Aufgabensammlung ADS-Repetitorium WS 25/26

Grundlagen – Analyse – Sortieren

Aufgabe 1: Zusammenhängende Mengen

Gegeben sei ein Feld A von positiven, natürlichen Zahlen. Das Feld habe n Elemente. Das Feld A heißt *zusammenhängend*, wenn es zwei Zahlen $m, l \in \mathbb{N}$ gibt, sodass $\{A[1], \ldots, A[n]\} = \{m, m+1, \ldots, m+l\}$. Zum Beispiel ist das Feld $\langle 10, 5, 6, 10, 8, 7, 8, 9\rangle$ zusammenhängend, wobei $\langle 10, 5, 6, 10, 8, 9\rangle$ nicht zusammenhängend ist, da die Zahl 7 fehlt. Geben Sie einen Algorithmus an, der in $\mathcal{O}(n)$ Zeit entscheidet, ob das gegebene Feld A zusammenhängend ist.

Lösung: Betrachte den folgendne Algorithmus. Sei max, min das größte bzw. das kleinste Element in A. Ein Feld A ist genau dann zusammenhängend, wenn $\{A[1], \ldots, A[n]\} = \{\mathsf{min}, \mathsf{min}+1, \ldots, \mathsf{max}\}$. Teste für jede natürliche Zahl in $[\mathsf{min}, \mathsf{max}]$, ob diese in A enthalten ist. Falls ja, gib *true* zurück, sonst *false*. Wir können den Test mithilfe eines zweiten Felds B durchführen. Die Länge des Feldes B ist $\mathsf{max} - \mathsf{min} + 1$ und $B[j]$ gibt an, ob ein $A[i]$ existiert, sodass $j = A[i] - \mathsf{max} + 1$. Das heißt, wenn am Ende des Algos ein $B[j]$ gibt, das *false* ist, kann A nicht zusammenhängend sein. Damit der Algorithmus tatsächlich in $\mathcal{O}(n)$ läuft, muss sichergestellt werden, dass $B.length \in \mathcal{O}(n)$. Falls $\mathsf{max} - \mathsf{min} + 1 > n$, so kann A nicht zusammenhängend sein, da dann $l > n$ ist aufgrund der vorherigen Beobachtung.

Algorithmus 1:

```
1  max, min = −∞, ∞
2  for i = 1 to n do
3  |  if A[i] < min then
4  |  |  min = A[i]
5  |  if A[i] > max then
6  |  |  max = A[i]
7  if max − min + 1 > n then
8  |  return false
9  bool[ ] B = new bool[max − min + 1]          // Standardwert ist false
10 for i = 1 to B.length do
11 |  B[A[i] − max +1] = true
12 for i = 1 to n do
13 |  if B[i] == false then
14 |  |  return false
15 return true
```

Aufgabe 2: Schleifeninvariante

Gegeben sei der folgende Algorithmus, der die Fakultät einer Zahl k berechnet.

(a) Geben Sie eine geeignete Invariante an, mit der wir zeigen können, dass $\mathsf{fakultät}$ für Eingaben ≥ 1 korrekt arbeitet.

Lösung: Vor jeder Ausführung des Schleifenkopfes in Zeile 2 hat f den Wert $k!/(j-1)!$.

(b) Zeigen Sie mit Hilfe der in (a) aufgestellten Invariante die Korrektheit des Algorithmus.

Algorithmus 2: int fakultät(int k)

```
1 if k = 0 then
2     return 1
3 f = j = k
4 while j > 1 do
5     j = j − 1
6     f = f · j
7 return f
```

Lösung:

Initialisierung Vor der ersten Iteration ($j = k$) hat f den Wert $k = k!/(k-1)! = k!/(j-1)!$.

Aufrechterhaltung Sei die Invariante korrekt zu Beginn einer Iteration. Zunächst wird $j = j - 1$ gesetzt, es gilt also nun $f = k!/j!$. Danach wird $f = f \cdot j$ berechnet, wonach wieder $f = k!/(j-1)!$ gilt. Damit ist die Schleifeninvariante auch weiterhin erfüllt.

Terminierung Die Schleife bricht ab, wenn $j \leq 1$. Da zu Beginn $j = k > 1$ war und j nur dekrementiert wurde, gilt also genau $j = 1$. Wegen der Schleifeninvariante gilt also $f = k!/(j-1)! = k!/(1-1)! = k!/0! = k!$.

Aufgabe 3: Induktionsbeweis eines Algorithmus

Betrachten sie folgenden Algorithmus zur Berechnung der Fakultät:

Algorithmus 3: int fakultät(int k)

```
1 f = j = k
2 while j > 1 do
3     j = j − 1
4     f = f · j
5 return f
```

(a) Geben Sie einen Algorithmus in Pseudocode an, der die Fakultät rekursiv berechnet.

Lösung:

Algorithmus 4: int fakultätRek(**int** k)

```
1 if k = 0 then
2     return 1
3 return k · fakultätRek(k − 1)
```

(b) Beweisen Sie mittels vollständiger Induktion die Korrektheit Ihrer rekursiven Variante.

Lösung: Wir beweisen die Korrektheit mittels vollständiger Induktion über k:

Induktionsanfang Für $k = 0$ ist $k! = 1$. Das gibt genau die **if**-Abfrage in Zeile 1 zurück.

Induktionsschritt Sei der Algorithmus korrekt für ein beliebiges $k - 1$. Wir zeigen, dass der Algorithmus dann auch für k korrekt ist. Wir wissen, dass $k! = k \cdot (k-1)!$. Genau dies

führt die Algorithmus in Zeile 3 durch. Da der Algorithmus für $k-1$ korrekt ist, ist auch die Berechnung für $k! = k \cdot (k-1)!$ korrekt.

Aufgabe 4: Laufzeiten von Pseudocodes bestimmen
Analysieren Sie die Algorithmen 19-21 bezüglich der Laufzeit. Geben Sie asymptotisch scharfe Schranken in Θ-Notation an!

Algorithmus 5: Algo1(**int**[] A)

```
1 k = 5
2 i = 0
3 while i ≤ A.length − k do
4 |   MergeSort(A, i + 1, i + k)
5 |   i = i + k
6 MergeSort(A, i + 1, A.length)
```

Algorithmus 6: Algo2(**int**[] A)

```
1 x = 0
2 i = 1
3 while i ≤ A.length do
4 |   x = x + A[i]
5 |   i = 2 · i
6 return x
```

Algorithmus 7: Algo1(**int** n)

```
1 total = 0
2 for i = 1 to n do
3 |   x = 0
4 |   for j = 1 to i do
5 |   |   x = x + 1
6 |   total = total + x
7 return total
```

Lösung:

- Algo1: MergeSort wird immer für ein Teilfeld der Länge höchstens k = 5 aufgerufen und somit ist die Laufzeit aller MergeSort Aufrufe konstant. Die While-Schleife in Zeile 2 wird $n/k = n/5$ mal aufgerufen. Das heißt ingesamt hat Algo1 eine Laufzeit von $\Theta(n)$
- Algo2: Pro Schleifendurchlauf haben wir konstanten Aufwand. Mit jedem Aufruf der Schleife wird i verdoppelt. Um die Anzahl der Schleifendurchläufe k zubestimmen, müssen wir herausfinden wann $2^k > n$ ist, wobei $n = A.length$. Dies ist der Fall, wenn $k > \log_2(n)$. Ingesamt läuft Algo2 also in $\Theta(\log n)$ Zeit.
- Algo3: Der Aufwand des Schleifenkörpers der inneren Schleife in Zeile 4-5 ist konstant. Demnach können wir die Gesamtlaufzeit von Algo3 mit folgender Gleichung ausdrücken:

$$\sum_{i=1}^{n}\left(\Theta(1)+\sum_{j=1}^{i}\Theta(1)\right)=\Theta(n)+\sum_{i=1}^{n}\Theta(i)=\Theta(n)+\Theta(n^2)=\Theta(n^2)$$

Aufgabe 5: Asymptotisches Wachstum von Funktionen
Ordnen Sie die Liste von Funktionen nach ihrem asymptotischen Wachstum ($\mathcal{O}(\dots) \subsetneq \mathcal{O}(\dots) \cdots \subsetneq \mathcal{O}(\dots)$). Nutzen Sie $=$ für das gleiche asymptotische Wachstum und $\subsetneq$ für unterschiedliches Wachstum. Beispiel: $f(n) = n$, $g(n) = 2n$, $h(n) = n^2$ dann gilt: $\mathcal{O}(f(n)) = \mathcal{O}(g(n)) \subsetneq \mathcal{O}(h(n))$.

$$\sqrt{n}\log_4(n^2),\ 5\sqrt{n}, 2^n,\ \log_2(n),\ n^2-7n,\ n\log_{10}(n/3),\ n^{\log_3(4)},\ \log_4(n^3),\ \sqrt{n}\log_2(n^{\sqrt{n}}),$$
$$4^{\log_3 n},\ 2^{2n},\ (n+1)^2,\ n(\log_2(n))^2,\ n!,\ 2^{3n\log_2(n)}$$

Lösung:

- $\sqrt{n}\log(n^2) = 2\sqrt{n}\log(n)$
- $n\log(n/3) = n\log(n) - n\log(3)$
- $\log_4(n^3) = 3\log_4(n)$
- $\sqrt{n}\log(n^{\sqrt{n}}) = n\log(n)$
- $4^{\log_3(n)} = n^{\log_3(4)}$
- $n^{\log_3(4)} \approx n^{1,26}$
- $\mathcal{O}(n!) \subset \mathcal{O}(n^n) = \mathcal{O}(2^{n\log_2 n}) \subset \mathcal{O}(8^{n\log_2 n})$
- Basiswechsel für Logarithmen

$$\mathcal{O}(\log_2(n)) = \mathcal{O}(\log_4(n^3)) \subsetneq \mathcal{O}(5\sqrt{n}) \subsetneq \mathcal{O}(\sqrt{n}\log_4(n^2)) \subsetneq \mathcal{O}(\sqrt{n}\log(n^{\sqrt{n}})) = \mathcal{O}(n\log_{10}(n/3))$$
$$\subsetneq \mathcal{O}(n\log_2^2(n)) \subsetneq \mathcal{O}(n^{\log_3(4)}) = \mathcal{O}(4^{\log_3(n)}) \subsetneq \mathcal{O}(n^2 - 7n) = \mathcal{O}((n+1)^2)$$
$$\subsetneq \mathcal{O}(2^n) \subsetneq \mathcal{O}(2^{2n}) \subsetneq \mathcal{O}(n!) \subsetneq \mathcal{O}(2^{3n\log_2(n)})$$

Aufgabe 6: Sortieralgorithmen

(a) Sortieren Sie das Feld A = [4, 3, 7, 2, 0, 9 ,8, 1, 5, 6] mit InsertionSort. Geben Sie nach jeder Iteration der äußeren Schleife das Feld an.

Lösung:

1. A = [3, 4, 7, 2, 0, 9 ,8, 1, 5, 6] – Die 3 wird eingeordnet
2. A = [3, 4, 7, 2, 0, 9 ,8, 1, 5, 6] – Die 4 ist bereits richtig
3. A = [3, 4, 7, 2, 0, 9 ,8, 1, 5, 6] – Die 7 ist bereits richtig
4. A = [2, 3, 4, 7, 0, 9 ,8, 1, 5, 6] – Die 2 wird eingeordnet
5. A = [0, 2, 3, 4, 7, 9 ,8, 1, 5, 6] – Die 0 wird eingeordnet
6. A = [0, 2, 3, 4, 7, 9 ,8, 1, 5, 6] – Die 9 ist bereits richtig
7. A = [0, 2, 3, 4, 7, 8, 9, 1, 5, 6] – Die 8 wird eingeordnet
8. A = [0, 1, 2, 3, 4, 7, 8, 9, 5, 6] – Die 1 wird eingeordnet
9. A = [0, 1, 2, 3, 4, 5, 7, 8, 9, 6] – Die 5 wird eingeordnet
10. A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] – Die 6 wird eingeordnet

(b) Sortieren Sie das Feld B = [3, 7, 2, 9, 1, 4, 6, 5, 8, 0] mit HeapSort. Geben Sie bei jedem Schritt den entstandenen Heap an.

Lösung: Zuerst müssen wir das Feld in einen Heap umbauen. Die Schritte dafür sind:

1. B = [3, 7, 2, 9, 1, 4, 6, 5, 8, 0] – Heap [1, 0] ist bereits ein Heap.
2. B = [3, 7, 2, 9, 1, 4, 6, 5, 8, 0] – Heap [9, 5, 8] ist bereits ein Heap.
3. B = [3, 7, 6, 9, 1, 4, 2, 5, 8, 0] – Heap [6, 4, 2] wird gebildet.
4. B = [3, 9, 6, 7, 1, 4, 2, 5, 8, 0] – Heap [9, 7, 1] wird gebildet.

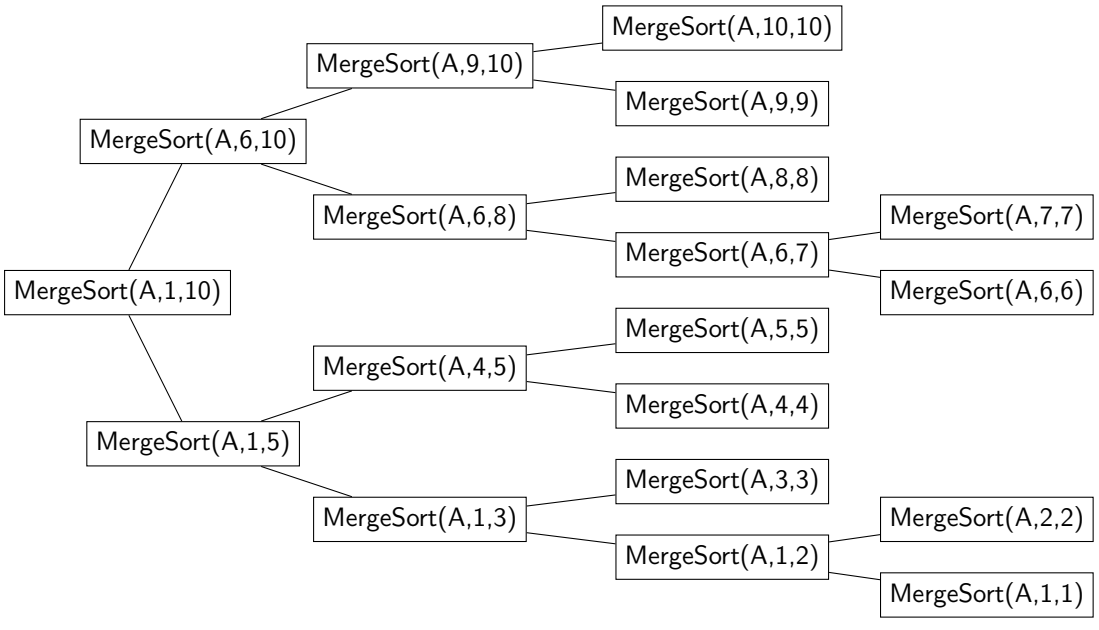
5. B = [3, 9, 6, 8, 1, 4, 2, 5, 7, 0] – 7 versickert.
6. B = [9, 3, 6, 8, 1, 4, 2, 5, 7, 0] – Heap [9, 3, 6] wird gebildet.
7. B = [9, 8, 6, 3, 1, 4, 2, 5, 7, 0] – 3 versickert.
8. B = [9, 8, 6, 7, 1, 4, 2, 5, 3, 0] – 3 versickert.

Nun zur Sortierung:

1. B = [8, 7, 6, 5, 1, 4, 2, 0, 3 | 9] – 0 mit 9 tauschen und versickern
2. B = [7, 5, 6, 3, 1, 4, 2, 0 | 8, 9] – 3 mit 8 tauschen und versickern
3. B = [6, 5, 4, 3, 1, 0, 2 | 7, 8, 9] – 0 mit 7 tauschen und versickern
4. B = [5, 3, 4, 2, 1, 0 | 6, 7, 8, 9] – 2 mit 6 tauschen und versickern
5. B = [4, 3, 0, 2, 1 | 5, 6, 7, 8, 9] – 0 mit 5 tauschen und versickern
6. B = [3, 2, 0, 1 | 4, 5, 6, 7, 8, 9] – 1 mit 4 tauschen und versickern
7. B = [2, 1, 0 | 3, 4, 5, 6, 7, 8, 9] – 1 mit 3 tauschen und versickern
8. B = [1, 0 | 2, 3, 4, 5, 6, 7, 8, 9] – 0 mit 2 tauschen und versickern
9. B = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] – 1 mit 0 tauschen und versickern

(c) MergeSort arbeitet rekursiv. Geben Sie für das Feld C = [9, 4, 1, 3, 5, 2, 6, 0, 8, 7] den Rekursionsbaum von MergeSort an. In jedem Knoten soll der jeweilige Aufruf von MergeSort und die zu sortierende Teilliste stehen, jeweils *vor* der Sortierung.

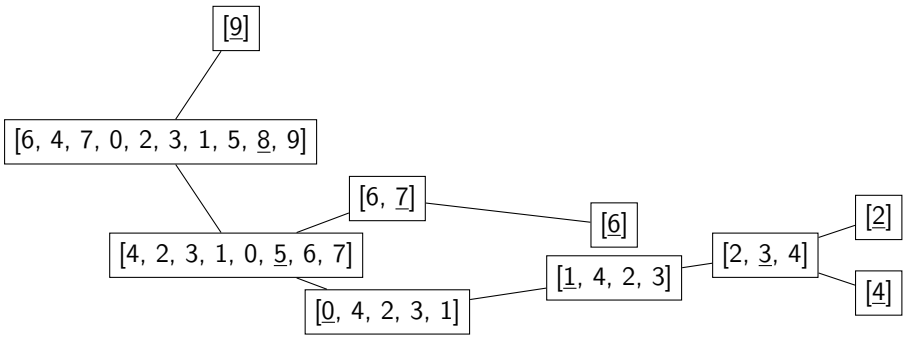
Lösung:



(d) Sortieren Sie das Feld D = [6, 4, 7, 9, 2, 3, 1, 5, 0, 8] mit einem vereinfachten QuickSort. Dieser schreibt alle Elemente, die kleiner als das Pivotelement sind, links und alle größeren rechts neben das

Pivotelement. Zeichnen Sie den Rekursionsbaum. Schreiben Sie in jeden Knoten das zu sortierende Teilfeld nach dem Aufruf von Partition und markieren Sie das Pivotelement, die Wurzel sieht also so aus: [6, 4, 7, 0, 2, 3, 1, 5, 8, 9]. *Achtung: Das ist nicht der VL-Algorithmus, aber die Idee ist die gleiche!*

Lösung:



Aufgabe 7: Pseudocode – Spot the Error

Die folgenden Algorithmen berechnen nicht das, was sie sollen. Erklären Sie, was der Fehler ist und schreiben Sie den richtigen Algorithmus auf. Geben Sie auch die asymptotische Worst-Case-Laufzeit in Θ-Notation an.

(a) Der Algorithmus soll $f(n) = n$ berechnen.

Algorithmus 8: int Algo1(int n)

```
1 zähler = 0
2 for i = 1 to n do
3 |  return zähler +1
```

Lösung: return beendet den Algorithmus im ersten Schleifendurchlauf.

(b) Der Algorithmus soll $f(n) = \sum_{i=0}^{n} i$ berechnen

Algorithmus 9: int Algo2(int n)

```
1 zähler = 0
2 for i = 1 to n do
3 |  zähler+1
4 return zähler
```

Lösung: Der Zähler wird nicht verändert.

(c) Der Algorithmus soll $f(n) = n!$ berechnen.

Algorithmus 10: int Algo3(int n)

```
1 return Algo3(n − 1) · n
```

Lösung: Der Basisfall fehlt.

(d) Der Algorithmus soll **true** zurückgeben, wenn i im Array A enthalten ist, sonst **false**.

Algorithmus 11: boolean Algo4(int i, int[] A, int $l = 0$)

```
1 if A.length == l then
2 |  return false
3 else
4 |  return (i == A[l]) or Algo4(i, l + 1)
```

Lösung: Das Array wird nicht übergeben.

Aufgabe 8: Vereinigung

Geben Sie in gut kommentiertem Pseudocode einen Algorithmus an, der als Eingabe zwei aufsteigend sortierte Felder A und B erhält. die Ausgabe soll ein Feld C sein, das jede Zahl aus A und B genau einmal enthält. Die Laufzeit soll $O(n)$ sein, wobei n = A.length + B.length.

Lösung: Wie Merge von MergeSort, wobei Vielfache entweder gleich ignoriert, oder herausgefiltert werden, wenn das Hilfsfeld in C kopiert wird.

Aufgabe 9: Ähnliche Zahlen

Sei A ein Feld der Länge $n > 1$ von zufälligen Zahlen, wobei Zahlen mehrfach vorkommen dürfen.

(a) Geben Sie einen Algorithmus in Pseudocode an, der zwei Zahlen A[i] und A[j] mit $i \neq j$ sucht, sodass $|A[i]-A[j]|$ minimal ist. Der Algorithmus soll die Indizes beider Zahlen ausgeben und $\Theta(n^2)$ Zeit benötigen.

Lösung:

Algorithmus 12: findClosestPair(int[ ] A)

```
1  n = A.length
2  minI = 1
3  minJ = 2
4  min = ∞
5  for i = 1 to n − 1 do
6      for j = i + 1 to n do
7          abs = |A[i] − A[j]|
8          if abs < min then
9              min = abs
10             minI = i
11             minJ = j
12 return (minI,minJ)
```

(b) Begründen Sie die Korrektheit Ihres Algorithmus, indem Sie die Korrektheit der inneren Schleife mit einer Invariante zeigen.

Lösung: Wir geben eine Schleifeninvariante für die innere Schleife bei einem festen i an: „Vor der k. Ausführung enthält min den minimalen Abstand eines Tupels in der Menge $\{(s,t) \in \mathbb{N}^2 \mid (s = i \Rightarrow t < i + k) \wedge (s < i \Rightarrow t < n + 1) \wedge (0 < s \leq i < t \leq n)\}$ “.

Initialisierung Vor der 1. Iteration der inneren **for**-Schleife kann min nur den minimalen Abstand eines Tupels (s,t) mit $s < i$ beinhalten, da min noch gar nicht angefasst wurde.

Aufrechterhaltung Gelte die Invariante vor der k. Iteration, also enthalte min das Minimum obiger Menge. In der k. Iteration ist $j = i + k$. Falls nun $|A[i] - A[j]|$ kleiner als das bisherige Minimum war, wird es ausgetauscht, andernfalls passiert nichts. Vor der $k + 1$. Iteration ist also die Invariante wieder korrekt.

Terminierung Bei Beendigung der **for**-Schleife sind $n-i$ Iterationen vergangen, also $k = n-i+1$. Setzen wir diesen Wert in die obige Menge ein, fallen die ersten beiden Terme zusammen und min enthält folglich das Minimum der Tupel in $\{(s,t) \in \mathbb{N}^2 \mid (t < n + 1) \wedge (0 < s \leq i < t \leq n)\}$.

Die Korrekheit der äußeren **for**-Schleife lässt sich analog zeigen; sie folgt sofort aus der Korrektheit der inneren **for**-Schleife.

Aufgabe 10: SelectionSort
Gegeben sei der folgende Sortieralgorithmus.

Algorithmus 13: SelectionSort(int[] A)

```
1 n = A.length
2 for i = 1 to n − 1 do
3 |   ℓ = i
4 |   for j = i to n do
5 |   |   if A[j] < A[ℓ] then
6 |   |   |   ℓ = j
7 |   Swap(A, i, ℓ)
8 return A
```

(a) Welche Laufzeit hat SelectionSort jeweils im besten und im schlechtesten Fall?

Lösung: Die innere Schleife benötigt $n - i + 1$ Schritte mit jeweils konstantem Aufwand. Dadurch erhalten wir über alle Iterationen eine Laufzeit von

$$\sum_{i=1}^{n-1} n - i + 1 = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 \in \Theta(n^2)$$

(b) Geben Sie eine geeignete Invariante an, um die Korrektheit von SelectionSort zu beweisen.

Lösung: Zu Beginn der $i$ten Iteration

(i) enthält $A[1..i-1]$ die $i-1$ kleinsten Zahlen aus A aufsteigend sortiert und

(ii) A enthält noch die gleichen Zahlen wie zu Beginn.

(c) Beweisen Sie die Korrektheit von SelectionSort. Für die innere Schleife muss kein Korrektheitsbeweis angegeben werden, es ist ausreichend zu beschreiben, was die Schleife berechnet.

Lösung:

Initialisierung Vor der ersten Iteration ($i = 1$) wurde A noch nicht verändert. Daher ist (ii) erfüllt. Außerdem enthält $A[1..0]$ trivialerweise die kleinsten 0 Elemente aus A.

Aufrechterhaltung Seien (i) und (ii) zu Beginn der $i$ten Iteration erfüllt. Innerhalb der Iteration wird A nur durch einen Tauschbefehl verändert. Somit bleibt (ii) auch nach der Iteration erhalten. Die innere Schleife berechnet den Index des kleinsten Elements aus $A[i..n]$ und tauscht dieses Minimum an die Stelle $A[i]$. Wegen (i) ist $A[i]$ mindestens so groß wie die Zahlen in $A[1..i-1]$, daher enthält $A[1..i]$ nun die i kleinsten Zahlen aufsteigend sortiert. Somit bleibt auch (i) erhalten.

Terminierung Der Algorithmus terminiert für $i = n$. Aus (ii) folgt direkt, dass A noch die gleichen Zahlen wie zu Beginn enthält. Dabei ist wegen (i) das Teilfeld $A[1..n-1]$ bereits korrekt sortiert. Die Zahl in $A[n]$ muss wegen (i) mindestens so groß wie $A[n-1]$ sein, folglich ist A korrekt aufsteigend sortiert.

Aufgabe 11: Vollständige Induktion

Zeigen Sie die folgenden Aussagen mittels vollständiger Induktion.

(a) Für jede natürliche Zahl n ist 3 ein Teiler von $n^3 - n$.

Lösung: Wir beweisen die Aussage mit vollständiger Induktion über n:

Induktionsanfang Sei $n = 1$. Dann ist $n^3 = 1$ und $n^3 - n = 0$. Die Zahl 3 ist tatsächlich ein Teiler von 0.

Induktionsschritt Sei die Aussage richtig für beliebiges, festes n. Wir zeigen, dass sie auch für $n + 1$ richtig ist.

$$\begin{aligned}(n+1)^3 - (n+1) &= n^3 + n^2 + 2n^2 + 2n + n + 1 - (n+1)\\ &= (n^3 - n) + 3n^2 + 3n\\ &= (n^3 - n) + 3(n^2 + n)\end{aligned}$$

Laut Induktionsannahme ist $n^3 - n$ durch 3 teilbar. Nun addieren wir zu einer durch 3 teilbaren Zahl ein Vielfaches von 3. Folglich ist die Summe ebenfalls durch 3 teilbar.

(b) Zeigen Sie, dass für alle $n \in \mathbb{N}$ gilt: $1 + 3 + \cdots + (2n - 1) = n^2$

Lösung:

Induktionsanfang Sei $n = 1$. Dann gilt $2 \cdot 1 - 1 = 1 = 1^2$.

Induktionsschritt . Sei die Aussage richtig für ein beliebiges aber festes n. Wir zeigen, dass sie auch für $n + 1$ richtig ist.

$$\begin{aligned}1 + 3 + \cdots + (2(n+1) - 1) &= 1 + 3 + \cdots + (2n - 1) + (2(n+1) - 1)\\ &= n^2 + 2(n+1) - 1 = n^2 + 2n + 1\\ &= (n+1)^2\end{aligned}$$

(c) Die Fibonacci-Folge ist eine rekursiv definierte Zahlenfolge. Dabei ist $F(0) = 0$ und $F(1) = 1$. Die n-te Fibonacci-Zahl für ein $n > 1$ ist dann $F(n-1) + F(n-2)$. Die Berechnungsvorschrift dauert für große n jedoch sehr lange. Mit der Formel von Moivre-Binet kann die n-te Fibonacci-Zahl direkt ausgerechnet werden. Beweisen Sie die Richtigkeit der Formel:

$$F(n) = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}$$

Lösung: Sei $a = (1 + \sqrt{5})/2$ und $b = (1 - \sqrt{5})/2$. Wir zeigen die Korrektheit der Aussage durch eine Induktion über n.

Induktionsanfang Sei $n' = 2$. Dann ist $F(n) = F(0) + F(1) = 1$. Setzen wir $n' = 2$ direkt in die obige Gleichung ein, erhalten wir ebenfalls 1. Da die Fibonacci-Zahlen auf den jeweils zwei vorherigen Folgengliedern aufbauen, müssen wir auch $n' = 3$ testen: $F(n) = F(2) + F(1) = 3$, was mit dem Ergebnis der direkten Gleichung übereinstimmt.

Induktionschritt Sei die Aussage richtig für $n - 1$ und $n - 2$. Wir zeigen, dass die Aussage dann

auch für n richtig ist:

$$\begin{aligned} F(n) = F(n-1) + F(n-2) &\underbrace{=}_{\text{IA}} \frac{a^{n-1} - b^{n-1}}{\sqrt{5}} + \frac{a^{n-2} - b^{n-2}}{\sqrt{5}} \\ &= \frac{a^{n-1} - b^{n-1} + a^{n-2} - b^{n-2}}{\sqrt{5}} \\ &= \frac{a^{n-1}(1 + \frac{1}{a}) - b^{n-1}(1 + \frac{1}{b})}{\sqrt{5}} \end{aligned}$$

Es wäre schön, wenn $1 + 1/a = a$. Also überprüfen wir das:

$$1 + \frac{1}{a} = a \Rightarrow a + 1 = a^2 \Rightarrow a^2 - a - 1 = 0 \Rightarrow a = \frac{1 \pm \sqrt{1 - 4 \cdot (-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

Wir setzen $a = 1 + 1/a$ oben ein und erhalten das gewünschte Ergebnis:

$$\frac{a^{n-1}a - b^{n-1}b}{\sqrt{5}} = \frac{a^n - b^n}{\sqrt{5}} = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}$$

(d) Auf einem quadratischen Schachbrett mit einer Seitenlänge von mehr als drei Feldern kann der Springer jedes Feld von jedem anderen Feld erreichen. Dafür hat er beliebig viele Züge zur Verfügung.

Lösung: Wir zeigen die Aussage durch Induktion über die Seitenlänge n in Feldern:

Induktionsanfang Für $n = 4$ finden wir durch Versuchen heraus, dass die Aussage stimmt.

Induktionsschritt Sei die Aussage für $n-1$ bewiesen. Wir zeigen die Richtigkeit der Aussage für n. Dank der Induktionsannahme wissen wir, dass sich der Springer im Teilbrett $n-1 \times n-1$ überallhin bewegen kann. Betrachten wir das Brett $n \times n$. Alle Randfelder sind durch nur einen „Springersprung “ von einem Mittelfeld erreichbar. Jedes Mittelfeld ist aber wegen der Induktionsannahme ebenfalls erreichbar. Deshalb kann der Springer auch im $n \times n$ Schachbrett mit beliebig vielen Sprüngen auf alle Felder springen.

Aufgabe 12: Korrektheitsbeweise und Rekursion
Betrachten Sie den folgenden Algorithmus.

Algorithmus 14: int doSomethingSimple(**int** A[ ], **int** $i = 1$)

```
Data: Feld mit natürlichen Zahlen A, natürliche Zahl i
Result: Ein Wert, der mit A zusammenhängt
1 if i == A.length then
2     return A[i]
3 k = doSomethingSimple(A, i + 1);
4 if k > A[i] then
5     return k
6 else
7     return A[i]
```

(a) Beschreiben Sie in einem Satz, was der Algorithmus macht.

Lösung: Der Algorithmus findet das Maximum des Feldes A.

(b) Beweisen Sie die Korrektheit des Algorithmus.

Lösung:

Beweis. Wir beweisen die Korrektheit per Induktion über $n = \mathsf{A.length} - i + 1$.

Induktionsanfang Sei $n = 1$. Dann ist $\mathsf{A.length} = i$ und die **If**-Abfrage in Zeile 1 wird mit *wahr* ausgewertet. Es wird $\mathsf{A}[i]$ zurückgegeben, was das Maximum des ein-elementigen Feldes A[A.length..A.length] ist.

Induktionsschritt Angenommen, der Algorithmus ist korrekt für n. Wir zeigen, dass er auch für $n + 1$ korrekt ist. In Zeile 3 wird doSomethingSimple für ein $i + 1$ aufgerufen, das heißt, für diesen Aufruf ist $n' = \mathsf{A.length} - (i + 1) + 1 = \mathsf{A.length} - i < n$. Nach Induktionsannahme liefert Zeile 3 folglich das Maximum aus dem Teilfeld $\mathsf{A}[i + 1..\text{A-length}]$. In der nun folgenden **If**-Abfrage wird das Maximum aus $\mathsf{A}[i + 1..\text{A-length}]$ und $\mathsf{A}[i]$ ausgewählt. Damit liefert der Algorithmus insgesamt das Maximum aus $\mathsf{A}[i..\text{A-length}]$.

□

(c) Geben Sie einen Algorithmus an, der äquivalent zu doSomethingSimple ist, ohne Rekursion zu verwenden.

Lösung:

Algorithmus 15: int findMax(int[ ] A)

```
1 if A.length == 0 then
2     return ∅
3 max = A[1]
4 for i = 2 to A.length do
5     if A[i] > max then max = A[i]
```

(d) Geben Sie eine Schleifeninvariante für Ihren inkrementellen Algorithmus an.

Lösung: Vor der j. Iteration enthält max immer das Maximum des Teilfelds A[1..j].

(e) Beweisen Sie die Korrektheit Ihres Algorithmus mit der von Ihnen aufgestellten Schleifeninvariante.

Lösung:

Intialisierung Unmittelbar vor der ersten ($j = 1$) Iteration der **for**-Schleife steht in max der Wert von A[i]. Damit ist max das Maximum aus A[1..1], und die Invariante ist erfüllt.

Aufrechterhaltung Es gelte die Schleifeninvariante vor der j Ausführung, das heißt, max enthält das Maximum aus A[1..j]. In der j. Iteration ist i=$j + 1$ und der Wert von max wird in der **if**-Abfrage mit A[i] ausgetauscht, falls A[i] größer als max ist. Nach Auswertung von Zeile 5 enthält max das Maximum aus A[1..i]. Da $j = i+1$, enthält max das Maximum aus A[1..$j+1$], womit die Schleifeninvariante vor der $j + 1$. Iteration erfüllt ist.

Terminierung Die Schleife wird A.length-1 mal durchlaufen. Damit gilt nach der letzten Iteration wegen der Schleifeninvariante, dass max das Maximum aus A[1..A.length$-1+1$] enthält.

Aufgabe 13: Algorithmen und Laufzeiten

(a) Was berechnet der Algorithmus?
Wie viele Vergleiche, Additionen und Multiplikationen werden in Abhängigkeit von n ausgeführt?

Algorithmus 16: SomeAlgo(n)

```
1 int j = 0; int s = 1; int S = 0
2 while j < n do
3 |   S = S + s
4 |   j = j + 1
5 |   s = s · 2
6 return S
```

Lösung: Die Funktion berechnet: $1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$
Insgesamt gibt es $n + 1$ Vergleiche von j und n (n positive Vergleiche und 1 Vergleich, der die Schleife abbricht) sowie n Multiplikationen von s mit 2, sowie $2n$ Additionen, je Durchlauf eine für j und eine für S.

(b) Sei folgender Algorithmus zur Berechnung des Produkts $i \cdot (i+1) \cdot \ldots \cdot (j-1) \cdot j$ für natürliche Zahlen i und j mit $i < j$ gegeben:

Algorithmus 17: int Produkt(int j, int i)

```
1 return Fakultaet(j)/Fakultaet(i − 1)
```

Algorithmus 18: int Fakulaet(int x)

```
1 if x == 0 then
2 |   return 1
3 return x · Fakultaet(x − 1)
```

Begründen Sie kurz, warum der Algorithmus Produkt korrekt ist. Geben Sie die Worst-Case-Laufzeit von Produkt in Abhängigkeit von i und j an.

Lösung: Methode Produkt ist korrekt, da $\frac{j!}{(i-1)!} = \frac{1 \cdot 2 \cdot \ldots \cdot (i-1) \cdot i \cdot (i+1) \cdot \ldots \cdot j}{1 \cdot 2 \cdot \ldots \cdot (i-1)} = i \cdot (i+1) \cdot \ldots \cdot j$.
Die Worst-Case-Laufzeit von Produkt ist $\Theta(j)$.

Aufgabe 14: Aufwandsanalyse

Gegeben seien die Funktionen $f \in \Theta(\log n)$ und $g(n) \in \Theta(n)$. Bestimmen Sie die Laufzeit der folgenden Programmfragmente in der Θ-Notation.

(a)

Algorithmus 19:

```
1 for i = 1 to n do
2     g(n)
```

Lösung: $\Theta(n^2)$, da $g(n)$ n-mal aufgerufen wird.

(b)

Algorithmus 20:

```
1 i = 1
2 while i < n do
3     f(n)
4     g(n)
5     i = i · 2
```

Lösung: $\Theta(n \log n)$. i wird in jedem Schleifendurchlauf verdoppelt, deswegen wird die while-Schleife $\Theta(\log_2 n)$-mal ausgeführt. Innerhalb der Schleife ist der Aufwand $\Theta(\log n)+\Theta(n) = \Theta(n)$. Insgesamt ist der Aufwand also:

$$\Theta(\log_2 n) \cdot \Theta(n) = \Theta(n \log n)$$

(c)

Algorithmus 21:

```
1 i = n
2 while i > 0 do
3     f(n)
4     i = i/2
```

Lösung: $\Theta(\log^2 n)$. Die Schleife wird $\Theta(\log_2 n)$-mal ausgeführt. Also ist der Aufwand:

$$\Theta(\log_2 n) \cdot \Theta(\log n) = \Theta(\log^2 n)$$

(d)

Algorithmus 22:

```
1 for i = 1 to n do
2     j = n
3     while j ≥ 1 do
4         for k = 1 to n do
5             g(n)
6         j = j/2
```

Lösung: $\Theta(n^3 \log n)$. Die innere Schleife wird n-mal betreten, hat also einen Gesamtaufwand von $\Theta(n^2)$. Die while Schleife wird $\Theta(\log n)$-mal betreten, hat also einen Gesamtaufwand von $\Theta(n^2 \log n)$. Die äußere for-Schleife wird n-mal betreten, also ist der Gesamtaufwand $\Theta(n^3 \log n)$.

Aufgabe 15: $\mathcal{O}$-, Θ- und Ω-Notation

Beweisen oder widerlegen Sie die Behauptungen. Arbeiten Sie mit der Definition aus der Vorlesung.

(a) $f(n) = \frac{1}{2}n - 2 \in \Omega(\log_2 n)$

Lösung: Die Aussage ist *wahr*. Dementsprechend ist zu zeigen: $\exists n_0 \exists c \forall n \geq n_0 \colon c \cdot \log_2 n \leq \frac{1}{2}n - 2$
Man wähle $c = 1/2$, dann gilt:

$$\frac{1}{2}\log_2 n \leq \frac{1}{2}n - 2 \Leftrightarrow \log_2 n + 4 \leq n$$

Also wähle $n_0 = 8$, da $\log_2 n \leq n$ für alle $n > 0$.

(b) $f(n) = n^n + n^2 \in O(n^{n-1})$

Lösung: Die Aussage ist *falsch*. Dementsprechend ist zu zeigen: $\forall c \forall n_0 \exists n \geq n_0 \colon c \cdot n^{n-1} < f(n)$
Seien n_0 und c beliebig. Wir beobachten zunächst $c \cdot n^{n-1} < n^n \Longrightarrow c \cdot n^{n-1} < f(n)$.

$$\begin{aligned} & c \cdot n^{n-1} < n^n \\ \Longleftrightarrow\ & c < \frac{n^n}{n^{n-1}} = n \end{aligned}$$

Damit wählen wir $n = \max(n_0, c+1)$. Für diese Wahl gilt $n > c$ und somit auch $c \cdot n^{n-1} < n^n$, aus der Beobachtung folgt die gewünschte Aussage.

(c) $f(n) = \frac{n^4 - 4n^2}{2n+7} \notin O(n^3)$

Lösung: Die Aussage ist *falsch*. Dementsprechend ist zu zeigen: $\exists n_0 \exists c \forall n \geq n_0 \colon \frac{n^4-4n^2}{2n+7} \leq c \cdot n^3$
Wir vereinfachen die Ungleichung:

$$\begin{aligned} \frac{n^4 - 4n^2}{2n+7} &\leq c \cdot n^3 \\ n^4 - 4n^2 &\leq c \cdot n^3 \cdot (2n+7) \\ n^4 - 4n^2 &\leq c \cdot (2n^4 + 7n^3) \end{aligned}$$

Mit der letzten Ungleichung wählen wir $n_0 = 1$ und $c = 1$.

(d) $f(n) = \log_3(n^5 \cdot 9^{n^2}) \in \Omega(n \log_3 n)$

Lösung: Die Aussage ist *wahr*. Also ist zu zeigen: $\exists n_0 \exists c \forall n \geq n_0 \colon c \cdot n \log_3 n \leq \log_3(n^5 \cdot 9^{n^2})$. Wir zeigen dies:

$$c \cdot n \log_3 n \leq \log_3(n^5 \cdot 9^{n^2}) = \log_3 n^5 + \log_3 9^{n^2} = 5\log_3 n + n^2 \log_3 9 = 5\log_3 n + 2n^2$$

$$c \cdot n \log_3 n \underbrace{\leq}_{\text{Wir müssen dieses zeigen}} 2n^2 \leq 5\log_3 n + 2n^2$$

$$c \cdot \log_3 n \leq 2n$$

Also können wir $c = 1$ wählen und $n_0 = 0$, da $\log_3 n \leq 2n$ für alle n.

(e) $f(n) = \log_a n \in \Theta(\log_b n)$ für beliebige $a, b > 1$

Lösung: Die Aussage ist *wahr*. Also ist zu zeigen: $\exists n_0 \exists c_1 \forall n \geq n_0 \colon c_1 \cdot \log_b n \leq \log_a n$ und $\exists n_0 \exists c_2 \forall n \geq n_0 \colon \log_a n \leq c_2 \cdot \log_b n$. Die Auflösung der Ungleichung liefert auch gleich die Werte für c_1 und c_2, unabhängig von n, also können wir $n_0 = 0$ wählen.

$$\begin{aligned} c_1 \log_b n \leq & \log_a n \leq c_2 \log_b n \\ c_1 \leq & \frac{\log_a n}{\log_b n} \leq c_2 \\ c_1 \leq & \frac{\log_b n}{\log_b a \cdot \log_b n} \leq c_2 \\ c_1 \leq & \frac{1}{\log_b a} \leq c_2 \end{aligned}$$

(f) $f(n) = \frac{1}{100} n^2 + n \sin n \in \Theta(n^2)$

Lösung: Die Aussage ist *wahr*. Wir zeigen zunächst $f(n) \in \Omega(n^2)$, dafür ist folgendes zu zeigen: $\exists n_1 \exists c_1 \forall n \geq n_1 \colon c_1 \cdot n^2 \leq \frac{1}{100} n^2 + n \sin n$:

$$\begin{aligned} & c_1 \cdot n^2 \leq \frac{1}{100} n^2 + n \sin n \\ & c_1 \cdot n^2 \underbrace{\leq}_{\text{Das zeigen wir}} \frac{1}{100} n^2 - n \leq \frac{1}{100} n^2 + n \sin n \\ & c_1 \cdot n \leq \frac{1}{100} n - 1 \end{aligned}$$

Nun wählen wir $n_1 = 101$, sodass $f(n) \geq 0$. Anschließend wählen wir passendes c_1, sodass die Ungleichung erfüllt ist, zum Beispiel $c_1 = 1/100 - 1/101$ (diesen Wert erhält man durch Multipliation obiger Ungleichung mit $1/n$. Jetzt zeigen wir noch $f(n) \in \mathcal{O}(n^2)$, dafür müssen wir zeigen, dass $\exists n_2 \exists c_2 \forall n \geq n_2 \colon \frac{1}{100} n^2 + n \sin n \leq c_2 \cdot n^2$:

$$\begin{aligned} \frac{1}{100} n^2 + n \sin n & \leq c_2 \cdot n^2 \\ \frac{1}{100} n^2 + n \sin n \leq \frac{1}{100} n^2 + n & \underbrace{\leq}_{\text{Das zeigen wir}} c_2 \cdot n^2 \\ \frac{1}{100} n + 1 & \leq c_2 \cdot n \\ \frac{1}{n} + \frac{1}{100} & \leq c_2 \end{aligned}$$

Für $n_2 = 2$ und $c_2 = 1$ ist die obige Ungleichung immer erfüllt. Für das geforderte n_0 nehmen wir $\max(n_1, n_2) = 100$.

(g) $f(n) = n^4 - 10n^3 + 2n \in O(n^3)$

Lösung: Die Aussage ist *falsch*. Also ist zu zeigen: $\forall c \forall n \exists n_0 \geq n \colon n^4 - 10n^3 + 2n > c \cdot n^3$. Seien c und n_0 beliebig:

$$\begin{aligned} c \cdot n^3 &< n^4 - 10n^3 + 2n \\ c \cdot n^3 &< n^4 - 10n^3 < n^4 - 10n^3 + 2n \\ c + 10 &< n \end{aligned}$$

Wähle also $n = \max(n_0, c + 11)$.

(h) $f(n) = \frac{9}{n} \notin \Omega(\frac{1}{\sqrt{n}})$

Lösung: Die Aussage ist *korrekt*. Also ist zu zeigen: $\forall c \forall n \exists n_0 \geq n \colon c \cdot 1/\sqrt{n} > 9/n$. Seien n_0 und c beliebig:

$$\begin{aligned} \frac{9}{n} &< c \cdot \frac{1}{\sqrt{n}} \text{ Nächste Gleichung erhält man durch } n/\sqrt{n} = \sqrt{n} \\ \frac{9}{c} &< \sqrt{n} \\ \frac{81}{c^2} &< n \end{aligned}$$

Wähle also $n = \max(n_0, 81/c^2 + 1)$.

Aufgabe 16: Suppentöpfe

Sie kennen das. Man will sich eine Nudelsuppe kochen, findet aber nicht den passenden Deckel für den Topf, da alle Deckel und Töpfe durcheinandergekommen sind. Da Sie immer auf Ihre Töpfe geachtet haben wissen Sie, dass zu jedem Topf ein Deckel vorhanden ist.

(a) Sie möchten ein *beliebiges* passendes Deckel-Topf-Paar finden. Wie viele Vergleiche sind dafür im besten Fall nötig?

Lösung: Da ein *beliebiges* Paar gesucht wird, nehmen Sie irgendeinen Topf und probieren alle Deckel aus. Im besten Fall passt gleich der erste Deckel, Sie benötigen nur einen Vergleich.

(b) Geben Sie einen Algorithmus in Pseudocode an, der ein Feld T mit Topfgrößen und ein Feld mit Deckelgrößen D entgegennimmt. Die Ausgabe soll aus zwei Indizes i und j bestehen, sodass $\mathsf{D}[i] = \mathsf{T}[j]$. Wie viele Vergleiche braucht Ihr Algorithmus am schlechtesten Fall, um ein solches Paar zu finden? Können Sie Ihren Algorithmus verbessern, sodass er im schlechtesten Fall weniger Vergleiche braucht?

Lösung: Die Algorithmus soll also ein *beliebiges* passendes Paar finden. Wir suchen einfach den Deckel zum ersten Topf:

Algorithmus 23: findPair(int[ ] T, int[ ] D)

```
1 for j = 1 to D.length do
2     if D[j] = T[1] then
3         /* Da das Topfset vollständig ist, wird die folgende Zeile irgendwann
             erreicht und der Algo gibt immer ein Ergebnis zurück.            */
4         return (1, j)
```

Der Algorithmus braucht im schlechtesten Fall D.length -1 Vergleiche. Dies kann nicht verbessert werden, da wir im schlechtesten Fall immer den passenden Topf zuletzt erwischen, egal welchen Topf wir zuerst auswählen.

(c) Nun haben Sie genug von der Unordnung und möchten zu jedem Topf den passenden Deckel finden. Wie gehen Sie vor, um jedem Topf einen passenden Deckel zuzuordnen? Sie dürfen dabei nur Topf mit Topf und Deckel mit Deckel vergleichen. Verwenden Sie $\Theta(n \log n)$ Vergleiche.

Lösung: Man sortiert die Deckel und die Töpfe zum Beispiel mit Quicksort. Anschließend kann man den ersten Deckel auf den ersten Topf setzen ...

(d) Lösen Sie nun Teilaufgabe c), aber diesmal sollen nur Vergleiche zwischen je einem Topf und einem Deckel verwendet werden. Die Anzahl der Vergleiche soll wieder in (erwartet) $\Theta(n \log n)$ liegen. Welchem Verfahren aus der Vorlesung ähnelt Ihre Vorgehensweise?

Lösung: Sie wählen einen beliebigen Deckel und sortieren alle Töpfe, die zu klein für den Deckel sind, nach links und alle Töpfe die zu groß für den Deckel sind, nach rechts. Es bleibt ein Topf in der Mitte übrig, der zu diesem Deckel passt. Nun nehmen Sie einen Topf aus der linken Topfreihe und sortieren die Deckel mit diesem. Deckel, die zu klein sind, kommen nach links, Deckel die zu groß sind, kommen nach rechts. Es bleibt wieder ein Deckel übrig, mit dem Sie ein Paar bilden können. Sie haben nun je eine linke und rechte Seite für Deckel und Töpfe. Sie wissen, dass die Deckel auf der linken Seite zu den Töpfen auf der linken Seite passen müssen und ebenso auf der rechten Seite. Sie wiederholen den Vorgang also für links und rechts rekursiv, bis Sie alle Paare gefunden haben. Dieses Vorgehen ähnelt der QuickSort-Sortierung.