Übungen zur Vorlesung

Algorithmen und Datenstrukturen

Übungsblatt 2 – Lösungen der Präsenzübungen

Arbeitsgruppe Kryptographie und Komplexitätstheorie
Prof. Dr. Marc Fischlin
Dr. Christian Janson
Patrick Harasser
Felix Rohrbach

Sommersemester 2019
Veröffentlicht am: 04.05.2019
Letzte Änderung am: 05.05.2019
Abgabe am: 10.05.2019, 12:00 Uhr

P1 (Gruppendiskussion)

Nehmen Sie sich etwas Zeit, um die folgenden Fachbegriffe in einer Kleingruppe zu besprechen, sodass Sie anschließend in der Lage sind, die Begriffe dem Rest der Übungsgruppe zu erklären:

(a) Asymptotische Notation (O, o, Ω, ω, Θ)
(b) Divide-and-Conquer Ansatz
(c) Bubblesort, Mergesort, Quicksort

P2 (Rechenregeln für asymptotische Notation)

Es seien $f_1, f_2, g_1, g_2, f, g, h \colon \mathbb{N} \to \mathbb{R}_{>0}$ Funktionen und $k \in \mathbb{R}_{>0}$ eine Konstante. Beweisen Sie folgende Aussagen:

(a) i) $f(n) = O(g(n))$ genau dann wenn $g(n) = \Omega(f(n))$;
ii) $f(n) = o(g(n))$ genau dann wenn $g(n) = \omega(f(n))$;
(b) $o(g(n)) \subseteq O(g(n))$, und $\omega(g(n)) \subseteq \Omega(g(n))$.
(c) $O(g(n)) \cap \Omega(g(n)) = \Theta(g(n))$, und $o(g(n)) \cap \Omega(g(n)) = \emptyset$.
(d) i) Ist $f_1(n) = O(g_1(n))$ und $f_2(n) = O(g_2(n))$, dann gilt $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$;
ii) Ist $f_1(n) = O(g_1(n))$ und $f_2(n) = O(g_2(n))$, dann gilt $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$;
iii) Es gilt $f(n) = O(f(n))$;
iv) Ist $f(n) = O(g(n))$ und $g(n) = O(h(n))$, dann gilt $f(n) = O(h(n))$.

Lösung. Wir erinnern zunächst an die Definitionen der verschiedenen asymptotischen Klassen:

- $f(n) = O(g(n))$ genau dann wenn $(\exists C > 0)(\exists N_0 \in \mathbb{N})(\forall n \geq N_0)(0 \leq f(n) \leq Cg(n))$;
- $f(n) = \Omega(g(n))$ genau dann wenn $(\exists C' > 0)(\exists N_0' \in \mathbb{N})(\forall n \geq N_0')(0 \leq C'g(n) \leq f(n))$;
- $f(n) = \Theta(g(n))$ genau dann wenn $(\exists C > 0)(\exists C' > 0)(\exists M_0 \in \mathbb{N})(\forall n \geq M_0)(0 \leq C'g(n) \leq f(n) \leq Cg(n))$;
- $f(n) = o(g(n))$ genau dann wenn $(\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(0 \leq f(n) < cg(n))$;
- $f(n) = \omega(g(n))$ genau dann wenn $(\forall c' > 0)(\exists n_0' \in \mathbb{N})(\forall n \geq n_0')(0 \leq c'g(n) < f(n))$.

Wir kommen nun zu den Lösungen der einzelnen Punkte:

(a) i) Angenommen, $f(n) = O(g(n))$, und seien $C > 0$, $N_0 \in \mathbb{N}$ wie in der Definition. Dann setze $C' = \frac{1}{C} > 0$ und $N_0' = N_0$. Nach Voraussetzung gilt, für alle $n \geq N_0' = N_0$, dass $0 \leq f(n) \leq Cg(n)$, und somit auch $0 \leq C'f(n) \leq g(n)$, also $g(n) = \Omega(f(n))$.
Ist nun umgekehrt $g(n) = \Omega(f(n))$, seien $C' > 0$, $N_0' \in \mathbb{N}$ wie in der Definition. Setze $C = \frac{1}{C'} > 0$ und $N_0 = N_0'$. Nach Voraussetzung gilt, für alle $n \geq N_0 = N_0'$, dass $0 \leq C'f(n) \leq g(n)$, also auch $0 \leq f(n) \leq Cg(n)$, und somit $f(n) = O(g(n))$.
ii) Angenommen, $f(n) = o(g(n))$, und sei $c' > 0$ beliebig wie in der Definition von $\omega(f(n))$. Sei $n_0' \in \mathbb{N}$ die Schranke n_0 in der Definition von $o(g(n))$ für $c = \frac{1}{c'}$. Dann gilt, für alle $n \geq n_0'$, dass $0 \leq f(n) < \frac{1}{c'}g(n)$, also $0 \leq c'f(n) < g(n)$ und somit $g(n) = \omega(f(n))$.
Ist nun ungekehrt $g(n) = \omega(f(n))$, betrachte ein beliebiges $c > 0$ wie in der Definition von $o(g(n))$. Sei $n_0 \in \mathbb{N}$ die Schranke n_0' in der Definition von $\omega(f(n))$ für $c' = \frac{1}{c}$. Dann gilt, für alle $n \geq n_0$, dass $0 \leq \frac{1}{c}f(n) < g(n)$, also $0 \leq f(n) < cg(n)$ und somit $f(n) = o(g(n))$.

(b) Angenommen, $f(n) = o(g(n))$. Dann gilt die Bedingung in der Definition von $o(g(n))$ für alle $c > 0$, im Besonderen also für $c = 1$. Sei $n_0 \in \mathbb{N}$ die entsprechende Schranke. Wenn wir nun $C = 1$ und $N_0 = n_0$ in der Definition von $O(g(n))$ setzen, erhalten wir dass $f(n) = O(g(n))$.

Wenn hingegen $f(n) = \omega(g(n))$, dann ist $g(n) = o(f(n))$ wegen (a), also wie eben gezeigt auch $g(n) = O(f(n))$, und somit wieder $f(n) = \Omega(g(n))$.

(c) Offensichtlich gilt $\Theta(g(n)) \subseteq O(g(n))$ und $\Theta(g(n)) \subseteq \Omega(g(n))$. In der Tat, wenn $f(n) = \Theta(g(n))$, kann man einfach $C > 0$, $C' > 0$ wie in der Definition von $\Theta(g(n))$, und $N_0 = N_0' = M_0 \in \mathbb{N}$ benutzen, um die Behauptung zu zeigen. Für die umgekehrte Richtung nehme man an, dass $f(n) \in O(g(n)) \cap \Omega(g(n))$, also $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$. Seien $C > 0$, $N_0 \in \mathbb{N}$, $C' > 0$, und $N_0' \in \mathbb{N}$ die entsprechenden Parameter. Wenn wir nun $M_0 = \max(N_0, N_0')$ setzen, dann sind die Ungleichungen in der Definition von $\Theta(g(n))$ klarerweise erfüllt.

Angenommen, $f(n) = o(g(n))$ und $f(n) = \Omega(g(n))$, und seien $C' > 0$, $N_0' \in \mathbb{N}$ die Parameter wie in der Definition von $\Omega(g(n))$. Betrachte dann $c = C' > 0$, und sei $n_0 \in \mathbb{N}$ die entsprechende Schranke wie in der Definition von $o(g(n))$. Dann gilt, für alle $n \geq \max(N_0', n_0)$, dass $C'g(n) \leq f(n) < C'g(n)$, also $C' < C'$, was ein Widerspruch ist. Folglich ist $o(g(n)) \cap \Omega(g(n)) = \emptyset$.

(d) i) Angenommen, $f_1(n) = O(g_1(n))$ und $f_2(n) = O(g_2(n))$, und seien $C^1 > 0$, $N_0^1 \in \mathbb{N}$, $C^2 > 0$, und $N_0^2 \in \mathbb{N}$ wie in der Definition von $O(g_1(n))$ und $O(g_2(n))$. Setze $C = C^1 + C^2$ und $N_0 = \max(N_0^1, N_0^2)$. Dann gilt in der Tat dass, für alle $n \geq N_0$,

$$\begin{aligned} 0 \leq f_1(n) + f_2(n) &\leq C^1 g_1(n) + C^2 g_2(n) \leq C^1 \max(g_1(n), g_2(n)) + C^2 \max(g_1(n), g_2(n)) = \\ &= (C^1 + C^2) \max(g_1(n), g_2(n)) = C \max(g_1(n), g_2(n)), \end{aligned}$$

also $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.

ii) Angenommen, $f_1(n) = O(g_1(n))$ und $f_2(n) = O(g_2(n))$, und seien $C^1 > 0$, $N_0^1 \in \mathbb{N}$, $C^2 > 0$, und $N_0^2 \in \mathbb{N}$ wie in der Definition von $O(g_1(n))$ und $O(g_2(n))$. Setze $C = C^1 \cdot C^2$ und $N_0 = \max(N_0^1, N_0^2)$. Dann gilt in der Tat, für alle $n \geq N_0$, dass $0 \leq f_1(n) \cdot f_2(n) \leq C^1 g_1(n) \cdot C^2 g_2(n) = C g_1(n) g_2(n)$, also $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

iii) Es ist $f(n) = f(n) \cdot 1 = O(f(n) \cdot 1) = O(f(n))$.

iv) Angenommen, $f(n) = O(g(n))$ und $g(n) = O(h(n))$, und seien $C^1 > 0$, $N_0^1 \in \mathbb{N}$, $C^2 > 0$, und $N_0^2 \in \mathbb{N}$ wie in der Definition von $O(g(n))$ und $O(h(n))$. Setze $C = C^1 \cdot C^2$ und $N_0 = \max(N_0^1, N_0^2)$. Dann gilt, für alle $n \geq N_0$, dass $0 \leq f(n) \leq C^1 g(n) \leq C^1 C^2 h(n) = C h(n)$, also $f(n) = O(h(n))$. □

P3 (Rechnen mit asymptotischer Notation)

Tragen Sie für jedes Paar von Funktionen $f, g \colon \mathbb{N} \to \mathbb{R}_{>0}$ in der folgenden Tabelle ein, ob f in der Ordnung O, o, Ω, ω, und/oder Θ ist. Hier sind $k \geq 1$, $\varepsilon > 0$, $c > 1$, und $r < s$ Konstanten. Begründen Sie Ihre Wahl.

$f(n)$	$g(n)$	O	o	Ω	ω	Θ
$\log^k(n)$	n^ε					
n^k	c^n					
2^n	$2^{n/2}$					
$n^{\log(c)}$	$c^{\log(n)}$					
n^r	n^s					
$\log(n!)$	$\log(n^n)$					

Lösung. Wir besprechen jede Zeile der Tabelle separat:

(a) Aus der Theorie der Grenzwerte ist bekannt, dass $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$, was genau bedeutet, dass $f(n) = o(g(n))$. Nach Aufgabe P2 ist nun bereits der ganze Punkt gelöst: Da $o(g(n)) \subseteq O(g(n))$ ist $f(n) = O(g(n))$; nachdem wir wissen, dass $o(g(n)) \cap \Omega(g(n)) = \emptyset$, muss $f(n) \neq \Omega(g(n))$, und somit auch $f(n) \neq \omega(g(n))$, da $\omega(g(n)) \subseteq \Omega(g(n))$. Und da wiederum $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ und $f(n) \neq \Omega(g(n))$, gilt auch $f(n) \neq \Theta(g(n))$.

(b) Die Lösung ist ident zu (a).

(c) Beachte, dass $g(n) = o(f(n))$, nachdem $\lim_{n\to+\infty} \frac{g(n)}{f(n)} = 0$. Das Argument ist nun wie in (a), aber mit f und g vertauscht: Es ist $g(n) = O(f(n))$, $g(n) \neq \Omega(f(n))$, $g(n) \neq \omega(f(n))$, und $g(n) \neq \Theta(f(n))$. Wir können nun wieder Aufgabe P2 benutzen, um die gesuchten Ordnungen herzuleiten: Wenn wir f und g vertauschen, müssen wir auch O durch Ω und o durch ω ersetzen, und umgekehrt. Wir erhalten also $f(n) = \omega(g(n))$, $f(n) = \Omega(g(n))$, $f(n) \neq O(g(n))$, $f(n) \neq o(f(n))$, und $f(n) \neq \Theta(g(n))$.

(d) Man beachte, dass $f(n) = 2^{\log(n)\log(c)} = g(n)$, also $f(n) = g(n)$. Somit gilt trivialerweise $f(n) = \Theta(g(n))$, also auch $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$. Und nachdem $o(g(n)) \cap \Omega(g(n)) = \emptyset$ und $O(g(n)) \cap \omega(g(n)) = \emptyset$, muss auch $f(n) \notin o(g(n))$ und $f(n) \notin \omega(g(n))$ gelten.

(e) Nach Annahme ist $r < s$, also gilt $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} n^{r-s} = 0$. Man argumentiere nun wie in (a).

(f) Offensichtlich gilt $n! \leq n^n$ für alle $n \in \mathbb{N}$, und da $\log(\cdot)$ streng monoton wachsend ist, ist auch $\log(n!) \leq \log(n^n)$. Daraus folgt unmittelbar, dass $\log(n!) = O(\log(n^n)) = O(n \log n)$, und damit $f(n) \neq \omega(g(n))$.

Wir zeigen nun, dass auch $f(n) = \Omega(g(n))$. Wir behaupten, dass für $C' = \frac{1}{3} > 0$ und $N_0' = 1 \in \mathbb{N}$ die Definition erfüllt ist: Für alle $n \geq N_0'$ gilt

$$\begin{aligned}\log(n!) = \sum_{i=1}^{n} \log(i) = 0 + \sum_{i=2}^{\lfloor n/2 \rfloor} \log(i) + \sum_{i=\lfloor n/2 \rfloor + 1}^{n} \log(i) \geq \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) + \left(n - \left\lfloor \frac{n}{2} \right\rfloor\right) \log\left(\frac{n}{2}\right) = \\ = \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) + \left(n - \left\lfloor \frac{n}{2} \right\rfloor\right)(\log(n) - 1) = \left(n - \left\lfloor \frac{n}{2} \right\rfloor\right) \log(n) + \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) - \left(n - \left\lfloor \frac{n}{2} \right\rfloor\right) \\ \geq \left(n - \left\lfloor \frac{n}{2} \right\rfloor\right) \log(n) \geq \left(n - \frac{n}{2}\right) \log(n) \geq \frac{1}{3} n \log(n) = \frac{1}{3} g(n).\end{aligned}$$

Damit erhalten wir $f(n) = \Omega(g(n))$ und $f(n) = \Theta(g(n))$, und somit $f(n) \neq o(g(n))$.

Hier noch einmal die ausgefüllte Tabelle:

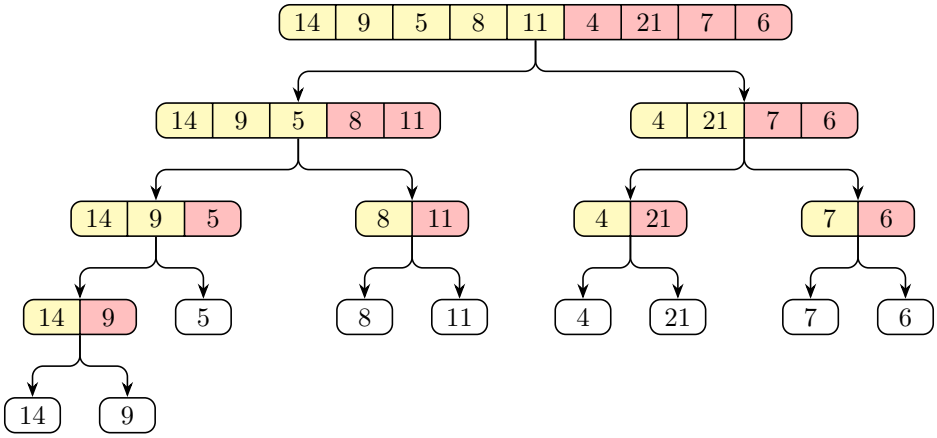
$f(n)$	$g(n)$	O	o	Ω	ω	Θ
$\log^k(n)$	n^ε	✓	✓	✗	✗	✗
n^k	c^n	✓	✓	✗	✗	✗
2^n	$2^{n/2}$	✗	✗	✓	✓	✗
$n^{\log(c)}$	$c^{\log(n)}$	✓	✗	✓	✗	✓
n^r	n^s	✓	✓	✗	✗	✗
$\log(n!)$	$\log(n^n)$	✓	✗	✓	✗	✓

□

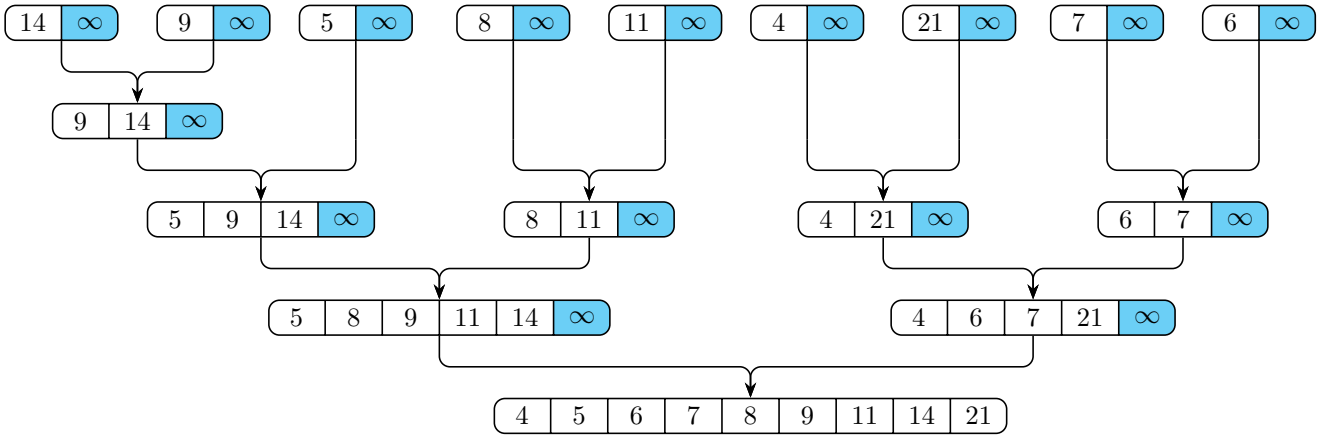
P4 (Darstellung von Mergesort)

Betrachten Sie das Array ganzer Zahlen $A = [14, 9, 5, 8, 11, 4, 21, 7, 6]$. Illustrieren Sie die Operationen von MERGESORT anhand von A.

Lösung. Der Algorithmus MERGESORT sortiert das Array A, indem er A in zwei ungefähr gleich große Arrays aufteilt, diese dann rekursiv via MERGESORT sortiert, und anschließend die zwei Ergebnisse mittels MERGE wieder zusammenführt. Die Rekursion bricht ab, sobald das zu sortierende Array die Länge 1 hat.
Durch rekursives Aufrufen von MERGESORT wird das Array A also wie folgt in Arrays der Länge 1 aufgeteilt:



Die jeweiligen Zwischenergebnisse werden nun durch wiederholte Anwendung von MERGE schrittweise wieder zusammengeführt, wie das folgende Bild zeigt. Die Grafik enthält auch die von MERGE angelegten "Sentinelwerte": Diese sind per Definition größer als jedes andere zu sortierende Element, und dienen nur dazu, den Code von MERGE zu vereinfachen.



□

P5 (Bubblesort)

BUBBLESORT ist ein sehr einfacher, aber ineffizienter Sortieralgorithmus. In dieser Aufgabe werden wir seine Korrektheit beweisen.

BUBBLESORT(A)

```
1 : n = A.length
2 : for i = 0 to n − 2 do
3 :     for j = n − 1 downto i + 1 do
4 :         if A[j] < A[j − 1] then
5 :             tmp = A[j]
6 :             A[j] = A[j − 1]
7 :             A[j − 1] = tmp
8 : return A
```

(a) Sei A' die Ausgabe von BUBBLESORT(A). Um zu beweisen, dass BUBBLESORT korrekt ist, müssen wir zeigen, dass der Algorithmus terminiert und dass

$$A'[0] \leq A'[1] \leq \cdots \leq A'[n-1] \tag{1}$$

gilt. Welche weiteren Eigenschaften müssen wir beweisen?

(b) Wir wollen nun mithilfe von Schleifeninvarianten zeigen, dass Ungleichung (1) gilt und dass BUBBLESORT terminiert. Stellen Sie dazu zunächst eine Schleifeninvariante für die Zeilen 2-7 auf, die die beiden Eigenschaften beweist.
(c) Stellen Sie nun eine weitere Schleifeninvariante für die Zeilen 3-7 auf und beweisen Sie diese.
(d) Nutzen Sie anschließend die Schleifeninvariante aus c), um die äußere Schleifenvariante von Zeile 2-7 beweisen zu können.

Lösung. (a) Wir müssen zusätzlich noch beweisen, dass A' genau die Elemente enthält, die auch A enthalten hat, also dass A' eine Permutation von A ist. Dies ist der Fall, da A nur in Zeile 5-7 verändert wird und dies immer eine Tauschoperation ist.

(b) Die Schleifeninvariante ist wie folgt: Beim Start jeder neuen Iteration der äußeren for-Schleife enthält $A[0..i-1]$ die i kleinsten Elemente in A in sortierter Reihenfolge.

Für den Fall nach dem letzten Schleifendurchlauf (also quasi der Fall $i = n-1$, auch wenn die Variable diesen Wert nie annehmen wird) wissen wir, dass $A[0..n-2]$ sortiert ist und dass $A[n-1]$ größer ist als alle Elemente in $A[0..n-2]$. Somit ist auch $A[0..n-1] = A$ sortiert. Da i sich in jedem Schleifendurchlauf um eins erhöht, wissen wir, dass wir diesen Zustand in endlich vielen Schritten erreichen (falls die innere Schleife terminiert) und wir somit die Terminierungseigenschaft der äußeren for-Schleife gezeigt haben.

(c) Für die innere for-Schleife stellen wir die folgende Schleifeninvariante auf: Am Anfang jedes Schleifendurchlaufs ist kein Element in $A[j+1..n-1]$ kleiner als $A[j]$.

Zunächst müssen wir die Initialisierungs-Eigenschaft überprüfen. Da vor der ersten Iteration des Loops $j = n-1$ gilt, ist damit die Menge $A[(n-1)+1..n-1]$ leer und die Schleifeninvariante gilt.

Für die Maintenance-Eigenschaft wissen wir, dass am Anfang jedes Schleifendurchlaufs kein Element in $A[j+1..n-1]$ kleiner als $A[j]$ ist und müssen zeigen, dass am Ende des Schleifendurchlaufs kein Element in $A[j..n-1]$ kleiner ist als $A[j-1]$. Dazu unterscheiden wir zwei Fälle: Falls $A[j-1] \leq A[j]$, verändert der Schleifendurchlauf nichts an A. Da schon kein Element in $A[j+1..n-1]$ kleiner ist als $A[j]$, ist auch keins kleiner als $A[j-1]$ und damit ist auch kein Element in $A[j..n-1]$ kleiner als $A[j-1]$ und die Maintenance-Eigenschaft damit für diesen Fall bewiesen. Im Fall $A[j-1] > A[j]$ tauscht der Schleifendurchlauf beide Elemente. Dadurch landen wir dann wieder im Fall $A[j-1] \leq A[j]$, für den wir die Maintenance-Eigenschaft schon bewiesen haben.

Zuletzt müssen wir noch die Terminierungseigenschaft überprüfen. Für $j = i+1$ gilt, dass nach dem Schleifendurchlauf kein Element in $A[i+1..n-1]$ kleiner ist als $A[i]$. Da sich j in jedem Schleifendurchlauf um eins verringert, landen wir in diesem Fall in endlich vielen Schritten.

(d) Die Initialisierungs-Eigenschaft der äußeren for-Schleife besagt, dass $A[0..-1]$ die kleinsten Elemente aus A enthält. Da $A[0..-1]$ leer ist, ist dies erfüllt.

Für die Maintenance-Eigenschaft wissen wir, dass am Anfang des Schleifendurchlaufs $A[0..i-1]$ die i kleinsten Elemente in sortierter Reihenfolge beinhaltet, und müssen zeigen, dass am Ende des Schleifendurchlaufs $A[0..i]$ die $i+1$ kleinsten Elemente in A in sortierter Reihenfolge beinhaltet. Da wir aus der Terminierungseigenschaft der inneren Schleife wissen, dass nach ihrer Ausführung kein Element in $A[i+1..n-1]$ kleiner ist als $A[i]$. Damit beinhaltet $A[0..i]$ die $i+1$ kleinsten Elemente in A. Da $A[i]$ zudem mindestens so groß ist wie alle Elemente in $A[0..i-1]$ ist $A[0..i]$ zudem auch noch sortiert.

Die Terminierungseigenschaft hatten wir in b) schon unter der Bedingung, dass die innere Schleife terminiert, bewiesen. Dies haben wir jetzt nachgeholt. Damit haben wir jetzt die Korrektheit von BUBBLESORT bewiesen. □

H1 (Sortieralgorithmen) (3+10+2 Punkte)

In dieser Übung sollen Sie lernen, Sortieralgorithmen selbst zu implementieren. **Bitte lesen Sie sich zunächst die allgemeinen Hinweise für die praktischen Übungen auf Moodle durch!** Laden Sie sich anschließend das vorgegebene Framework herunter, um die Aufgabe implementieren zu können.

In der Vorlesung haben Sie (unter anderem) QUICKSORT und INSERTIONSORT als Sortieralgorithmen kennengelernt. Wie Sie auch gelernt haben, ist QUICKSORT asymptotisch deutlich schneller als INSERTIONSORT im Durchschnitt. Jedoch gilt dies nicht für sehr kleine Datenmengen - durch die Komplexität von QUICKSORT ist hier INSERTIONSORT oft schneller.

In dieser Aufgabe sollen Sie ein hybrides Sortierverfahren programmieren, welches zunächst genauso funktioniert wie QUICKSORT, jedoch innerhalb von QUICKSORT auf INSERTIONSORT wechselt, sobald die Anzahl der Elemente, die in einem Unterschritt sortiert werden sollen, kleiner als k wird.

(a) Die Elemente, die Sie sortieren sollen, sind Karten eines verallgemeinerten Kartenspiels: Jede Karte hat eine Farbe, Hearts (Herz), Diamonds (Karo), Clubs (Kreuz) oder Spades (Pik), und einen Wert, der eine beliebige, ganze Zahl sein kann. Diese Datenstruktur ist in `lab/Card.java` vorgegeben. Implementieren Sie für diese Klasse die Funktion `compareTo`, die die aktuelle Instanz mit einer anderen Instanz vergleicht. Dabei wird zunächst nach Wert verglichen und falls dieser identisch ist, nach Farbe (wobei Diamonds < Hearts < Spades < Clubs gilt). `compareTo` soll -1 zurückgeben falls `this` echt kleiner ist als `other`, 1 falls `this` echt größer ist als `other` und 0 falls beide gleich groß sind.

(b) Implementieren Sie nun die Methode `HybridSort.sort` in der Datei `lab/HybridSort.java`, die eine Liste von Karten aufsteigend sortieren soll. Nuzten Sie für die Array-Zugriffe die Klasse `frame/SortArray.java`. Wir nutzen diese Klasse, um die Lese- und Schreibzugriffe auf das Array zu zählen - versuchen Sie auf keinen Fall, diese Klasse zu umgehen! Verwenden Sie für das Pivot-Element in QUICKSORT immer das erste Element im zu sortierenden Abschnitt des Arrays.

(c) Eine deterministische Wahl des Pivot-Elements (wie die Wahl des ersten, letzten oder mittleren Elements) kann immer dazu führen, dass QUICKSORT eine quadratische Laufzeit auf eine bestimmte Weise formatierten Eingaben hat. Unsere Wahl des ersten Elements (in b) führt beispielsweise bei weitgehend schon sortieren Arrays zu langen Laufzeiten. Implementieren Sie daher die von `HybridSort` abgeleitete Klasse `lab/HybridSortRandomPivot.java`, welche das Pivotelement zufällig wählt. Um möglichst wenig doppelten Code zu produzieren, sollten Sie Ihren Code in `lab/HybridSort.java` so ändern, dass die Wahl des Pivotelements in eine eigene Funktion ausgelagert ist, welche Sie dann in `HybridSortRandomPivot` überschreiben können.

Testen Sie ihre Implementierung! Wir haben Ihnen mit `lab/PublicTests.java` einige Tests vorgegeben, aber Sie können und sollten auch eigene Testfälle konstruieren. Wir haben Ihnen dazu in `lab/YourTests.java` ein Template vorgegeben, in dem Sie Ihre Tests implementieren können. Wie in den allgemeinen Hinweisen beschrieben, können Sie die Formatierung Ihrer Abgabe und ob diese unter openJDK 8 läuft auf unserem Testserver überprüfen.