Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

# Algorithmen und Datenstrukturen
# Musterlösung Übungsblatt 4

Abgabe: Dienstag, 19. Mai, 2026, 10:00 Uhr

## Aufgabe 1: Partitioning *(12 Punkte)*

In der Vorlesung wurde die *binäre Suche* mithilfe von Vorbedingung, Nachbedingung und Schleifeninvariante gezeigt. In dieser Aufgabe wenden Sie dieselbe Methodik auf den *Partitionierungsalgorithmus* von QuickSort an. Zur Wiederholung: Bei QuickSort wird zunächst ein *Pivot-Element* ausgewählt. Nun wird das Array in 2 Teile geteilt, wobei im vorderen Teil alle Elemente kleiner und im hinteren Teil alle größer als das Pivot sein sollen. Dieser Schritt wird *Partitionierung* genannt. Anschließend wird QuickSort rekursiv auf die beiden Teilarrays angewendet.

**Algorithm 1** Quicksort pseudocode, sortiert $A[start..end]$

```
1: procedure QUICKSORT(A, start, end)
2:     if start ≥ 0 and end ≥ 0 and start < end then
3:         p ← partition(A, start, end)
4:         QUICKSORT(A, start, p)                ▷ Pivot is included
5:         QUICKSORT(A, p + 1, end)
```

Eine der folgenden Implementierungen ist korrekt, die andere nicht. Ihre Aufgabe ist es, mithilfe von Vorbedingung, Nachbedingung und Schleifeninvariante die Korrektheit einer Implementierung zu beweisen oder ein Beispiel zu konstruieren, bei dem die Partitionierung nicht wie gewünscht funktioniert. Nachstehend sind die Bedingungen für ein Array $A[0..n-1]$ der Größe $n$ und ein *Pivot-Element* `pivot` aufgeführt, die für einen korrekten *Partitionierungsalgorithmus* erfüllt sein sollten.

**Algorithm 2** Partition1(A, start, end)

```
1: procedure PARTITION(A, start, end)
2:     pivot := A[start]
3:     l := start
4:     r := end
5:     while True do
6:         while l ≤ r and A[l] < pivot do
7:             l := l + 1
8:         while l ≤ r and A[r] ≥ pivot do
9:             r := r − 1
10:        if l > r then
11:            swap(A[start], A[l])
12:            return l
13:        else
14:            swap(A[l], A[r])
```

**Algorithm 3** Partition2(A, start, end)

```
1: procedure PARTITION(A, start, end)
2:     pivot := A[start]
3:     l := start
4:     r := end
5:     while True do
6:         while l ≤ r and A[l] ≤ pivot do
7:             l := l + 1
8:         while l ≤ r and A[r] > pivot do
9:             r := r − 1
10:        if l > r then
11:            swap(A[start], A[r])
12:            return r
13:        else
14:            swap(A[l], A[r])
```

- Vorbedingung:

$$0 \leq start < end < n.$$

- Schleifeninvariante:

$$\forall x \in [start, l-1] : A[x] \leq pivot$$
$$\forall x \in [r+1, end] : A[x] > pivot$$

- Nachbedingung:

$$\forall x \in [start, r] : A[x] \leq pivot$$
$$\forall x \in [r+1, end] : A[x] \geq pivot$$

Hinweis: Wenn Sie ein Beispiel konstruieren, bei dem der Algorithmus nicht wie gewünscht funktioniert, müssen Sie die Vorbedingung erfüllen, damit das Beispiel gültig ist. Falls Sie ein Beispiel angeben, bei dem der Algorithmus nicht terminiert, ist dies ebenfalls ein gültiges Gegenbeispiel, da die Nachbedingung in diesem Fall nicht erfüllt werden kann. Wenn Sie die Korrektheit eines Algorithmus behaupten, begründen Sie bitte auch kurz, warum dieser terminiert (Laufzeiten sind hier nicht relevant; es muss lediglich gezeigt werden, dass der Algorithmus irgendwann terminiert).
Die Schleifeninvariante soll sowohl vor als auch nach jeder Iteration der äußeren Schleife gelten. Es muss also gezeigt werden, dass die Schleifeninvariante zu Beginn der ersten Iteration der äußeren Schleife gilt, und dass sie, wenn sie vor einer Iteration der äußeren Schleife gültig ist, auch nach dieser Iteration gültig bleibt.

## Musterlösung

**Der erste Algorithmus ist nicht korrekt.** Nehmen wir das Array $A = [5, 8, 2]$ mit pivot 5. Hier würde im ersten Schritt die 5 mit der 2 getauscht werden ($[2, 8, 5]$) ohne dass sich $l$ und $r$ verändern. In der nächsten Iteration erhöht sich $l$ um 1 und $r$ verringert sich um 2, es gilt also $l > r$. Wir tauschen nun also 2 und 8, und terminieren mit $[8, 2, 5]$. Das stimmt aber nicht mit der Nachbedingung überein.
**Der zweite Algorithmus ist korrekt.** Begründen wir zuerst, warum dieser immer terminiert. Wir wissen, dass bei $l > r$ garantiert terminiert wird, wir nehmen also an, dass $l \leq r$. Sobald $l$ größer oder $r$ kleiner wird, machen wir Fortschritt, da beides ganzzahlige Werte sind, können wir diese nur endlich oft vergrößern/verringern. Es gibt nur einen Fall, in dem wir keinen Fortschritt machen, und das ist, wenn $A[l] > pivot$ und $A[r] \leq pivot$. Dann werden diese beiden aber garantiert mit SWAP vertauscht und somit wird in der nächsten Iteration garantiert $l$ erhöht oder $r$ verringert und somit Fortschritt gemacht. Somit kommen sich in jeder Iteration entweder $l$ und $r$ näher, oder sie werden vertauscht und tun dies dann nachfolgend. Somit muss irgendwann $l > r$ gelten.
Nun schauen wir uns die Korrektheit an.

Bei gegebener Vorbedingung gilt hier $l = start$ und $r = end$. Somit werden die beiden Intervalle der Schleifeninvariante zu leeren Intervallen, was die Aussage trivialerweise wahr werden lässt. Nun muss gezeigt werden, dass die Schleifeninvariante nach jeder Iteration der äußeren Schleife weiterhin gilt. Wir nehmen also an, dass die Schleifeninvariante für unser aktuelles $l$ und $r$ am Anfang der Schleife gilt. Die Invariante muss nun auch am Ende der Schleife gelten, also entweder wenn wir im IF-Case (bei dem die Schleife terminiert) oder im ELSE-Case (in welchem nur ein SWAP aufgerufen wird) sind. Wir führen eine Fallunterscheidung durch.

- Wir enden im ELSE-Case:
  Wenn wir in diesen Fall kommen, muss $l \leq r$ sein, sonst wären wir im IF-Case. Außerdem muss wegen den beiden inneren While-Loops gelten, dass $A[l] > pivot$ und $A[r] \leq pivot$. Für alle Indizes $i < l$ muss hier gelten, dass $A[i] \leq pivot$, da wir sonst noch oben in der While-Schleife wären. Aus dem gleichen Grund muss auch für alle $j > r$ gelten, dass $A[j] > pivot$. Der Swap verändert diese Vorgänger und Nachfolger nicht, sondern nur $A[l]$ und $A[r]$ und somit gilt die Schleifeninvariante nach dem SWAP.

- Wir enden im IF-Case:
  Wir betrachten nun den Zustand vor dem SWAP im IF. Wir wissen, dass $l > r$, und wie oben schon erwähnt, implizieren alle Indizes $i < l$, dass $A[i] \leq pivot$, und umgekehrt gilt auch $A[j] > pivot$ für alle $j > r$. Da $l > r$ und wir beide immer nur in Einzelschritten vergrößern bzw. verringern, muss sogar $l = r + 1$ bzw. $r = l - 1$ gelten. Somit ist $A[r] = A[l-1] \leq pivot$ und $A[l] = A[r+1] > pivot$ nach der Schleifeninvariante. Nun wird $A[start]$ mit $A[r]$ geswappt; da aber nach Schleifenbedingung $A[start] \leq pivot$ (tatsächlich sogar $= pivot$) und eben auch $A[r] \leq pivot$ gilt, ändert dieser SWAP nichts an der Korrektheit der Schleifeninvariante.

Die Nachbedingung folgt aus der Schleifeninvariante gepaart mit der Bedingung $l = r + 1$ (wie eben beschrieben, diese muss gelten, bevor die Schleife terminiert). Setzt man dies in die Schleifeninvariante ein, folgt direkt die Nachbedingung.

## Aufgabe 2: Quadratwurzel *(4 Punkte)*

Gegeben sei eine natürliche Zahl $n \in \mathbb{N}$. Bestimmen Sie die größte ganze Zahl $x$, sodass

$$x^2 \leq n.$$

Beschreiben Sie einen Algorithmus zur Berechnung von $x$, der eine Laufzeit in $O(\log n)$ hat. Beweisen Sie, dass Ihr Algorithmus die erwünschte Laufzeit hat.

## Musterlösung

Wir benutzen einen Algorithmus, der die binäre Suche aus der Vorlesung adaptiert.

1. Initialisiere '$low := 0$' und '$high := n$'. Dies definiert den Suchbereich für $x$.
2. Führe eine binäre Suche im Bereich '[low, high]' durch:
   a. Berechne die Mitte $m := \lfloor \frac{low+high}{2} \rfloor$.
   b. Wenn $m \cdot m \leq n$, dann ist '$m$' ein potenzieller Kandidat für $x$. Verschiebe den Suchbereich nach rechts: '$low := m$'. c. Wenn $m \cdot m > n$, dann ist '$m$' zu groß. Verschiebe den Suchbereich nach links: '$high = m - 1$'.
3. Wiederhole Schritt 2, bis '$high - low \leq 1$'.
4. In diesem Fall teste, ob $high \cdot high \leq n$, wenn ja, gib $high$ zurück, sonst gib $low$ zurück.

**Laufzeit und Korrektheit:** Zu Beginn liegt das gesuchte $x$ sicher im Bereich $\{0, 1, \ldots, n\}$, sprich zwischen $low$ und $high$. Nach jeder Iteration, in der sich $low$ vergrößert, gilt dennoch $x \geq low$, denn wenn $x < low$ wäre, dann würde $low$ nicht vergrößert werden (da $m \cdot m > n$ gelten würde). Aus demselben Grund wissen wir auch, dass $x \leq high$ immer gelten muss. Somit liegt $x$ immer im Suchbereich. Sobald der Suchbereich auf wenige Felder reduziert ist, können wir direkt testen, welches dieser Felder das korrekte $x$ ist. Somit ist die Korrektheit des Algorithmus gewährleistet.

Dieser Algorithmus verwendet eine binäre Suche. In jeder Iteration wird der Suchbereich $[low, high]$ halbiert. Der anfängliche Suchbereich ist von 0 bis $n$. Die Anzahl der Iterationen, die erforderlich sind, um den Suchbereich auf 1 zu reduzieren, ist $O(\log n)$. Da die Operationen innerhalb der Schleife (Multiplikation, Vergleich, Addition/Subtraktion) konstante Zeit in Anspruch nehmen, ist die Gesamtlaufzeit des Algorithmus $O(\log n)$.

## Aufgabe 3: Schnelle Potenzierung *(4 Punkte)*

Gegeben seien zwei Zahlen $a \in \mathbb{R}$ und $n \in \mathbb{N}$. Geben Sie einen Algorithmus, der den Wert $a^n$ berechnet. Ihr Algorithmus muss eine Laufzeit von $O(\log n)$ haben. Beweisen Sie, dass Ihr Algorithmus die erwünschte Laufzeit hat.

## Musterlösung

Die Idee ist, die Aufgabe durch rekursives Quadrieren zu lösen. Wenn $n$ gerade ist, dann gilt $a^n = (a^{n/2})^2$. Wenn $n$ ungerade ist, dann gilt $a^n = a \cdot (a^{(n-1)/2})^2$. Somit können wir die Potenzierung durch wiederholtes Quadrieren und Multiplikation lösen. **Algorithmus zur Berechnung von $a^n$:**

**Algorithm 4** Schnelle Exponentiation

```
1: function Exp(a, n)
2:     if n == 0 then
3:         return 1                                  ▷ Basisfall
4:     tmp ← Exp(a, ⌊n/2⌋)                           ▷ Rekursiver Aufruf
5:     if n (mod 2) == 0 then
6:         return tmp · tmp                          ▷ Fall 1: n ist gerade
7:     else
8:         return a · tmp · tmp                      ▷ Fall 2: n ist ungerade
```

**Beweis der Laufzeit $O(\log n)$:**
In jeder Iteration der Schleife wird der Exponent halbiert (ganzzahlige Division durch 2). Wir bekommen folgende rekursive Laufzeit:

$$T(0) = O(1) \qquad \text{(Basisfall)}$$
$$T(n) = T(\lfloor n/2 \rfloor) + O(1) \qquad \text{(Rekursiver Fall)}$$

Da $T(\lfloor n/2 \rfloor) \leq T(n/2) + O(1)$ gilt (im Falle, dass $n$ ungerade ist, muss man noch konstante Zeit für die Multiplikation mit $a$ einrechnen), können wir die Rekursion zu

$$T(n) \leq T(n/2) + O(1)$$

vereinfachen. Da dies die selbe Rekursionsgleichung wie bei der binären Suche ist, wissen wir aus der Vorlesung, dass $T(n) = O(\log n)$.