

Sarah Ertel  
Patrick Greher  
Eugen Ljavin

1	2	3	$\Sigma$

## Übungsblatt Nr. 2

(Abgabetermin 03.05.2018)

### Aufgabe 1

a)

$A = [0, 2, 4, 6, 8, 10]$

$x = 4$

b)

$A = [0, 2, 4, 6, 8, 10]$

$x = 2$

c)

---

#### Algorithm 1: Ternary Search

---

```

1 function ternarySearch(array[ ], key, lowerBound, upperBound)
2   if upperBound  $\geq$  lowerBound then
3     indexLeft  $\leftarrow$  (upperBound - lowerBound) / 3 + lowerBound;
4     indexRight  $\leftarrow$  (upperBound - lowerBound) / 3 + indexLeft;
5     if array[indexLeft] = key then
6       return indexLeft;
7     end
8     if array[indexRight] = key then
9       return indexRight;
10    end
11    if array[indexLeft] > key then
12      return ternarySearch(array, key, lowerBound, indexLeft - 1);
13    end
14    if array[indexRight] < key then
15      return ternarySearch(array, key, indexRight + 1, upperBound);
16    end
17    return ternarySearch(array, key, indexLeft + 1, indexRight - 1);
18  end
19 return -1;
```

---

1. Als Eingabeparameter werden das zu durchsuchende Array ( $array[ ]$ ), das gesuchte Element ( $key$ ) sowie die untere ( $lowerBound$ ) und obere ( $upperBound$ ) Grenze verlangt.  $lowerBound$  sowie  $upperBound$  werden als Eingabeparameter benötigt, um einen rekursiven Aufruf zu ermöglichen. Beim erstmaligen (nicht rekursiven) Aufruf, sollte für  $lowerBound$  der Wert 0 und für  $upperBound$  der Wert  $array.length - 1$  übergeben werden.

Der Algorithmus liefert den Index des zu suchenden Elements zurück, falls es im Array vorhanden ist. Anderenfalls wird -1 zurückgegeben.

2. Abbruchbedingung
3. Berechnung der Grenze des ersten und zweiten Drittels.
4. Berechnung der Grenze des zweiten und dritten Drittels.
5. Überprüfung, ob das gesuchte Element dem Element an Index *indexLeft* entspricht.
6. Element wurde gefunden, somit Rückgabe von *indexLeft*
8. Überprüfung, ob das gesuchte Element dem Element an Index *indexRight* entspricht.
9. Element wurde gefunden, somit Rückgabe von *indexRight*
11. Überprüfung, ob das gesuchte Element im ersten Drittel liegt
12. Rekursiver Aufruf, sodass im ersten Drittel gesucht wird  $\Rightarrow$  Grenzen werden von *lowerBound* sowie *indexLeft* (exklusiv) gebildet.
14. Überprüfung, ob das gesuchte Element im dritten Drittel liegt
15. Rekursiver Aufruf, sodass im dritten Drittel gesucht wird  $\Rightarrow$  Grenzen werden von *indexRight* (exklusiv) sowie *upperBound* gebildet.
17. Falls das gesuchte Element nicht im ersten und nicht im dritten Drittel liegt, liegt es im zweiten Drittel.  
Rekursiver Aufruf, sodass im dritten Drittel gesucht wird  $\Rightarrow$  Grenzen werden von *indexLeft* (exklusiv) sowie *indexRight* (exklusiv) gebildet.
19. Rückgabe des Wertes -1 wenn das gesuchte Element nicht im Array liegt.

#### d)

Die Rekursionsvorschrift für den Binary Search, der den Suchbereich auf zwei Teilbereiche aufteilt und **einen** Vergleich benötigt, um die Entscheidung zu treffen in welchem Bereich das gesuchte Element liegt, lautet  $T(n) = T(\frac{n}{2}) + 1$  (im worst case) und hat eine Komplexität von  $\mathcal{O}(\log_2 n)$

Der Ternary Search teilt den Suchbereich in **drei** Teile auf und benötigt zwei Vergleiche (im worst case), um zu entscheiden in welchem Bereich das gesuchte Element liegt. Folglich hat der Ternary Search die Rekursionsvorschrift  $T(n) = T(\frac{n}{3}) + 2$ . Mit Hilfe des Mastertheorems lässt sich folgende Komplexität ermitteln:

$$T(n) = T\left(\frac{n}{3}\right) + 2 \Rightarrow T\left(\frac{n}{3}\right) + \mathcal{O}(1)$$

Nach dem Mastertheorem gilt:  $a = 1$ ;  $b = 3$ ;  $f(n) = \mathcal{O}(1)$

$$\text{Es ist: } \Theta(n^{\log_3 1}) = \Theta(n^0) = \Theta(1) = f(n)$$

Nach Fall 2 des Mastertheorems gilt:  $T(n) = \mathcal{O}(1) \cdot \log_3 n = \mathcal{O}(\log_3 n)$

Da gilt  $\mathcal{O}(\log_3 n) < \mathcal{O}(\log_2 n)$  ist die Laufzeit des Ternary Search im Vergleich zur Laufzeit des Binary Search besser.

e)

- Minimale Anzahl an Vergleichen: Ein Vergleich je Rekursionsschritt, wenn davon ausgegangen wird, dass das Element im ersten Drittel liegen müsste  $\Rightarrow \log_3 n$
- Maximale Anzahl an Vergleichen: Zwei Vergleiche je Rekursionsschritt, wenn davon ausgegangen wird, dass das Element im zweiten oder im dritten Drittel liegen müsste  $\Rightarrow 2 \cdot \log_3 n$
- Durchschnittliche Anzahl an Vergleichen:

## Aufgabe 2

a)

---

**Algorithm 2:** Finde ein  $x$  und  $y$  mit  $x \neq y$ , sodass  $x + y = z$ 


---

```

1 function findTwoSummands(array [ ], z)
2 for  $i \leftarrow 0; i < \text{array.length}; i \leftarrow i + 1$  do
3    $x \leftarrow \text{array}[i];$ 
4   for  $j \leftarrow 0; j < \text{array.length}; j \leftarrow j + 1$  do
5      $y \leftarrow \text{array}[j];$ 
6     if  $x + y = z \wedge x \neq y$  then
7       return x, y;
8     end
9   end
10 end
11 return -1;
```

---

Für den Algorithmus werden zwei verschachtelte Schleifen benötigt. Jedes Element des Arrays wird dazu mit jedem Element des Arrays verglichen, ausgenommen sich selbst. Es werden somit  $n \cdot (n - 1) = n^2 - n$  Schleifendurchläufe (im Worst Case) getätigt, was einer Laufzeit von  $\mathcal{O}(n^2)$  entspricht.

b)

---

**Algorithm 3:** Finde ein  $x$  und  $y$  mit  $x \neq y$ , sodass  $x + y = z$  mit Binary Search
 

---

```

1 function findTwoSummands(array [ ], z)
2 for  $i \leftarrow 0; i < \text{array.length}; i \leftarrow i + 1$  do
3    $x \leftarrow \text{array}[i];$ 
4    $y \leftarrow z - x;$ 
5   if  $\text{binarySearch}(\text{array}, y) \neq -1$  then
6     return x, y;
7   end
8 end
9 return -1;
```

---

Der angegebene Algorithmus führt  $n$  mal die Binäre Suche durch (im Worst Case). Es

ist bekannt, dass die Binäre Suche eine Laufzeit von  $\mathcal{O}(\log n)$  hat. Es ergibt sich also eine Gesamtlaufzeit von  $\mathcal{O}(n \cdot \log n)$

c)

---

**Algorithm 4:** Finde ein  $x$  und  $y$  mit  $x \neq y$ , sodass  $x + y = z$

---

```
1 function findTwoSummands(array [ ], z)
2   hashTable;
3   for  $i \leftarrow 0; i < \text{array.length}; i \leftarrow i + 1$  do
4      $x \leftarrow \text{array}[i]$ ;
5      $y \leftarrow z - x$ ;
6     if  $\text{hashTable}[y] \neq \text{empty}$  then
7       return  $x, y$ ;
8     end
9      $\text{hashTable}[x] = x$ ;
10  end
11  return -1;
```

---

Vorausgesetzt wird eine Datenstruktur, die einen Einfüge- und Suchzugriff konstanter Laufzeit, also  $\mathcal{O}(1)$ , ermöglicht. Eine solche Datenstruktur ist beispielsweise eine Hash Table. Für alle  $n$  Element des Arrays wird eine Such- und ggfs. Einfügeoperation durchgeführt womit sich eine Gesamtlaufzeit von  $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$  ergibt.