn man den Code einfach nicht versteht? Oder was, wenn man einfach sehen will, wie eine andere Person auf den Ansatz gekommen ist? Dazu dient das Swiss Army Knife der Programmierer - der Debugger.

Beispiel: Invertieren einer Zahl

Betrachten wir diesen Code-Abschnitt, um eine Zahl zu invertieren (z.B. aus 1234 wird 4321):

```java
public static int reverseNumber(int n) {
    int rev = 0;
    // what the fuck?
    while(n > 0) {
        rev = rev * 10;
        rev = rev + n % 10;
        n = n / 10;
    }
    return rev;
}
```

Warum funktioniert das? Das sieht man entweder mit Hilfe eines Debuggers, oder mit einem Stift, einem Blatt Papier und viel Geduld. Schauen wir uns das Programm schrittweise an, so wie es ein

Debugger machen würde, anhand des Beispiels `n = 123`.

- `rev`: Die Variable `rev` wird mit 0 initialisiert und wird die umgedrehte Zahl enthalten.
- Wir erstellen eine Schleife, um schrittweise die Zahl `n` zu überarbeiten.
  - **Idee**: Nehme immer die letzte Ziffer von `n` (von rechts) und setze sie als erste Ziffer von `rev` (von links)
  - **Wie?**: Rest der Ganzzahldivision von `n` mit 10 (`n % 10`)
  - **Abbruchbedingung**: `n == 0`
- Wir betrachten jetzt jeden Iterationsschritt:
  - **Schritt 1** (`n > 0 == true`):
    - `rev = rev * 10 = 0 * 10 = 0`
    - `rev = rev + n % 10 = 0 + (123 % 10) = 0 + 3 = 3`
    - `n = n / 10 = 123 / 10 = 12`
  - **Schritt 2** (`n > 0 == true`):
    - `rev = rev * 10 = 3 * 10 = 30`
    - `rev = rev + n % 10 = 30 + (12 % 10) = 30 + 2 = 32`
    - `n = n / 10 = 12 / 10 = 1`
  - **Schritt 3** (`n > 0 == true`):
    - `rev = rev * 10 = 32 * 10 = 320`
    - `rev = rev + n % 10 = 320 + (1 % 10) = 320 + 1 = 321`
    - `n = n / 10 = 1 / 10 = 0`
  - **Schritt 4** (`n > 0 == false`): Abbruch
- Gebe `rev` zurück.

Indem wir schrittweise den Algorithmus analysiert haben, konnten wir folgern, wie der Algorithmus aufgebaut wurde und warum dieser eigentlich funktioniert.

TIPP: Macht euch ggf. Notizen auf ein Blatt Papier und versucht, jeden Schritt zu visualisieren.

Häufige Muster

Zusammenfassend sind hier häufige Muster, die oft direkt in Code-Segmente übersetzt werden können.

Schleifen

- **Wiederholungen** bedeuten, dass Schleifen notwendig sind.
  - *Unbedingte* Wiederholungen fordern `while`-Schleifen.

    - Wenn ein *Zähler* notwendig ist (häufig für Arrays oder Potenzen, z.B. 2^i), ist eine `for`-Schleife notwendig.
    - *Beispiele*: gebe etwas wiederholt auf der Konsole aus, führe wiederholt Operationen mit einer Zahl durch...
- Wenn man jede **Komponente eines Objektes durchlaufen** muss, ist eine Schleife notwendig.
    - *Beispiele*: jedes Element eines Arrays, jede Ziffer einer Zahl, jeder Buchstabe eines Wortes, jeder Teiler einer Zahl...
- **Edge-Case**: Prüft immer, ob die Abbruchbedingung **immer** stimmt (z.B. dekrementieren einer negativen Zahl mit Abbruchbedingung `n > 0` führt zu einer Endlosschleife).
- **Edge-Case**: Prüft im Falle einer Schleife mit Zähler bei Arrays, ob der Index noch innerhalb der Grenzen des Arrays liegt (siehe Off-By-One-Fehler)

Arrays

- Arrays fordern knapp immer, dass man sie mittels **`for`-Schleifen** durchläuft.
- Die Größe von Arrays ist **fix**.
    - Wenn man die Größe eines Arrays ändern will, hilft dazu eine extra `resize`-Methode (siehe W03P01)
- "Prüfe, ob eine **Bedingung** für **alle** Elemente eines Arrays wahr ist" $\equiv$ Breche Schleifeniteration ab, wenn ein **Gegenbeispiel** gefunden wurde
    - *Beispiel*:
        - **Problem**: prüfe, ob alle Elemente eines Arrays gerade sind, und gebe einen `boolean` zurück
        - **Idee**: iteriere durch das Array so lange, bis eine ungerade Zahl (`a[i] % 2 != 0`) gefunden wurde $\rightarrow$ wenn ungerade Zahl gefunden wurde, dann `return false`, sonst **außerhalb** der Schleife `return true`
- **Edge-Case**: Prüft immer, ob ein Array leer ist, um ein Out-Of-Bounds-Zugriff zu vermeiden (m.a.W. `a[0]` auf einem leeren Array führt logischerweise zu einem Fehler)
