# Inhalt

# Kapitel 1

# Algorithmen und Datenstrukturen in ES6+

Dieses Buch ist noch in Bearbeitung....

## 1.1 Warum in ES6+

- ES is eating the world
    - meaningful understanding
    - I spent most of my professional life writing in JS and I think I know most about it

## 1.2 Behandelte Themen

- Algos und DS - Übungen am Ende. Versuche die Aufgaben zu machen ohne auf die Lösung zu schauen. Da ich finde, dass man von Fehlern oft viel mehr lernen kann als von richtigen Lösungen, habe ich vor der eigentlichen Lösung noch typische Fehler aufgelistet.

## 1.3 Nicht behandelte Themen

- Promises - async/await - webAPI - Browser spezifische Unterschiede

# Kapitel 2

# Entwicklungsumgebung

JS ist eine interpretierte Sprache. Sie läuft auf einer JS Engine. Die JS Engine läuft auf jeden Browser. Man kann direkt im Browser den Code ausführen.

Sobald man den Browser aber neu lädt ist unser Code weg. Besser ist es den Code entweder in nodejs auszuführen.

File speichern und mit node [filename] ausführen

# Kapitel 3

# JavaScript Paradigmen

JS ist eine sog. Multiparadigmen Programmiersprache. JS ist imperativ, objektorientiert und funktional.

## 3.1 Object-Orientiertes Programmieren

## 3.2 Funktionales Programmieren

# Kapitel 4

# JavaScript Konstrukte

## 4.1 Entscheidungskonstrukte

In JS gibt es if Statements, ternary und switch statement. if if else if else if ternary operator

Return

statement vs. expression switch

## 4.2 Wiederholungen

for loop, while loop, do while loop, symbol iterator; Generator; yields, for of

## 4.3 Funktionen

## 4.4 Scope

## 4.5 Closures

## 4.6 Call by Reference vs. Call by Value

## 4.7 Truthy und falsy

# Kapitel 5

# Datenstrukturen

## 5.1 Array

JS hat im Vergleich zu Java oder C/C++ nur sehr wenige Datenstrukturen. Eines ihrer wichtigsten Datenstrukturen ist der Array. Das Array werden wir später dazu benutzen, um alle anderen komplexeren Datenstrukturen zu implementieren. Der Unterschied zu JSs Arrays im Vergleich zu anderen Programmiersprachen ist, dass Arrays in JS keine fixe Länge haben. Durch das Hinzufügen und Entfernen von Elementen verändert sich die Array-Länge dynamisch mit. Bei der Initialisierung muss man dem Array dadurch auch keine bestimmte Länge mitgeben werden.

Arrays können in JS auf zwei Arten erstellt werden: Mit dem Array Konstruktor oder mit Array Literal:

```
1 var myArrayConstructor = new Array();
2 var myArrayLiteral = [];
```

Listing 5.1: Array Konstruktor

Der Array Konstruktur wird mit `new` eingeleitet und darauf folgt `Array()`. Beim Array Literal wird nur eine eckige Klammer `[]` benötigt. Beide Möglichkeiten erstellen einen Array. Jedoch wird angeraten zur Erstellung eines Arrays das Array Literal zu nehmen. Nicht nur ist er kürzer und auch schneller, er ist auch syntaktisch eindeutiger. Denn mit dem Array Konstruktur kann man auch die Länge des Arrays definieren als auch initialisieren. Die Syntax von beiden Konstrukten sind sich ähnlich, sodass es zu Verwirrung kommen kann, wenn man den Array mit Zahlen initialisiert:

```
1 var myArrayLength = new Array(3);
2 var myArrayInit = new Array(3,2,1);
```

Listing 5.2: Array Konstruktor

Die Länge eines Arrays wird als Zahl (hier: 3) in den Konstruktur reingeschrieben. Damit hat das Array in unserem Beispiel eine Länge von 3, die man mit `length`

überprüfen kann. Die (hier 3) Elemente sind noch `undefined`, da sie noch nicht initialisiert sind. Bei der Initialisierung gibt man die Elemente ebenfalls in den Konstruktur mit ein, jeweils getrennt durch einen Komma.

```
var myArrayLength = new Array(3);
console.log(myArrayLength);
// [undefined, undefined, undefined]
console.log(myArrayLenth.length);
// 3

var myArrayInit = new Array(3,2,1);
console.log(myArrayInit);
// [3, 2, 1]
console.log(myArrayInit.length);
// 3
```

Listing 5.3: Array Konstruktor

JS ist nicht static typed. D.h. ein Array kann Elemente nicht nur eines Typen gleichzeitig aufnehmen, sondern auch verschiedene. Ein Array in JS kann damit auch Zahlen, Strings, Bool und Objekte gleichzeitig aufnehmen:

```
var myArray = [1, "42", true, "hi", {"hello": "world"}];
```

Listing 5.4: Array Konstruktor

Intern werden die Elemente in einen String gecastet. Dadurch sind Arrays in JS langsamer als in anderen Sprachen. Um auf ein Array-Element zuzugreifen benutzen wir die eckige Klammer `[]`. Ein Array ist Index basiert und fängt mit `0` an. Um auf das zweite Element in einen Array zuzugreifen, schreiben wir also `myArray[1]`:

```
var myArray = [1, "42", true, "hi", {"hello": "world"}];
console.log(myArray[1]);
// "42"
```

Listing 5.5: Array Konstruktor

JS bieten viele Funktionen zur Manipulationen von Arrays an. Zum Hinzufügen am Ende wird `push()` benutzt. Um ein Element am Anfang des Arrays hinzuzufügen, wird `unshift()` verwendet:

```
var myArray = [1, 2, 3];
myArray.push(4);
console.log(myArray);
// [1, 2, 3, 4]

myArray.unshift(0);
console.log(myArray);
// [0, 1, 2, 3, 4]
```

Listing 5.6: Array Konstruktor

Für das Entfernen am Ende des Arrays gibt es `pop()`. Die Funktion `pop()` entfernt das letzte Element und gibt das entfernte Element zurück. Für das Entfernen am Anfang des Arrays verwendet man `shift()`.

```
1  var myArray = [0, 1, 2, 3, 4];
2  console.log(myArray.pop());
3  // 4
4  console.log(myArray);
5  // [0, 1, 2, 3]
6
7  console.log(myArray.shift());
8  // 0
9  console.log(myArray);
10 // [1, 2, 3]
```

Listing 5.7: Array Konstruktor

Um Elemente hinzuzufügen, zu ersetzen oder zu entfernen, die sich in der Mitte des Arrays befinden, verwendet man `splice()`. `splice()` nimmt als ersten Parameter den Index, an den das neue Element man hinzufügen will. Als zweiten Parameter wieviele Elemente danach ersetzt wird. Alle darauffolgenden Paramter die hinzuzufügenden Elemente.

```
1  var myArray = [1, 3, 4];
2  myArray.splice(1,0,2);
3  // adds a new element, 2, at index 1
4
5  myArray.splice(3,0,5,6,7);
6  // adds new elements, 5,6, and 7, at index 3
7
8  myArray.splice(5, 1);
9  // removes one element at index 5
10
11 myArray.splice(2, 3);
12 // removes 3 elements starting at index 2
13
14 myArray.splice(3, 1, 99);
15 // replaces one element at index 3 with the new element 99
16
17 myArray.splice(3, 2, 42);
18 // replaces 2 elements starting at index 3 with the new element 42
19
20 myArray.splice(999,0,2);
21 // if the index (first parameter) is larger than the array length,
       then it will just adds a new element to the end
```

Listing 5.8: Array Konstruktor

Die Elemente in einen Array kann man mit `sort()` sortieren. `sort()` nimmt eine Methode auf, die zwei Elemente miteinander vergleicht und entweder eine positive oder negative Zahl zurückgibt oder `0`. Eine negative Zahl steht dafür, dass die Zahl kleiner ist. Eine positive Zahl steht dafür, dass die Zahl größer ist. Eine `0` steht dafür, dass beide Zahlen gleich sind.

Beim Sortieren von Strings ist keine Funktion notwendig. Jedoch kann man eine mitgeben bei der überprüft wird, ob ein String größer ist als ein anderer String. Wenn man jedoch bei der Sortierung von Zahlen sich auf die Default `sort()` Funktion verlässt, dann können die Zahlen falsch sortiert werden:

```
1  var myArray = [1, 5, 1001, 8, 4];
2  myArray.sort((a,b) => a - b);
3  console.log(myArray);
4  // [1, 4, 5, 8, 1001]
5
6  var myArray = [1, 5, 1001, 8, 4];
7  myArray.sort();
8  console.log(myArray);
9  // [1, 1001, 4, 5, 8]
10
11 var myArray = ["a", "c", "xxx", "bd"];
12 myArray.sort();
13 console.log(myArray);
14 // ["a", "bd", "c", "xxx"]
15
16 var myArray = ["a", "c", "xxx", "bd"];
17 myArray.sort((a, b) => a > b ? 1 : -1);
18 console.log(myArray);
19 // ["a", "bd", "c", "xxx"]
```

Listing 5.9: Array Konstruktor

Man kann die `sort()` Methode auch benutzen, um ein Array absteigend zu sortieren. Dazu kehrt man das Vorzeichen der mitzugebenden Funktion um:

```
1  var myArray = [1, 5, 1001, 8, 4];
2  myArray.sort((a,b) => b - a);
3  console.log(myArray);
4  // [1001, 8, 5, 4, 1]
5
6  var myArray = ["a", "c", "xxx", "bd"];
7  myArray.sort((a, b) => (a > b ? -1 : 1));
8  console.log(myArray);
9  // ["xxx", "c", "bd", "a"]
```

Listing 5.10: Array Konstruktor

Alternativ kann man zur Umkehrung des Arrays auch die `reverse()` Funktion benutzen. Diese ist sogar etwas schneller als nur die Sort Funktion mit der Methode und dem umgekehrten Vorzeichen von oben zu benutzen.

```
1  var myArray = [1, 5, 1001, 8, 4];
2  myArray.sort((a,b) => a - b).reverse();
3  console.log(myArray);
4  // [1001, 8, 5, 4, 1]
5
6  var myArray = ["a", "c", "xxx", "bd"];
7  myArray.sort().reverse();
8  console.log(myArray);
```

```
9 // ["xxx", "c", "bd", "a"]
```

Listing 5.11: Array Konstruktor

Die Funktionen `push()`, `unshift()`, `pop()`, `shift()`, `splice()`, usw. verändern den originären Array. Es gibt auch Funktionen, die das Original nicht ändert, sondern stattdessen einen neuen Array zurückliefert.

Die `map()` Methode geht durch die Elemente durch und wendet dabei eine ihr mitgegebene Methode `(currentValue, index, array)=> {}` auf jedes einzelne Element im Array an. Die ihr mitgegebene Methode nimmt als erstes Argument das aktuelle Element auf. Das zweite Element ist das momentane Index im Array. Das dritte Argument das ursprüngliche Array. Das Ergebnis ist wieder ein Array mit derselben Länge, wie das ursprüngliche Array:

```
1 var myArray = [1, 2, 3, 4];
2 function multiplyBy2 = item => item * 2;
3 var doubleArray = myArray.map(multiplyBy2);
4 console.log(doubleArray);
5 // [2, 4, 6, 8]
```

Listing 5.12: Array Konstruktor

Die `filter()` Methode nimmt ebenfalls eine Funktion als Argument auf und wendet sie auf alle Elemente im Array an. Nur wenn die Auswertung dieser Funktion auf das aktuelle Element `truthy` zurückgibt, wird dieses Element auch Teil des später zurückgegebenen Arrays. Die ihr übergebene Methode `(currentValue , index, array)=> {}` hat als erstes Argument das aktuelle Element. Das zweite Argument ist der aktuelle Index. Das dritte Argument das original Array auf das die `filter()` Methode angewendet wird. Nur das erste Argument ist verpflichtend. Die restlichen sind optional. Beispielhaft ist unten die `filter()` Methode zum Filtern von nur geraden Zahlen gezeigt:

```
1 var myArray = [1, 2, 3, 4, 5, 6];
2 function isEven = item => item % 2 ;
3 var onlyEven = myArray.map(isEven);
4 console.log(onlyEven);
5 // [2, 4, 6]
```

Listing 5.13: Array Konstruktor

Die `reduce()` Methode unterscheidet sich von `map()` und `filter()` dadurch, dass nicht immer ein Array zurück gegeben werden muss. Stattdessen reduziert die Methode das Array auf einen einzigen Wert. Dieser Wert kann eine Zahl oder auch ein Array sein. Als erstes Argument kann sie eine Methode nehmen. Als zweites Argument nimmt sie einen initial Wert an. Das zweite Argument ist nur optional. Die Methode, die sie aufnimmt, `(acc, item, index, array)=> {}` hat als erstes Argument einen Akkumulator, der den kumulierten Wert enthält und der am Ende das Ergebnis darstellt. Das zweite Argument ist der aktuelle Wert. Das dritte Argument der Index und das vierte das original Array. Nur die ersten beiden Argumente sind verpflichtend.

```
var myArray = [1, 2, 3];
var sum = (acc, item) => acc + item;
var sumOfArray = myArray.reduce(sum, 0);
// 6
```

Listing 5.14: Array Konstruktor

Die `reduce()` Funktion geht im Array von links nach rechts. Mit der `reduceRight()` geht die Funktion von rechts nach links.

Die Methode `flat()`[1] wird auf Arrays angewendet, die selbst wiederum Arrays enthalten. Sie erstellt ein neues Arrays mit allen Unterarrays. Dabei kann optional bestimmt werden bis zu welcher Ebene die Unterarrays aufgelöst werden. Die `flat()` Methode nimmt optional nur einen numerischen Wert an. Dieser legt fest bis zu welcher Ebene die Unterarrays aufgelöst werden.

```
var myArray = [1, 2, 3, [4, 5, 6]];
console.log(myArray.flat());
// [1, 2, 3, 4, 5, 6]

var myArray = [1, 2, 3, [4, [5, 6]]];
console.log(myArray.flat());
// [1, 2, 3, 4, [5, 6]]

var myArray = [1, 2, 3, [4, [5, 6]]];
console.log(myArray.flat(2));
// [1, 2, 3, 4, 5, 6]
```

Listing 5.15: Array Konstruktor

Die `flat()` Methode wird auch genutzt, um leere Elemente in Arrays zu entfernen

```
var myArray = [1, 2, 3, , ,6];
console.log(myArray.flat());
// [1, 2, 3, 6]
```

Listing 5.16: Array Konstruktor

Die Methode `flatMap()` [2] ist identisch zum Aufruf einer `map()` Methode gefolgt vom Aufruf einer `flat()` Methode. Die Methode wendet eine ihr mitgegebene Funktion auf alle Elemente an und flacht sie anschießend ab. Die ihr mitgegebene Funktion `(item, index, array)=> {}` nimmt als erstes Argument das aktuelle Element. Das zweite Argument der Index und das dritte Argument das Original Array.

```
var myArray = [1, 2, 3];
var duplicate = item => [item, item];
console.log(myArray.flatMap(duplicate));
// [1, 1, 2, 2, 3, 3]
```

Listing 5.17: Array Konstruktor

[1] noch im Experiment Status, d.h. es wurde noch nicht von allen JS Engines implementiert
[2] noch im Experimentier Status, d.h. nicht alle Browser haben es implementiert

Die Funktion `concat()` vereint zwei Arrays und liefert das vereinigte Array als neues Array wieder zurück:

```
var myArray1 = ["a", "b", "c", "x"];
var myArray2 = ["d", "e", "f"];
var newArray = myArray1.concat(myArray2);
console.log(newArray);
// ["a", "b", "c", "x", "d", "e", "f"]
```

Listing 5.18: Array Konstruktor

Eine andere Möglichkeit ist es den Rest/Spread Operator zu verwenden:

```
var myArray1 = ["a", "b", "c", "x"];
var myArray2 = ["d", "e", "f"];
var newArray = [...myArray1, ...myArray2];
console.log(newArray);
// ["a", "b", "c", "x", "d", "e", "f"]
```

Listing 5.19: Array Konstruktor

Die `slice()` Methode erstellt eine (flache) Kopie des Arrays und gibt diese Kopie als neues Array zurück. Die Methode nimmt zwei Argumente auf. Das erste Argument beschreibt ab welchem Index die Kopie erstellt wird. Das zweite optionale Argument beschreibt bis zu welchem (exklusve) Index das Array kopiert wird. Wird für das zweite Argument kein Wert gegeben, dann kopiert er bis zum Arrayende:

```
var myArray = ["a", "b", "c", "x"];
var newArray = myArray.slice(0);
console.log(newArray);
// ["a", "b", "c", "x"]
```

Listing 5.20: Array Konstruktor

Eine flache Kopie deshalb, weil die Funktion die Elemente als Referenz in das neue Array schreibt. D.h. jede Änderung im original Array hat Auswirkung auf das neue Array. Dies gilt jedoch nicht für `Number`, `String`, `boolean`, `null`, `undefined`, `symbol`, sondern für `object`, `Array`, `function`:

```
// Using slice, create newCar from myCar.
var myHonda = { color: 'red', wheels: 4, engine: { cylinders: 4,
    size: 2.2 } };
var myCar = [myHonda, 2, 'cherry condition', 'purchased 1997'];
var newCar = myCar.slice(0, 2);
console.log(newCar);
// Display the values of myCar, newCar, and the color of myHonda
//  referenced from both arrays.
console.log('myCar[0].color = ' + myCar[0].color);
console.log('newCar[0].color = ' + newCar[0].color);

// Change the color of myHonda.
myHonda.color = 'purple';
console.log('The new color of my Honda is ' + myHonda.color);

```

```
15 // Display the color of myHonda referenced from both arrays.
16 console.log('myCar[0].color = ' + myCar[0].color);
17 console.log('newCar[0].color = ' + newCar[0].color);
```

Listing 5.21: Array Konstruktor

Eine weitere Möglichkeit den Array zu kopieren ist mit Hilfe des Rest/Spread-Operators:

```
1 var myArray = ["a", "b", "c", "x"];
2 var newArray = [...myArray];
3 console.log(newArray);
4 // ["a", "b", "c", "x"]
```

Listing 5.22: Array Konstruktor

Der Unterschied zwischen beiden Möglichkeiten ist, dass `slice()` eine flache Kopie und der Rest/Spread Operator eine tiefen Kopie erstellt:

```
1 var myArray = ["a", "b", "c", "x"];
2 var newArrayShallow = myArray.slice(0);
3 var newArrayDeep = [...myArray];
4 console.log(myArray === newArrayShallow);
5 // true --> it's pointing to the same memory space
6
7 console.log(myArray === newArrayDeep);
8 // false --> it's pointing to a new memory space
```

Listing 5.23: Array Konstruktor

Es muss aber hier erwähnt werden, dass der Rest/Spread Operator nur eine Ebene tief kopiert. Bei mehrdimensionalen Arrays muss man andere Methoden wählen. Unten sind Methoden aufgelistet mit der man eine tiefen Kopie, also einen Klon, erstellen kann:

```
1 var myArray = ["a", "b", "c", "x"];
2
3 var clonedArray1 = JSON.parse(JSON.stringify(myArray))
4 var clonedArray2 = [].concat(myArray);
5 var clonedArray2 = Array.from(myArray);
```

Listing 5.24: Array Konstruktor

Wie im vorherigen Kapitel angesprochen gibt es die Möglichkeit durch ein Array mit einer Schleife zu iterieren. Die Arrays in JS bieten eigene Funktionen zum Iterieren an. Die `forEach()` Funktion nimmt eine Methode und geht durch alle Elemente durch und wendet auf jedes einzelene ELement die Methode an. Die Funktion kann, im Gegensatz zum `for`, `while`, `do while`, usw. Schleifen nicht unterbrochen werden - außer durch das Werfen einer Ausnahme (was aber nicht empfohlen wird). Die `forEach()` Funktion liefert kein Ergebnis zurück.

Listing 5.25: Array Konstruktor

Will man vorher abbrechen so kann man entweder die `every()` oder `some()` verwenden. Die `every()` nimmt eine Funktion auf und überprüft sie für alle Elemente. Sobald bei der Überprüfung bei eines der Elemente `falsy` zurück gegeben wird, bricht sie ab. Bei der `some()` wird die Iteration abgebrochen sobald bei der Auswertung `truthy` zurückgegeben wird.

Listing 5.26: Array Konstruktor

Oft will man ein Element im Array finden. Dazu kann man wieder Schleifen benutzen oder die Built-in Funktionen verwenden.

Will man nur feststellen, ob ein Element auch im Array vorhanden ist, so kann man `includes()` verwenden. Dieses gibt entweder `true` oder `false` zurück, wenn das Element im Array gefunden bzw. nicht gefunden wurde.

Listing 5.27: Array Konstruktor

Eine Möglichkeit herauszufinden, an welcher Position das gesuchte Element sich befindet, bietet `indexOf()`. Diese nimmmt eine Funktion auf und liefert den ersten Index des Elements, bei der die mitgegebene Funktion `truthy` zurückliefert. `lastIndexOf()` macht genau das Gegenteil: sie gibt den Index des zu letzt gefundenen Elements zurück. Falls keines der Elemente den Bedingungen entspricht, liefert `indexOf()` und `lastIndexOf()` `-1` zurück.

```
1 asdf
```

Listing 5.28: Array Konstruktor

Will man einen `String`in einen Array umwandeln, so kann man wieder den Rest/Spread-Operator verwenden. Dabei wird jedes einzelne Zeichen, inklusive Leerzeichen, Komma, Sonderzeichen usw., als eigenes Element in ein neues Array gepackt und zurückgegeben.

```
1 var myString = "hello, world";
2 var myArray = [...myString];
3 console.log(myArray);
4 // ["h", "e", "l", "l", "o", ",", " ", "w", "o", "r", "l", "d"]
```

Listing 5.29: Array Konstruktor

Mit `split()` kann man festlegen ab wann man die Elemente in einen Array übergeben will. Die Funktion nimmt einen `String` als Prädikat auf. Will man z.B. beim obigen Beispiel nur die Wörter in den Array geben, die durch einen Komma getrennt sind, so gibt man das als String in die Funktion ein:

```
1 var myString = "hello, world";
2 var myArray = myString.split(",")
3 console.log(myArray);
4 // ["hello", " world"]
```

Listing 5.30: Array Konstruktor

## 5.2 Liste

Lists are convenient if the order doesn't matter or if you don't have to search for a certain item implement a fasfwe listlistSize (property) Number of elements in list pos (property) Current position in list length (property) Returns the number of elements in list clear (function) Clears all elements from list toString (function) Returns string representation of list getElement (function) Returns element at current position insert (function) Inserts new element after existing element append (function) Adds new element to end of list remove (function) Removes element from list front (function) Sets current position to first element of list end (function) Sets current position to last element of list prev (function) Moves current position back one element next (function) Moves current position forward one element currPos (function) Returns the current position in list moveTo (function) Moves the current position to specified position

While writing also test your impelmentation with console.assert()

## 5.3 Stack

## 5.4 Queue

## 5.5 Dequeue

## 5.6 Priority Queue

## 5.7 LinkedList

## 5.8 Zirkulare LinkedList

## 5.9 Zweifach verknüpfte LinkedList

## 5.10 Dictionary

## 5.11 Hashing

## 5.12 HashMap

## 5.13 MapTree

## 5.14 LinkedMap

## 5.15 Sets

## 5.16 Binäre Bäume

## 5.17 Graphen

## 5.18 AVL Tree

# Kapitel 6

# Algorithmen

Sortier Algorithmen und Such Algorithmen

## 6.1 Breitensuche

## 6.2 Tiefensuche

## 6.3 Bubble Sort

## 6.4 Selection Sort

## 6.5 Shellsort

## 6.6 Mergesort

## 6.7 Quicksort

## 6.8 Sequential Suche

## 6.9 Binäres Suchen

## 6.10 Suchen nach Minimum und Maximum

## 6.11 Rucksackproblem

## 6.12 Greedy Algorithm

# Kapitel 7

# Aufgaben

## 7.1 Binäres Suchen

**Aufgabe 1.** *Gegeben ist ein Array mit ganzen Zahlen. Gib den Index des gegebenen Keys. Falls kein Ergebnis gefunden wurde, gib -1 zurück.*

*Beispiel: Gegeben ist folgender Array, wenn der Key 47 ist, dann soll die Binäre Suche 2 zurückgeben.*

| *Index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *Key* | *23* | *31* | *47* | *65* | *69* | *73* | *75* | *89* | *91* | *93* |

**Stichwörter:** Array, Binary Search, Suche, Sliding Window, Zeiger, Pointer, Divide and Conquer, Teile und Herrsche

#### Vorgehensweise

1. Betrachte das Array von Anfang bis Ende
2. Berechne den Index der in der Mitte liegt
3. Wenn der Mitte Index genau auf den Key zeigt, dann gib den diesen Index zurück
4. Wenn das Element an der Mitte Index kleiner ist als der Key, dann betrachte nur das Subarray von der Mitte Index bis zum Ende
5. Wenn das Element an der Mitte Index größer ist als der Key, dann betrachte nur das Subarray von der Mitte Index bis zum Anfang
6. Wiederhole die obigen Schritte solange bis das Subarray leer ist

#### Typische Fehler

```
1  //a is sorted array
2  let binarySearch = function(a, key) {
3    let newA = a;
4    let currentIndex;
5    while(newA.length !== 0) {
6      currentIndex = newA.length / 2;
7      const currentKey = newA[currentIndex];
8
9      if (currentKey === key) {
10       return currentIndex;
11     } else if (currentKey < key) {
12       newA = newA.slice(currentIndex);
13     } else {
14       newA = newA.slice(0, currentIndex + 1);
15     }
16   }
17   return -1;
18 };
```

Listing 7.1: My Javascript Example

- Ein Array ist index basiert beginned bei 0

  Das weißt wahrscheinlich jeder bisher. Jedoch wird die Implikationen dessen sehr oft vergessen. Um auf den Index `currentIndex` zu kommen muss die `(arr.length - 1)/ 2` genommen werden.

- Array Index sind immer ganze Zahlen

  Auch dies sollte mittlerweile bekannt sein. Doch wie im obigen Fall zu sehen wurde durch `2` geteilt. Bei ungeraden Zahlen entstehen dadurch rationale Zahlen, die auf ganze Zahlen wieder zurück gecastet werden müssen

- Es wird nach dem Index des original Arrays gefragt

  Ein neues Array zu erstellen und darauf den Index zu ermitteln bedeutet, dass man den Index des neuen Arrays zurück gibt. Gesucht ist aber der Index des gegebenen Arrays

- `currentIndex` nicht wiederholt überprüfen

  Selbst wenn es möglich sein sollte das original Array in Stücke so zu teilen dass dennoch die original Index beizubehalten, ist die Auswahl der Indexe falsch, denn `currentIndex` wurde bereits geprüft. Es ist daher unsinnig, dass dieser Index noch im Subarray erscheint. Man müsste also `currentIndex + 1` für `currentKey < key` bzw. `curentIndex` (`currentIndex` selbst wird also nicht betrachtet) für `currentKey > key` wählen.

- Programm liefert kein Ergebnis

  Das Programm läuft in Endlosschleife und kann somit kein Ergebnis liefern.

**Lösung** Wir versuchen immer am Anfang einen "early exit" zu erreichen, damit der Code effizient bleibt. Alternativ hätte man auch `includes()` verwenden können. Aber `includes()` geht durch alle Array Elemente durch und hätte damit eine Laufzeit von `O(n)`.

Wir erreichen einen "early exit", indem wir überprüfen, ob das Array keine Elemente enthält oder ein Element. Bei keinem Element geben wir sofort `-1` zurück. Wenn ein Element vorhanden ist können wir gleich überprüfen, ob dieses Element dem `key` entspricht. Falls ja, geben wir `0` zurück, falls nicht, ist es nicht vorhanden und wir geben `-1` zurück.

Die Vorgehensweise ist ähnlich wie oben beschrieben. Wir werden aber zwei Zeiger benutzen. Dadurch bleiben die Indexe immer die der originalen Arrays.

Die Laufzeit

Zeit komplexität

```
1  //a is sorted array
2  let binarySearch = function(a, key) {
3    if (a.length === 0) {
4      return -1;
5    }
6    if (a.length === 1) {
7      return a[0] === key ? 0 : -1;
8    }
9    let startIndex = 0;
10   let endIndex = a.length - 1;
11   while(startIndex <= endIndex) {
12     const currentIndex = (startIndex + endIndex) / 2 | 0;
13     const currentKey = a[currentIndex];
14     if (currentKey === key) {
15       return currentIndex;
16     } else if (currentKey < key) {
17       startIndex = currentIndex + 1;
18     } else {
19       endIndex = currentIndex - 1;
20     }
21   }
22   return -1;
23 };
```

Listing 7.2: My Javascript Example

## 7.2 Maximum im gleitenden Fenster

> **Aufgabe 2.** *Gegeben ist ein langes Array mit ganzen Zahlen. Zudem ist noch ein Fenster mit einer Breite `w` gegeben, das sich von Anfang bis Ende des Arrays bewegt. Finde alle Maxima, die im gleitenden Fenster auftauchen.*
>
> *Beispiel: Gegeben ist folgender Array mit einer Fensterbreite von 4.*
>
> | 3 | 23 | -31 | 47 | -65 | 69 |
> |---|---|---|---|---|---|
>
> *Das Fenster bewegt sich nun von links nach rechts und enthält folgende Zahlen:*

> 1. *Schritt*
>
>    | *3* | *23* | *-31* | *47* |
>    |---|---|---|---|
>
>    *Das Maximum ist 47*
>
> 2. *Schritt*
>
>    | *23* | *-31* | *47* | *-65* |
>    |---|---|---|---|
>
>    *Das Maximum ist wieder 47*
>
> 3. *Schritt*
>
>    | *-31* | *47* | *-65* | *69* |
>    |---|---|---|---|
>
>    *Das Maximum ist wieder 69*
>
> *Das erwartete Ergebnis ist in diesem Beispiel*

| *47* | *47* | *69* |
|---|---|---|

**Stichwörter:** Array, Suche, Sliding Window, Zeiger, Pointer, Dequeue

### Vorgehensweise

1. Wir erstellen zwei Zeiger

   Die Zeiger nennen wir startWindow und endWindow. startWindow wird mit `0` und `endWindow` wird mit `array.length - 1` initialisiert.

2. Hilfsfunktion zur Berechnung des Maximums

   Wir erstellen noch eine Hilfsfunktion, die das Maximum für gegebenes `startWindow` und `endWindow` berechnet und in ein Resultat-Array hinzufügt.

3. Iteration bis Arrayende

   Wir iterieren dann duch die verbleibenden Elemente und inkrementieren `startWindow` und `endWindow`. Dabei rufen wir jedesmal die Hilfsfunktion zur Berechnung des Maximums.

4. Rückgabe

   Am Ende geben wir das Resultat-Array zurück

```
1  const findMaxSlidingSindow = function(arr, window_size) {
2    const getMax = (startWindow, endWindow) => {
3      const max = arr
4      .slice(startWindow, endWindow + 1)
5      .reduce((max, item) => item > max ? item : max, -Infinity);
6      result.push(max);
7    }
8    const result = [];
9    let startWindow = 0;
10   let endWindow = window_size - 1;
11   let max =
12   while (endWindow < arr.length) {
13
```

```
14      getMax(startWindow++, endWindow++);
15    }
16    return result;
17  };
```

Listing 7.3: My Javascript Example

### Typische Fehler

- Kein early exit verwendet

  Man sollte immer versuchen einen early exit zu verwenden, d.h. schauen, ob anhand einer einfachen überprüfung am Anfang gleich auf das richtige Ergebnis geschlossen werden kann. Beispw. muss man nichts berechnen, wenn das Array leer ist oder die gegebene Fensterbreite größer ist als das Array.

- Keine geeignete Datenstruktur verwendet

  Die Aufgabe bietet es sich an ein `Dequeue` zu verwenden. Dadurch brauchen wir auch keine zwei Zeiger.

- Ungünstige Laufzeit

  Durch die Hilfsfunktion zur Berechnung des Maximums werden bereits besuchte Elemente erneut besucht. Bei einer kleinen Fensterbreite ist das kein großes Problem aber angenommen es handelt sich um ein langes Array mit einer sehr großen Fensterbreite. Dies führt zu einer langsamen Laufzeit. Das ist ein erster Hinweis, dass der vorliegende Code nicht optimal ist.

**Lösung** Die Schwierigkeit liegt darin, das Maximum zu finden ohne erneut die Zahlen im Fenster durchzugehen. Idealerweise wollen wir nur eimal durch die Zahlen im Array gehen und dabei das Maximum speichern. Nachfolgend werde ich begründen, warum eine `Dequeue` die geeignete Datenstruktur ist.

Angenommen wir haben folgendes Array und eine Fensterbreite von `3` und benutzen dazu wir zunächst eine Variable zur Speicherung des Maximums:

| 7 | 6 | 2 | 4 | 1 | 7 |
|---|---|---|---|---|---|

Wenn wir durch die einzelnen Arrays gehen und das Maximum jedesmal in der Variable speichern, würden wir folgendes Ergebnis bekommen:

1. Schritt

   | 7 |
   |---|

   Erste Zahl ist 7. Dadurch ist max auch 7

2. Schritt

   | 7 | 6 |
   |---|---|

   Die 6 ist kleiner als das aktuelle max und daher ist max immer noch 7

3. Schritt

| 7 | 6 | 2 |
|---|---|---|

Max bleibt nach der 2 immer noch bei 7. Dadurch haben wir die Fensterbreite erreicht. Das max ist für die erste Fensterbreite ist 7.

4. Schritt

| 6 | 2 | 4 |
|---|---|---|

Ab jetzt bekommen wir eine konstante Anzahl an 3 Zahlen. Die 7 ist raus. Die 4 kommt rein. Das Maximum kann nun nicht mehr 7 sein. Aber was ist es stattdessen? Es kann nicht 4 sein. Es ist 6. Aber unser Ansatz hat keine Möglichkeit zu erkennen, dass es die 6 ist.

Was wir daher brauchen ist also keine Variable zur Speicherung des Maximums, sondern wir müssen eine Sequenz speichern. Wie wollen die Sequenz so speichern, dass die größere Zahl immer vorne steht. Die Zahlen im `Dequeue` werden von links nach rechts immer kleiner - die Zahlen links sind also immer größer als die Zahlen rechts. Wenn wir also das Maximum haben wollen, dann greifen wir immer auf die erste Zahl im `Dequeue`. Falls die Zahl außerhalb des Arrays rausgeht, dann löschen wir diese Zahl von der Spitze. Damit wir leichter wissen können, welche Zahl aus dem Fenster rausgeht, speichern wir daher nicht den Wert der Zahl, sondern den Index der Zahl.

Wir benutzen nachfolgend dieselbe Zahlenfolge. Diesmal habe ich noch die Indexe hinzugefügt.

| Index: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value: | 7 | 6 | 2 | 4 | 1 | 7 |

Wenn wir obiges Beispiel nehmen, dann würde der Ablauf wie folg aussehen:

1. Schritt

| Index: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value: | 7 | 6 | 2 | 4 | 1 | 7 |

Erste Zahl ist 7. Weil die `Dequeue` noch leer ist, pushen wir das erste Index sie in unsere `Dequeue` rein | 0 |

2. Schritt

| Index: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value: | 7 | 6 | 2 | 4 | 1 | 7 |

Die 6 ist kleiner (man erinnert sich: Wir wollen in der `Dequeue` die Zahlen der kleiner nach speichern) als das aktuelle max und wir pushen sie ins `Dequeue`: | 0 | 1 |

3. Schritt

| Index: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value: | 7 | 6 | 2 | 4 | 1 | 7 |

Die 2 ist kleiner als die 6. Dadurch pushen wir auch sie in unsere `Dequeue` | 0 | 1 | 2 |.

Damit haben wir die Fensterbreite erreicht. Jetzt müssen wir aufpassen, wann eine Zahl aus dem Fenster rausrutscht. Falls diese Zahl nicht mehr im Fenster ist, müssen wir auch dessen Index rauslöschen.

4. Schritt

| Index: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value: | 7 | 6 | 2 | 4 | 1 | 7 |

Die 4 ist in das Fenster reingekommen. Die 4 ist aber größer als die 2. Daher löschen wir sie (bzw. dessen Index). Wir überprüfen, ob vor der Zahl 2 noch eine Zahl ist, die kleiner ist als die 4. Dies ist nicht der Fall. Wäre es aber der Fall, so müssten wir auch diesen Index löschen. Wir müssten nämlich so lange die Indexe löschen, bis der `Dequeue` die Struktur der Gestalt hat, dass sie absteigend ist, also vorne die großen Zahlen, hinten die kleinen. Daher löschen wir nur die Index der Zahl 2 aus der `Dequeue` und fügen den Index der Zahl 4 hinzu:

| 0 | 1 | 3 |
|---|---|---|

.

Es passiert an dieser Stelle aber noch mehr: Die 7 (Index Stelle 0) ist aus dem Fenster gerutscht. Das ist deshalb bemerkenswert, weil die 7 bisher das Maximum war und damit an erster Stelle unserer `Dequeue` steht. Weil die Zahl 7 jetzt raus ist, löschen wir auch die den Index 0 aus unserem `Dequeu` | 1 | 3 |. Dadurch steht der Index 1 an erster Stelle unserer `Dequeue`. D.h. die 6 ist das Maximum an dieser Stelle.

Ich könnte das jetzt weiter fortführen, aber ich glaube das Prinzip sollte mittlerweile klar sein.

Unter der Berücksichtigung der "typischen Fehler" und den Ansatz mit der `Dequeue`, schaut die Implementierung wie folgt aus:

```
1  const findMaxSlidingWindow = function(arr, windowSize) {
2    // early exit
3    if(arr.length == 0 || windowSize > arr.length) {
4      return;
5    }
6    let result = [];
7    let dequeue = [];
8    const dequeueIsNotEmpty = () => dequeue.length > 0;
9    const currentElementLargerThanLastDequeueElement = currentEl =>
         currentEl >= arr[dequeue[dequeue.length - 1]]
10
11   // setup for first step
12   for (let i = 0; i < windowSize; i++) {
13     // remove all elements that are smaller than the current
         element
14     while (dequeueIsNotEmpty()
15     && currentElementLargerThanLastDequeueElement(arr[i])) {
16       dequeue.pop();
17     }
18
19     dequeue.push(i);
20   }
21
```

```
22    result.push(arr[dequeue[0]])
23
24    // remaining steps
25    for (let i = windowSize; i < arr.length; i++) {
26      // remove all elements that are smaller than the current
            element
27      while (dequeueIsNotEmpty()
28      && currentElementLargerThanLastDequeueElement(arr[i])) {
29        dequeue.pop();
30      }
31
32      // if number falls out from the window, we have to delete it
            from the dequeue
33      if (dequeueIsNotEmpty() && (dequeue[0] < i - (windowSize - 1)))
             {
34        dequeue.shift();
35      }
36
37      dequeue.push(i);
38      result.push(arr[dequeue[0]]);
39    }
40
41    return result;
42  };
```

Listing 7.4: My Javascript Example

Eine alternative Lösung ist es, einen `Heap()` zu verwenden.

```
1  // test...
```

Listing 7.5: My Javascript Example