Sarah Ertel
Patrick Greher
Eugen Ljavin

1	2	3	Σ
9	8	12	29

Übungsblatt Nr. 2

(Abgabetermin 03.05.2018)

Aufgabe 1

a)

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

Je nach Rundung ist ternery sogar besser

b)

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

Je nach Rundung ist binary gleich schnell

c)

Algorithm 1: Ternary Search

```
function ternarySearch(array[ ], key, lowerBound, upperBound)
if upperBound ≥ lowerBound then
    indexLeft ← (upperBound - lowerBound) / 3 + lowerBound;
    indexRight ← (upperBound - lowerBound) / 3 + indexLeft;
    if array[indexLeft] = key then
        return indexLeft;
    end
    if array[indexRight] = key then
        return indexRight;
    end
    if array[indexLeft] > key then
        return ternarySearch(array, key, lowerBound, indexLeft − 1);
    end
    if array[indexRight] < key then
        return ternarySearch(array, key, indexRight + 1, upperBound);
    end
    return ternarySearch(array, key, indexLeft + 1, indexRight − 1);
end
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 weden.
   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 gesucht 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 bnötigt zwei Vergleiche (im worst case), um zu enscheiden 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:

$$
\begin{aligned}
T(n) = T\left(\frac{n}{3}\right) + 2 &\Rightarrow T\left(\frac{n}{3}\right) + \mathcal{O}(1) \\
\text{Nach dem Mastertheorem gilt: } a &= 1;\ b = 3;\ f(n) = \mathcal{O}(1) \\
\text{Es ist: } \Theta(n^{\log_3 1}) &= \Theta(n^0) = \Theta(1) = f(n) \\
\text{Nach Fall 2 des Mastertheorems gilt: } T(n) &= \mathcal{O}(1) \cdot \log_3 n = \mathcal{O}(\log_3 n)
\end{aligned}
$$

Da gilt $\mathcal{O}(\log_3 n) < \mathcal{O}(\log_2 n)$ ist die Laufzeit des Ternary Sarch 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 $\Rightarrow 2 \cdot \log_3 n$

  korrekt
- Durchschnittliche Anzahl an Vergleichen:

Aufgabe 2

a)

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

```
function findTwoSummands(array[ ], z)
for i ← 0; i < array.length; i ← i + 1 do
    x ← array[i];
    for j ← 0; j < array.length; j ← j + 1 do
        y ← array[j];
        if x + y = z ∧ x ≠ y then
            return x, y;
        end
    end
end
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

```
function findTwoSummands(array[ ], z)
for i ← 0; i < array.length; i ← i + 1 do
    x ← array[i];
    y ← z - x;
    if binarySearch(array, y) ≠ −1 then
        return x, y;
    end
end
return -1;
```

Der angegebende 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 $x + y = z$

```
function findTwoSummands(array[ ], z)
hashTable;
for i ← 0; i < array.length; i ← i + 1 do
    x ← array[i];
    y ← z - x;
    if hashTable[y] ≠ empty then
        return x, y;
    end
    hashTable[x] = x;
end
return -1;
```

Ungleichheit von x und y forcen

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 durchgefürt womit sich eine Gesamtlaufzeit von $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$ ergibt.