

2019_t1_A

⚠ Disclaimer

Alles was hier drinnen steht kann Fehler enthalten!, Falls dir etwas auffällt melde dich gerne auf Discord bei mir ([@xmozz](#))

A1 - Algorithmenanalyse

Stoff: [2. Analyse von Algorithmen](#)

a)

(12 Punkte) Tragen Sie für die Codestücke **FunktionA** und **FunktionB** die Laufzeit und den Rückgabewert (z) in Abhängigkeit von n , jeweils in Θ -Notation in die nachfolgende Tabelle ein.

FunktionA(n):

```

 $x \leftarrow n^2$ 
 $z \leftarrow 1$ 
while  $x > 1$ 
     $x \leftarrow \frac{x}{3}$ 
     $z \leftarrow 3z$ 
return  $z$ 
```

FunktionB(n):

```

 $j \leftarrow 0$ 
 $z \leftarrow 1$ 
for  $i \leftarrow 1$  bis  $3n$ 
    if  $i \bmod 3 = 0$  then
         $j \leftarrow j + i$ 
     $j \leftarrow \frac{2}{3}j - n$ 
while  $j > 0$ 
     $z \leftarrow nz$ 
     $j \leftarrow j - 1$ 
return  $z$ 
```

	FunktionA	FunktionB
Laufzeit	$\Theta(\log n)$	$\Theta(n^2)$
Rückgabewert (z)	$\Theta(3^{2 \cdot \log n})$	$\Theta(n^{n^2})$

b)

(4 Punkte) Sei $c = \frac{1}{2}$. Finden Sie das kleinste $n_0 \in \mathbb{N}^+$, sodass für alle $n \geq n_0$ gilt

$$2 \cdot \sqrt{\frac{n+1}{4}} \leq cn.$$

Kleindestes $n_0 =$

Was haben Sie somit gezeigt? Kreuzen Sie Zutreffendes an:

- $2 \cdot \sqrt{\frac{n+1}{4}}$ ist in $\Omega(n)$
- $2 \cdot \sqrt{\frac{n+1}{4}}$ ist in $O(n)$
- $2 \cdot \sqrt{\frac{n+1}{4}}$ ist in $\Theta(n)$
- keine der zuvor genannten Aussagen

$$2 \cdot \sqrt{\frac{n+1}{4}} \leq \frac{1}{2} \cdot n$$

1. Ungleichung vereinfachen:

$$\begin{aligned} 2 \cdot \sqrt{\frac{n+1}{4}} &= \sqrt{n+1} \\ \Rightarrow \sqrt{n+1} &\leq \frac{1}{2}n \end{aligned}$$

2. Beide Seiten quadrieren (gültig, da beide positiv für ($n \geq 1$):

$$\begin{aligned} n+1 &\leq \left(\frac{1}{2}n\right)^2 = \frac{1}{4}n^2 \\ \Rightarrow 4(n+1) &\leq n^2 \Rightarrow 4n+4 \leq n^2 \Rightarrow n^2 - 4n - 4 \geq 0 \end{aligned}$$

3. Nullstellen der quadratischen Ungleichung berechnen:

$$\begin{aligned} n &= \frac{4 \pm \sqrt{(-4)^2 + 4 \cdot 1 \cdot 4}}{2} = \frac{4 \pm \sqrt{16 + 16}}{2} = \frac{4 \pm \sqrt{32}}{2} = \frac{4 \pm 4\sqrt{2}}{2} = 2 \pm 2\sqrt{2} \\ \Rightarrow n &\geq 2 + 2\sqrt{2} \approx 4.8284 \end{aligned}$$

$$n_0 = 5$$

c)

- (i) Wenn $f = O(g)$ und $g = \Omega(h)$ gilt, dann gilt auch $f = \Theta(h)$.

Wahr Falsch

- (ii) Wenn die Worst-Case Laufzeit eines Algorithmus über alle Eingaben der Eingabegröße n in $O(n^2)$ liegt, so kann seine asymptotische Laufzeit nicht in $\Omega(n^3)$ liegen.

Wahr Falsch

- (iii) Wenn die Worst-Case Laufzeit eines Algorithmus über alle Eingaben der Eingabegröße n in $\Omega(n^2)$ liegt, so kann seine asymptotische Laufzeit nicht in $\Omega(n^3)$ liegen.

Wahr Falsch

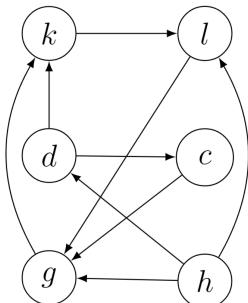
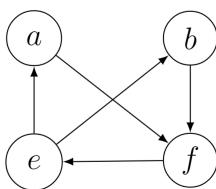
A2 - Graphen

Stoff: 3. Graphen

a)

- a) (5 Punkte) Gegeben ist der nachfolgende Graph $H = (V_H, E_H)$ mit 12 Knoten. Verwenden Sie die aus der Vorlesung bekannten Definitionen und kreuzen Sie Zutreffendes in der untenstehenden Tabelle an. Betrachten Sie für jede Zeile der Tabelle den gerichteten Teilgraphen $G = (V_G, E_G)$ von H mit der in der Tabelle gegebenen Knotenmenge V_G und $E_G = \{(a, b) \mid a, b \in V_G, (a, b) \in E_H\}$.

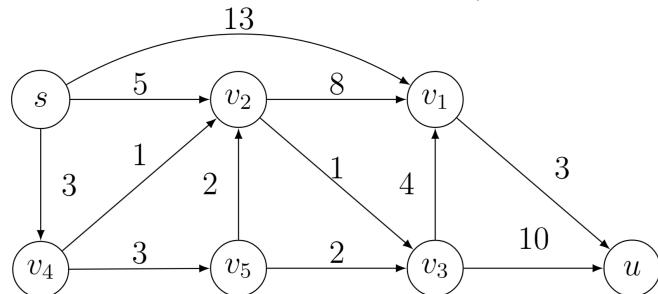
(korrekte Zeile: +1 Punkt, inkorrekte Zeile: -1 Punkt, keine Antwort (Zeile ohne Kreuz): 0 Punkte. Minimum für diese Unteraufgabe: 0 Punkte)



	V_G	G ist schwach zusammenhängend	G ist schwache Zusammenhangskomp. von H	G ist stark zusammenhängend	G ist starke Zusammenhangskomp. von H	keine der zuvor genannten Aussagen trifft zu
$\{c, l\}$	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
$\{a, e, f\}$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
$\{c, d, g, h, k, l\}$	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
$\{g, k, l\}$	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
$\{i\}$	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

b)

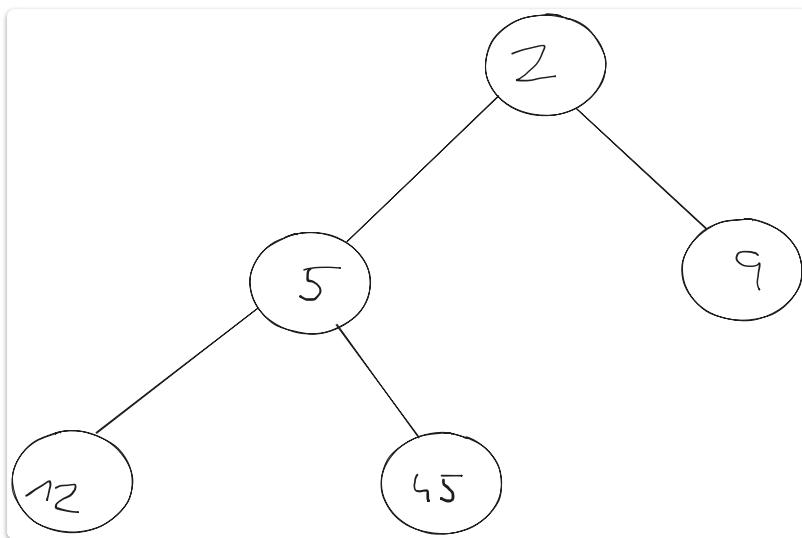
(10 Punkte) Wenden Sie den aus der Vorlesung bekannten *Algorithmus von Dijkstra in der Implementierung mit einer Liste*, zum Finden eines kürzesten Pfades von s nach u auf dem gegebenen Graphen an. Dokumentieren Sie für jeden Schritt, bei dem sich die Menge Discovered ändert (diese enthält die bereits untersuchten Knoten), für jeden Knoten $d(\cdot)$ und die Mengen Discovered und L. Extrahieren Sie anschließend aus dem Ablauf einen kürzesten Pfad, sowie seine Länge.



Knoten	Discovered	s	v_1	v_2	v_3	v_4	v_5	u
$d(\cdot)$	s	0	13	5	∞	3	∞	∞
	s, v_4	0	13	4	∞	3	6	∞
	s, v_4, v_2	0	12	4	5	3	6	∞
	s, v_4, v_2, v_3	0	9	4	5	3	6	15
	s, v_4, v_2, v_3, v_1	0	9	4	5	3	6	12
	s, v_4, v_2, v_3, v_1, u	0	9	4	5	3	6	12

c)

(3 Punkte) Zeichnen Sie einen Min-Heap zur Menge $\{12, 9, 2, 45, 5\}$.



A3 - Greedy

Stoff: 4. Greedy-Algorithmen

a)

Gegeben ist eine Währung deren Münzen eine Stückelung von 1, 3, 6 und 13 haben. Liefert der Greedy-Algorithmus zum Geldwechseln aus der Vorlesung bei dieser Stückelung immer ein optimales Ergebnis?

- Ja Nein

Begründung/Gegenbeispiel:

Wenn man 24 haben will, liefert der Greedy Algorithmus: $1 \cdot 13 + 1 \cdot 6 + 1 \cdot 3 + 2 \cdot 1$

Man braucht also: 5 Münzen.

Die Optimale Lösung ist: $4 \cdot 6$

b)

Sei G ein gewichteter Graph, bei dem alle Kanten unterschiedliche Gewichte haben. Wenn Sie den Algorithmus von Kruskal und den Algorithmus von Prim verwenden um einen Minimum Spanning Tree (MST) von G zu berechnen, finden dann beide Algorithmen immer denselben MST?

- Ja Nein

Begründung/Gegenbeispiel:

Wenn alle Kantengewichte gleich sind, dann ist der MST eindeutig. Sowohl Prim als auch Kruskal können nur den einen finden, da die kleineren Kanten immer bevorzugt werden.

c)

- c) Sei T ein Spannbaum in einem gewichteten Graphen G . Wir nennen die Kanten in T mit dem höchsten Gewicht die Bottleneck-Kanten von T . Wir sagen T ist ein Minimum Bottleneck Spanning Tree (MBST) falls es keinen Spanning Tree von G gibt, dessen Bottleneck-Kanten ein geringeres Gewicht als die Bottleneck-Kanten von T haben. Ist jeder MBST auch ein minimaler Spannbaum?

- Ja Nein

Begründung/Gegenbeispiel:

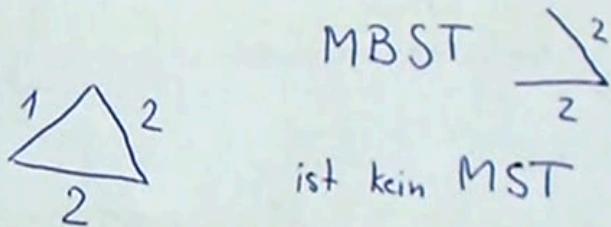
Bottleneck Kante --> Größte Kante

MBST --> Wenn er die kleinste Bottleneck Kante hat von allen Spannbäumen

Frage ist das ein minimaler Spannbaum?

Ja Nein

Begründung/Gegenbeispiel:



Aber:

Jeder MST ist auch MBST

Beweis: die schwierigste Kante ist am leichtesten in „ihrem“ Schnitt

d)

Sei G ein Graph, bei dem jedem Knoten v eine reelle Zahl $w(v)$ als Gewicht zugeordnet ist. Für eine Menge von Knoten S schreiben wir $w(S)$ für die Summe $\sum_{v \in S} w(v)$ der Gewichte der Knoten in S . Eine überdeckende Knotenmenge von G ist eine Menge S von Knoten, so dass jede Kante von G zu einem Knoten in S incident ist. Eine überdeckende Knotenmenge S heißt minimal, wenn für jede andere überdeckende Knotenmenge S^* stets $w(S) \leq w(S^*)$ gilt. Gegeben ist der folgende Greedy-Algorithmus:

Greedy(n):

Input: Graph $G = (V, E)$ mit gewichteten Knoten

$S \leftarrow \emptyset$

$S_E \leftarrow \emptyset$

while $S_E \neq E$

Wähle einen Knoten $v \in V \setminus S$ mit minimalem Gewicht.

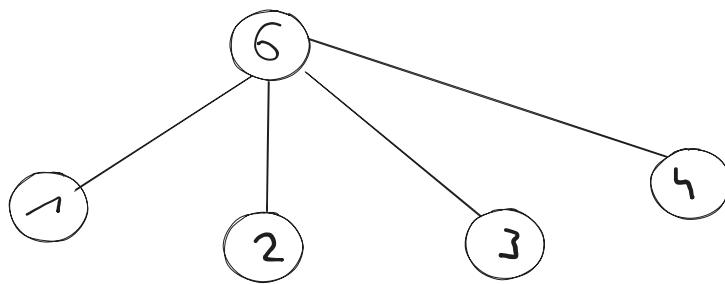
Füge v zu S hinzu.

Füge alle Kanten die indizent zu v sind zu S_E hinzu.

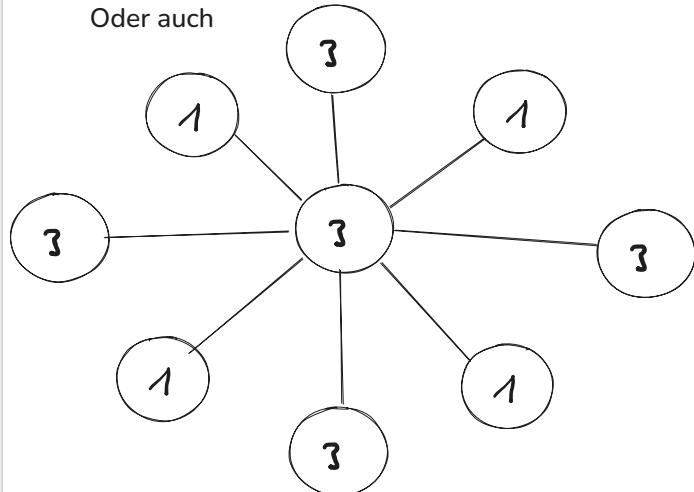
Findet dieser Algorithmus immer eine minimale überdeckende Knotenmenge?

Ja Nein

Begründung/Gegenbeispiel:



Oder auch



A4 - Suchbäume und Hashing

Stoff: [6. Suchbäume](#) und [7. Hashing](#)

(12 Punkte) Im Folgenden geht es um die Best-Case- und Worst-Case-Laufzeiten unterschiedlicher Operationen für Hashing mit Verkettung von Überläufern, offenes Hashing und bei binären Suchbäumen in Abhängigkeit der n bereits eingefügten Elemente.

(i)

AVL-Bäume

Operation	Best-Case	Worst-Case
Erfolgreiches Einfügen	$\Theta()$	$\Theta()$
Erfolgreiches Suchen	$\Theta()$	$\Theta()$
Erfolgloses Suchen	$\Theta()$	$\Theta()$

Alles $\log n$

(ii)

Offenes Hashing mit linearem Sondieren

Operation	Best-Case	Worst-Case
Erfolgreiches Einfügen	$\Theta()$	$\Theta()$
Erfolgreiches Suchen	$\Theta()$	$\Theta()$
Erfolgloses Suchen	$\Theta()$	$\Theta()$

- Erfolgreiches Einfügen
 - Bestcase $\Theta(1)$
 - Worstcase $\Theta(n)$
- Erfolgreiches Suchen
 - Bestcase $\Theta(1)$
 - Worstcase $\Theta(n)$
- Erfolgloses Suchen
 - Bestcase $\Theta(1)$
 - Worstcase $\Theta(n)$

(iii)

Hashing mit Verkettung der Überläufer

Operation	Best-Case	Worst-Case
Erfolgreiches Einfügen	$\Theta()$	$\Theta()$
Erfolgreiches Suchen	$\Theta()$	$\Theta()$
Erfolgloses Suchen	$\Theta()$	$\Theta()$

- Erfolgreiches Einfügen
 - Bestcase $\Theta(1)$
 - Worstcase $\Theta(1)$ Kommt auf die Implementierung an, wenn es am Ende von der Liste ergänzt wird, wäre das hier nicht $\Theta(1)$ sondern $\Theta(n)$. Allerdings beispielsweise bei der Programmieraufgabe P4 haben wir das mit $\Theta(1)$ implementiert. In den Slides steht es auch mit der effizienteren Variante.
- Erfolgreiches Suchen
 - Bestcase $\Theta(1)$
 - Worstcase $\Theta(n)$
- Erfolgloses Suchen
 - Bestcase $\Theta(1)$
 - Worstcase $\Theta(n)$

(iv)

Binäre Suchbäume ohne Höhenbalancierung

Operation	Best-Case	Worst-Case
Erfolgreiches Einfügen	$\Theta()$	$\Theta()$
Erfolgreiches Suchen	$\Theta()$	$\Theta()$
Erfolgloses Suchen	$\Theta()$	$\Theta()$

- Erfolgreiches Einfügen
 - Bestcase $\Theta(\log n)$
 - Worstcase $\Theta(n)$
- Erfolgreiches Suchen
 - Bestcase $\Theta(\log n)$
 - Worstcase $\Theta(n)$
- Erfolgloses Suchen
 - Bestcase $\Theta(\log n)$
 - Worstcase $\Theta(n)$

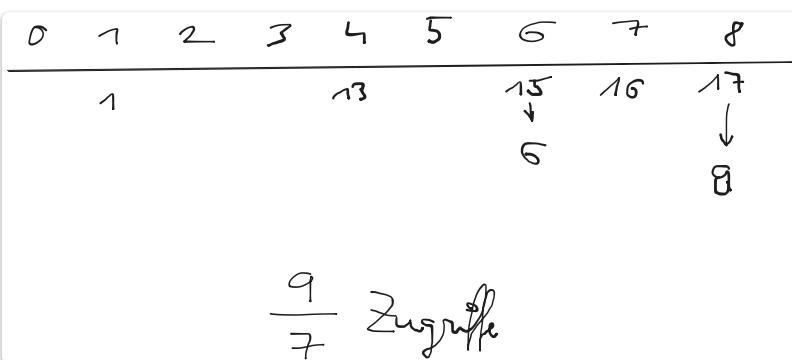
b)

(8 Punkte) Gegeben sind folgende natürliche Zahlen:

$$[16, 8, 17, 6, 1, 13, 15]$$

Fügen Sie diese in der vorgegebenen Reihenfolge jeweils in folgende Varianten von anfangs leeren Hashtabellen der Größe $m = 9$ ein. Berechnen Sie dann die durchschnittliche Anzahl von Schlüsselvergleichen bei einer erfolgreichen Suche eines Elements, wobei jedes dieser Elemente mit gleicher Wahrscheinlichkeit gesucht wird. Für das Sondieren gilt $i = 0, 1, \dots, m - 1$.

(i)

(i) Verkettung der Überläufer mit $h(k) = k \bmod m$.

(ii)

- (ii) Double-Hashing mit $h(k) = (h_1(k) + ih_2(k)) \bmod m$, $h_1(k) = k \bmod m$ und $h_2(k) = 1 + (k \bmod 5)$.

0	1	2	3	4	5	6	7	8
1	15	17	13		6	16	8	

$$\frac{10}{7}$$

A5 - Sortierverfahren und Divide-and-Conquer

Stoff: 5. Divide and Conquer

a)

- a) (4 Punkte) Können die folgenden Arrays als Run in einem Ablauf eines Timsort Algorithmus mit $\text{Min_Run} = 6$ vorkommen? Kreuzen Sie Zutreffendes an.

	Ja	Nein
[2, 2, 2, 2, 2, 2]	<input type="checkbox"/>	<input checked="" type="checkbox"/>
[2, 4, 7, 6, 8, 11, 14]	<input checked="" type="checkbox"/>	<input type="checkbox"/>
[7, 7, 6, 6, 5, 5, 4]	<input type="checkbox"/>	<input checked="" type="checkbox"/>
[2, 6, 9, 11, 18, 15]	<input type="checkbox"/>	<input checked="" type="checkbox"/>

(korrektes Kreuz: +1 Punkt, inkorrekte Kreuze: -1 Punkt, kein Kreuz: 0 Punkte.
Minimum für diese Unteraufgabe: 0 Punkte)

1. Ja ist komplett Monoton
2. Nein Monotonie bricht bei 7 --> 6 daher sollte es nur 6 lang sein und nicht 7
3. Hier ist es nicht streng monoton, was bei fallenden so sein muss. Daher falsch dass noch 4er ergänzt wird
4. Passt so, da wir die ersten 6 in richtiger Monotonie haben und dann noch den 15 einfügen.

b)

(16 Punkte) Dual Pivot Quicksort ist eine Variante von Quicksort, die in der Praxis effizienter ist als ein normaler Quicksort. Dual Pivot Quicksort funktioniert im Wesentlichen wie Quicksort, mit den folgenden Änderungen:

Bei Dual Pivot Quicksort werden zwei Elemente als Pivot gewählt. Wir nennen das kleinere Pivot Element linkes Pivot (LP) und das größere Pivot Element rechtes Pivot (RP). Das Array wird dann in drei Teile partitioniert: Die Elemente die kleiner sind als LP, die Elemente die größer oder gleich LP aber kleiner oder gleich RP sind, und die Elemente die größer als RP sind. Danach wird Dual Pivot Quicksort rekursiv auf jede der drei Partitionen aufgerufen.

Kreuzen Sie im Pseudocode auf der nächsten Seite die Codezeilen an, die ausgeführt werden müssen, um eine funktionierende Implementierung von Dual Pivot Quicksort zu erhalten. Je Block (grau hinterlegte Box) ist genau eine Auswahl korrekt.

(richtiges Kreuz +4 Punkte, falsches Kreuz/mehrere Kreuze im Block -4 Punkte, kein Kreuz 0 Punkte. Minimum für diese Unteraufgabe: 0 Punkte)

DualPivotQuicksort(A , left , right):

```
// Array A, wird von Element left bis Element right sortiert
if  $\text{right} - \text{left} \geq 1$  then
    // Erstes (=left) und letztes (=right) Element als Pivot
     $p \leftarrow \min\{A[\text{left}], A[\text{right}]\}$  // p ist das kleinere Pivotelement
     $q \leftarrow \max\{A[\text{left}], A[\text{right}]\}$  // q ist das größere Pivotelement
     $l \leftarrow \text{left} + 1$  // l begrenzt die linke Partition
     $g \leftarrow \text{right} - 1$  // g begrenzt die rechte Partition
     $k \leftarrow l$  // k ist die Laufvariable der Partitionierung

    while  $k \leq g$ 
        while  $l \leq g$ 
            while  $k \leq q$ 
                if  $A[k] < p$  then
                    Tausche  $A[k]$  und  $A[l]$ 
                     $l \leftarrow l + 1$ 
                else if  $A[k] > q$  then
                    while  $A[g] > q$  und  $k < g$ 
                        g  $\leftarrow g - 1$ 
                        k  $\leftarrow k + 1$ 
                        l  $\leftarrow l + 1$ 
                    Tausche  $A[k]$  und  $A[g]$ 
                     $g \leftarrow g - 1$ 
                    if  $A[k] < p$  then
                        Tausche  $A[k]$  und  $A[l]$ 
                         $l \leftarrow l + 1$ 
                        g  $\leftarrow g - 1$ 
                        k  $\leftarrow k + 1$ 
                        l  $\leftarrow l + 1$ 
                     $l \leftarrow l - 1$ 
                     $g \leftarrow g + 1$ 
                     $A[\text{left}] \leftarrow A[l]$ 
                     $A[l] \leftarrow p$  // p wird an die finale Position gesetzt
                     $A[\text{right}] \leftarrow A[g]$ 
                     $A[g] \leftarrow q$  // q wird an die finale Position gesetzt
                    call DualPivotQuicksort( $A$ ,  $\text{left} + 1$ ,  $l$ )
                    call DualPivotQuicksort( $A$ ,  $l + 1$ ,  $g$ )
                    call DualPivotQuicksort( $A$ ,  $g + 1$ ,  $\text{right} - 1$ )
                call DualPivotQuicksort( $A$ ,  $\text{left}$ ,  $p - 1$ )
                call DualPivotQuicksort( $A$ ,  $p + 1$ ,  $q - 1$ )
                call DualPivotQuicksort( $A$ ,  $q + 1$ ,  $\text{right}$ )
            call DualPivotQuicksort( $A$ ,  $\text{left}$ ,  $l - 1$ )
            call DualPivotQuicksort( $A$ ,  $l + 1$ ,  $g - 1$ )
            call DualPivotQuicksort( $A$ ,  $g + 1$ ,  $\text{right}$ )
```