Algorithmen und Datenstrukturen 1

Ausgearbeitetes Übungsblatt 1

Datum: 17. Mai 2005
Erstellt mit LaTeX

Aufgabe 1.1

Aufgabenstellung:

(a) Beweisen oder widerlegen Sie, dass für die im Folgenden definierte Funktion $f(n)$ die Beziehung $f(n) = \Theta(n^2)$ gilt.
Beachten Sie, dass zu einem vollständigen Beweis gegebenenfalls auch geeignete Werte für die verwendeten Konstanten anzugeben sind.

$$f(n) = \begin{cases} 3n^2 - 7n\sqrt{n} + \pi & falls\ n\ teilbar\ durch\ 3 \\ 2n^3 + \sqrt{n} - 1/n^2 & falls\ n\ nicht\ teilbar\ durch\ 3 \end{cases}$$

(b) Beweisen oder widerlegen Sie die nachstehende Behauptung für eine beliebige positive Funktion $g(n)$:

$$g(n) = \Theta(g(n)) \quad \Rightarrow \quad f(n) = O(2g(n))$$

Lösung:

(a)

$$f(n) = \begin{cases} 3n^2 - 7n\sqrt{n} + \pi & falls\ n\ teilbar\ durch\ 3 \\ 2n^3 + \sqrt{n} - 1/n^2 & falls\ n\ nicht\ teilbar\ durch\ 3 \end{cases}$$

$f(n) - \Theta(n^2)$?
Überprüfung mit Hilfe der zweiten Variante der Funktion:

$$0 \leq c_1 \cdot n^2 \leq 2n^3 + \sqrt{n} - \frac{1}{n^2} \leq c_2 \cdot n^2$$

Es existieren keine Werte für c_2, sodass diese Bedingung erfüllt wäre. Also liegt $f(n)$ nicht in $\Theta(n^2)$.

(b) $g(n) = \Theta(g(n)) \quad \Rightarrow \quad f(n) = O(2g(n))$?

$f(n) - \Theta(g(n)) \quad \Leftrightarrow \quad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_3 \cdot g(n),\ c_1, c_2 > 0$
$f(n) = O(2g(n)) \quad \Leftrightarrow \quad f(n) \leq c \cdot 2g(n),\ c > 0$
Die zweite Bedingung ist zumindest dann wahr, wenn die erste Bedingung wahr ist und $c = \frac{c_2}{2}$ ist. Die Behauptung ist damit bestätigt.

Aufgabe 1.2

Aufgabenstellung:

Überlegen Sie, welche der folgenden Aussagen für die Funktion $f(n)$ gelten:

$$f(n) = \begin{cases} 2n\left(\sqrt{n} + 3n/2\right) - 1 & falls\ n > 10^4 \\ \left(n^3 log\, n - \sqrt{n}\right)/n + 4 & falls\ n \leq 10^4 \end{cases}$$

$f(n)$ ist	$O(...)$	$\Omega(...)$	$\Theta(...)$
$\sqrt{(n)}$	☐	☐	☐
$1/n^2$	☐	☐	☐
n^2	☐	☐	☐
$n^2 log\, n$	☐	☐	☐

Lösung:

$f(n)$ ist	$O(...)$	$\Omega(...)$	$\Theta(...)$
$\sqrt{(n)}$	☐	☒	☐
$1/n^2$	☐	☒	☐
n^2	☒	☒	☒
$n^2 log\, n$	☒	☐	☐

Aufgabe 1.3

Aufgabenstellung:

Welche Laufzeiten in Θ-Notation haben die folgenden Algorithmen abhäangig von n?

(i)
```
p = n + 10;
solange p > 0 {
  k = n2 - p;
  p = p - 1;
}
```

(ii)
```
k = n;
wiederhole {
  k = k/3;
  l = n * k;
} bis k <=1;
```

(iii)
```
m = n;
wiederhole {
  für i = 1, . . . . , 15 {
    p = p + i;
  }
  // floor: größte ganze Zahl <= m/2
  m = floor(m/2);
} bis m <= 0
```

(iv)
```
m = n;
solange m > 0 {
  für i = 1, . . . . , m + 3 {
    k = k + i;
  }
  m = floor(m/2);
}
```

Lösung:

(i) Die Schleife wird insgesamt $n + 10$-mal ausgeführt. Daher ist $T(n) = \Theta(n)$.

(ii) Bei jedem Schleifendurchgang wird die Variable k, von welcher die Abbruchbedingung der Schleife abhängt, durch 3 dividiert. Daher ist $T(n) = \Theta(^3log\, n)$.

(iii) Analog zu (ii): Die Variable m wird bei jedem Schleifendurchgang durch 2 dividiert; die innere Schleife wird 15-mal ausgeführt. Daher ist $T(n) = \Theta(ld\, n) \cdot \Theta(1) = \Theta(ld\, n)$.

(iv) Die Laufzeitfunktion der äußere Schleife (zunächst ohne Beachtung der inneren Schleife) liegt in $\Theta(ld\, n)$. In Abhängigkeit von der Variable m, welche in der äußeren Schleife in jedem Schleifendurchgang durch 2 dividiert wird, wird die innere Schleife jedes Mal $m+3$-mal ausgeführt. Da zu Beginn des Algorithmus $m = n$ gesetzt wird, wird die innere Schleife also beim ersten Durchlauf der äußeren Schleife n-mal ausgeführt, beim zweiten Durchlauf $\frac{n}{2}$-mal usw., in Summe also $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots$ Mal. Für große n, hier also eine große Anzahl von Iterationen der äußeren Schleife, konvergiert diese Summe gegen $2n$ (Summenformel der unendlichen geometrischen Reihe $\sum_{k=0}^{\infty} x^k = \lim_{n\to\infty} \sum_{k=0}^{n} = \lim_{n\to\infty} \frac{x^{n+1}-1}{x-1} = \frac{1}{1-x}$, hier: $x = \frac{1}{2}$, daher $\frac{1}{1-x} = \frac{1}{\frac{1}{2}} = 2$). Die Laufzeitfunktion der äußeren Schleife lautet also dann $T(n) = 2n + ld\, n$, und dieser Wert ist sicher linear, liegt also in $\Theta(n)$.

Aufgabe 1.4

Aufgabenstellung:

Die Folge der sogenannten *Fibonacci*-Zahlen $f(n) = \langle 1, 1, 2, 3, 5, \ldots \rangle$ kann durch die folgende rekursive Funktion $f(n)$ beschrieben werden:

$$f(n) = \begin{cases} 1 & falls\, n \leq 2 \\ f(n-1) + f(n-2) & falls\, n > 2 \end{cases}$$

(a) Schreiben Sie einen Algorithmus in Pseudocode, der diese Funktion in der angegebenen rekursiven Form implementiert.

(b) Geben Sie die Zahl der von Ihrer Implementierung vorgenommenen rekursiven Aufrufe bei der Berechnung von f_{10} an.

(c) Beweisen oder widerlegen Sie, dass die Laufzeit Ihrer Implementierung in $O(2^n)$ liegt.

(d) Beweisen oder widerlegen Sie, dass die Laufzeit Ihrer Implementierung polynomiell ist, dass also eine natürliche Zahl x existiert, für die $f(n) = O(n^x)$ gilt.

Lösung:

(a)

```
Fibonacci(index) {
  if(index<3) {
    return 1;
  }
  else {
    return Fibonacci(index-1) + Fibonacci(index-2);
  }
}
```

(b) Auf Basis der Formel von Binet ($f_n = \frac{1}{\sqrt{5}} \cdot (a^n - b^n)$ mit $a = \frac{1+\sqrt{5}}{2}$ sowie $b = \frac{1-\sqrt{5}}{2}$) lässt sich die Anzahl der rekursiven Aufrufe bei der Berechnung eines Gliedes der Fibonacci-Folge berechnen durch $2 \cdot \sqrt{5} \cdot (a^n - b^n) - 2$. Setzt man in diese Formel $n = 10$ ein, so erhält man 108 Aufrufe.

(c) Gemäß der Formel für die Anzahl der rekursiven Aufrufe erhält man für die Anzahl der Ausführungen des Algorithmus $2 \cdot \sqrt{5} \cdot (a^n - b^n) - 1$ (hier -1, weil der erste Aufruf des Algorithmus, welcher kein rekursiver Aufruf ist, hinzuaddiert werden muss). Dies kann man, auch als Laufzeitfunktion verwenden:

$$T(n) = 2 \cdot \sqrt{5} \cdot (a^n - b^n)$$

Wenn $T(n)$ nun in $O(2^n)$ liegt, müssen ein $c > 0$ sowie ein n_0 existieren, sodass für alle $n > n_0$ gilt:

$$2 \cdot \sqrt{5} \cdot (a^n - b^n) - 1 \leq c \cdot 2^n$$

Division durch 2^n ergibt:

$$2 \cdot \sqrt{5} \cdot \left[\left(\frac{1+\sqrt{5}}{4} \right)^n - \left(\frac{1-\sqrt{5}}{4} \right)^n \right] - 1 \leq c$$

und weiters:

$$2 \cdot \sqrt{5} \cdot \left(\frac{1+\sqrt{5}}{4} \right)^n - 2 \cdot \sqrt{5} \cdot \left(\frac{1-\sqrt{5}}{4} \right)^n - 1 \leq c$$

Da nun sowohl $\frac{1+\sqrt{5}}{4}$ als auch $\frac{1-\sqrt{5}}{4}$ kleiner als 2 sind, ist 2^n sicher eine obere Schranke für diese Funktion. Es sollte auch nicht besonders schwer sein, ein c zu finden, welches dieser Bedingung genügt. Also liegt $T(n)$ in $O(2^n)$.
Es wäre auch möglich gewesen, damit zu argumentieren, dass ein symmetrischer binärer Baum, der eine Tiefe von $n-1$ hat, immer 2^{n-1} Knoten hat. Nun ergibt die Aufrufkette des oben gebrachten Algorithmus für die Berechnung eines Gliedes der Fibonacci-Folge, wenn man diese Kette als binären Baum darstellt, einen Baum, der nicht symmetrisch ist, da dem Knoten, welcher dem Aufruf des Algorithmus mit dem Argument n entspricht, nicht zwei Knoten mit dem Aufruf mit $n-1$ folgen, sondern ein Knoten mit $n-1$ und ein Knoten mit $n-2$. Der entstehende Baum hat somit zwar eine Tiefe von $n-1$, doch es fehlt gegenüber dem symmetrischen binären Baum eine für große n immer größer werdende Anzahl an Knoten, also ist die Anzahl der Knoten dieses Aufrufbaumes sicher kleiner als 2^{n-1}. Damit wäre auch gezeigt, dass 2^n nicht nur für die Anzahl der Knoten dieses Baumes, sondern auch für die Anzahl der rekursiven Aufrufe bei der Berechnung eines Gliedes der Fibonacci-Folge und damit auch für die Laufzeit dieser Berechnung ist.

(d) Die Laufzeitfunktion dieses Algorithmus ist eine Exponentialfunktion. Jede solche Funktion wächst bei größer werdendem n stets schneller als jede Polynomfunktion. Dieser Algorithmus kann also keine polynomielle Laufzeitfunktion haben.

Aufgabe 1.5

Aufgabenstellung:

Modifizieren Sie den Algorithmus aus Aufgabe 1.4 unter Beibehaltung seiner Rekursivität derart, dass er lineare Laufzeit erhält, also Laufzeit in $O(n)$.
Nehmen Sie dabei an, dass Ihnen ein beliebig großes, globales Array $A = \langle A[1], A[2], \ldots \rangle$ zum Speichern von Zwischenergebnissen zur Verfügung steht. Beweisen Sie, dass Ihre verbesserte Version lineare Laufzeit hat, und analysieren Sie auch den Speicherverbrauch in Θ-Notation.

Lösung:

```
Fibonacci(index) {
  if(index>2) {
    Fibonacci(index-1);
    A[index]=A[index-1]+A[index-2];
  }
  else {
    A[1]=1; A[2]=1;
  }
  return A[index];
}
```

Die Laufzeit dieses Algorithmus ist linear (in $\Theta(n)$), ebenfalls sein Speicherbedarf (also auch in $\Theta(n)$).

Aufgabe 1.6

Aufgabenstellung:

Führen Sie das Sortierverfahren *Selection-Sort* (Sortieren durch Auswahl) mit dieser Zahlenfolge durch:

$$\langle 3, 1, 7, 6, 2, 4, 9, 5, 8 \rangle$$

Stellen Sie die Daten nach jeder Iteration dar. Markieren Sie jeweils die Datenelemente, deren Position sich verändert hat.
Ein Sortierverfahren heißt stabil, wenn es so implementiert werden kann, dass die Reihenfolge der Elemente mit gleichem Schlüssel vor und nach dem Sortiervorgang gleich ist. Ist Selection-Sort stabil? Begründen Sie Ihre Antwort.

Lösung:

3	1	7	6	2	4	9	5	8
1	3	7	6	2	4	9	5	8
1	2	7	6	3	4	9	5	8
1	2	3	6	7	4	9	5	8
1	2	3	4	7	6	9	5	8
1	2	3	4	5	6	9	7	8
1	2	3	4	5	6	9	7	8
1	2	3	4	5	6	7	9	8
1	2	3	4	5	6	7	8	9

Dieses Sortierverfahren ist nicht stabil, weil Elemente mit gleichem Schlüssel durch Anwendung dieses Algorithmus gegebenenfalls gegeneinander ausgetauscht werden. Hier ein Beispiel:

3_1	3_2	2	1
1	3_2	2	3_1
1	2	3_2	3_1

An diesem Beispiel ist deutlich zu erkennen, dass 3_1 vor der Sortierung vor 3_2 steht, nach der Sortierung allerdings dahinter. Daran ist die Instabilität von Selection-Sort zu erkennen.
Ein stabiles Sortierverfahren ist beispielsweise der in Aufgabe 1.7 vorgestellte *Bubblesort*. Dort wird die Stabilität dadurch gewährleistet, dass Elemente mit gleichem Schlüssel einander nicht „überholen“ können.

Aufgabe 1.7

Aufgabenstellung:

Eine gängige Implementierung des Sortierverfahrens *Bubblesort* sieht wie folgt aus:

```
für i = 1, . . . , n − 1 {
  für j = 1, . . . , n − i {
    falls A[j] > A[j + 1] dann {
      Vertausche A[j] und A[j + 1];
    }
  }
}
```

(a) Probieren Sie den Algorithmus mit dieser Eingabefolge aus:

$$\langle 3, 1, 7, 6, 2, 4, 9, 5, 8 \rangle$$

Stellen Sie die Daten nach jeder Iteration dar. Markieren Sie jeweils die Datenelemente, deren Position sich verändert hat.

(b) Hängt die Laufzeit dieses Algorithmus vom Inhalt der Eingabefolge A ab? - Wenn ja: wie? - Wenn nein: warum nicht? - Wie ist die Laufzeit dieses Algorithmus in Θ-Notation?

Lösung:

3	1	7	6	2	4	9	5	8
1	3	6	2	4	7	5	8	9
1	3	2	4	6	5	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Die Laufzeit dieses Algorithmus ist insofern abhängig von der Eingabe, als dass mehr oder weniger Vertauschungen in der inneren Schleife durchgeführt werden. In Θ-Notation hat dieser Algorithmus allerdings immer die Laufzeit $\Theta(n^2)$, unabhängig von der Eingabe.

Aufgabe 1.8

Aufgabenstellung:

Gegeben seien Daten in *aufsteigend* sortierter Form, die mit Insertion-Sort bzw. Merge-Sort *aufsteigend* sortiert werden sollen. Welches der beiden Sortierverfahren ist für diesen konkreten Fall effizienter? Zeigen Sie dies, indem Sie für beide Algorithmen den Aufwand für den konkreten Fall in Θ-Notation angeben.
Welches dieser beiden Sortierverfahren ist im Durchschnitt (gemittelt über alle möglichen Eingabekombinationen) effizienter? Zeigen Sie dies ebenfalls, indem Sie für beide Algorithmen den Aufwand für den durchschnittlichen Fall in Θ-Notation angeben.

Lösung:

Insertion Sort: $T_{best} = \Theta(n)$, $T_{avg} = T_{worst} = \Theta(n^2)$
Merge-Sort: $T_{best} = T_{avg} = T_{worst} = n \cdot ld\, n$
Im besten Fall ist zwar Insertion Sort besser, im durchschnittlichen Fall (und der tritt weit häufiger auf) wie auch im schlechtesten Fall ist Merge-Sort die bessere Wahl.

Aufgabe 1.9

Aufgabenstellung:

Ändern Sie den Algorithmus von *Quicksort* so, dass zur Wahl des Pivotelements jeweils das erste, mittlere und letzte Element der Teilfolge betrachtet werden und dann jenes Element gewählt wird, dessen Wert zwischen dem der beiden anderen liegt.

(a) Geben Sie den gesamten Pseudocode für den veränderten Quicksort an.

(b) Halten Sie die Änderung für sinnvoll? Warum (nicht)?

(c) Sortieren Sie die Zahlenfolge

$$\langle 40, 18, 93, 114, 90, 11, 26, 29, 48, 30, 20, 78 \rangle$$

mit dem von Ihnen veränderten Algorithmus.

Lösung:

Standard-Quicksort (gemäß Skriptum):

```
Quicksort(var A,l,r) {
  if(l<r) {
    x=A[r];
    p=Partition(A,l,r,x);
    Quicksort(A,l,p-1);
    Quicksort(A,p+1,r);
  }
}

Partition(var A,l,r,x) {
  i=l-1; j=r;
  while(i<j) {
    until(A[i]<x) i++;
    until(A[j]>x) j++;
    if(i<j) swap(A[i],A[j]);
  }
  swap(A[i],A[j]);
}
```

Variante 1
Hier wird lediglich der Algorithmus in **Quicksort** selbst verändert, der Algorithmus **Partition** bleibt unberührt. Die Veränderung besteht darin, dass jenes der Elemente, welche am Anfang, in der Mitte und am Ende der zu sortierenden Folge stehen, dessen Schlüssel zwischen den Schlüsseln der beiden anderen Elemente liegt, als Pivotelement ganz nach rechts verschoben wird, sodass Partition mit dem ganz rechts stehenden Pivotelement arbeiten kann.
Die im veränderten Algorithmus **Quicksort** verwendete Funktion **mitte** liefert den mittleren Wert der drei angegebenen Argumente; die Funktion **floor** liefert die größte ganze Zahl kleiner oder gleich dem angegebenen Argument, die Funktion **swap** vertauscht die Werte der beiden übergebenen Variablen.

```
Quicksort(var A,l,r) {
  if(l<r) {

    // Veränderung Anfang
    m=floor((r+l)/2);
    x=mitte(A[l],A[m],A[r]);
    y=index von x;
    if(y!=r) swap(y,r);

    // Veränderung Ende
    p=Partition(A,l,r,x);
    Quicksort(A,l,p-1);
    Quicksort(A,p+1,r);
  }
}

Partition(var A,l,r,x) {
  i=l-1; j=r;
  while(i<j) {
    until(A[i]<x) i++;
    until(A[j]>x) j++;
```

```
    if(i<j) swap(A[i],A[j]);
  }
  swap(A[i],A[j]);
}
```

Diese Veränderung ist in jener Weise interessant, da sie den Worst-Case des Quicksort-Algorithmus positiv beeinflusst, indem das mittlere von drei Elementen als Pivotelement auswählt. In dieser Hinsicht ist die Veränderung durchaus von Vorteil.

40	18	93	114	90	11	26	29	48	30	20	78	
78	18	93	114	90	11	26	29	48	30	20	40	Pivotelement: 40
20	18	93	114	90	11	26	29	48	30	78	40	
20	18	30	114	90	11	26	29	48	93	78	40	
20	18	30	29	90	11	26	114	48	93	78	40	
20	18	30	29	26	11	90	114	48	93	78	40	
20	18	30	29	26	11	40	114	48	93	78	90	neue Position des Pivotelements: 7
11	18	30	29	26	20	40	114	48	93	78	90	Pivotelement: 20
11	18	20	29	26	30	40	114	48	93	78	90	neue Position des Pivotelements: 3
11	18	20	29	26	30	40	114	48	93	78	90	Pivotelement: 18
11	18	20	29	26	30	40	114	48	93	78	90	neue Position des Pivotelements: 2
11	18	20	30	26	29	40	114	48	93	78	90	Pivotelement: 29
11	18	20	26	30	29	40	114	48	93	78	90	
11	18	20	26	29	30	40	114	48	93	78	90	neue Position des Pivotelements: 5
11	18	20	26	29	30	40	114	48	90	78	93	Pivotelement: 93
11	18	20	26	29	30	40	78	48	90	114	93	
11	18	20	26	29	30	40	78	48	90	93	114	neue Position des Pivotelements: 11
11	18	20	26	29	30	40	90	48	78	93	114	Pivotelement: 78
11	18	20	26	29	30	40	48	90	78	93	114	
11	18	20	26	29	30	40	48	78	90	93	114	neue Position des Pivotelements: 9
11	18	20	26	29	30	40	48	78	90	93	114	

Variante 2

Hier werden sowohl der Algorithmus **Quicksort** als auch der Algorithmus **Partition** weiträumig verändert, sodass eine mehr oder weniger komplett andere Quicksort-Variante entsteht; bezüglich der Funktionen **floor**, **swap** und **mitte** siehe bei Variante 1.

```
Quicksort(var A,l,r) {
  if(l<r) {

    // Veränderung Anfang
    m=floor((r+l)/2);
    x=mitte(A[l],A[m],A[r];
    // Veränderung Ende
    p=Partition(A,l,r,x);
    Quicksort(A,l,p-1);
    Quicksort(A,p+1,r);
  }
}

Partition(var A,l,r,x) {

  // komplett verändert
  while(l<r) {
    while(A[l]<x) l++;
    while(A[r]>x) r--;
    if(l<r) swap(A[l],A[r]);
  }
```

```
  return l;
}
```

Hier wird ebenfalls im Worst-Case des Quicksort-Algorithmus eingegriffen, allerdings ist hier der Unterschied zur ersten Variante, dass das Pivotelement zunächst an der Stelle verbleibt, an der es sich zu Beginn des Aufrufs der Prozedur **Quicksort** befindet, dann aber in die Sortierung mit einbezogen wird, d. h. auch das Pivotelement wird so lange „herumgeschoben“, bis es sich an der richtigen Stelle befindet. Durch die daraus resultierende höhere Anzahl an Datensatzbewegungen halte ich diese Veränderung für nicht sinnvoll.

40	18	93	114	90	11	26	29	48	30	20	78	Pivotelement: 40
20	18	93	114	90	11	26	29	48	30	40	78	
20	18	40	114	90	11	26	29	48	30	93	78	
20	18	30	114	90	11	26	29	48	40	93	78	
20	18	30	40	90	11	26	29	48	114	93	78	
20	18	30	29	90	11	26	40	48	114	93	78	
20	18	30	29	40	11	26	90	48	114	93	78	
20	18	30	29	26	11	40	90	48	114	93	78	neue Position des Pivotelements: 7
20	18	30	29	26	11	40	90	48	114	93	78	Pivotelement: 20
11	18	30	29	26	20	40	90	48	114	93	78	
11	18	20	29	26	30	40	90	48	114	93	78	neue Position des Pivotelements: 3
11	18	20	29	26	30	40	90	48	114	93	78	Pivotelement: 18
11	18	20	29	26	30	40	90	48	114	93	78	neue Position des Pivotelements: 2
11	18	20	29	26	30	40	90	48	114	93	78	Pivotelement: 29
11	18	20	26	29	30	40	90	48	114	93	78	neue Position des Pivotelements: 5
11	18	20	26	29	30	40	90	48	114	93	78	Pivotelement: 90
11	18	20	26	29	30	40	78	48	114	93	90	
11	18	20	26	29	30	40	78	48	90	93	114	neue Position des Pivotelements: 10
11	18	20	26	29	30	40	78	48	90	93	114	Pivotelement: 48
11	18	20	26	29	30	40	48	78	90	93	114	neue Position des Pivotelements: 8
11	18	20	26	29	30	40	48	78	90	93	114	Pivotelement: 114
11	18	20	26	29	30	40	48	78	90	93	114	neue Position des Pivotelements: 12
11	18	20	26	29	30	40	48	78	90	93	114	

Insgesamt ist zu den beiden Veränderungen jedoch anzumerken, dass keine der beiden Veränderungen eine Wirkung hat, wenn zufälligerweise $A[l] = A[m] = A[r]$ ist. Dennoch ist für den durchschnittlichen Fall die erste der beiden gebrachten Varianten die bessere.

Aufgabe 1.10

Aufgabenstellung:

Gegeben ist der folgende Algorithmus:

```
Sortiere (var A,l,r) {
  falls r - l > 1 dann {
    min = Position des minimalen Elementes in A[l . . . r];
    max = Position des maximalen Elementes in A[l . . . r];
    Vertausche A[l] und A[min];
    Vertausche A[r] und A[max];
    Sortiere (A, l + 1, r - 1);
  }
  sonst {
    falls r - l = 1 dann {
      falls A[l] > A[r] dann {
```

```
        Vertausche A[l] und A[r];
      }
    }
  }
}
```

Nehmen Sie an, der Algorithmus wird auf eine Folge A aus n Zahlen angewendet. Der entsprechende Funktionsaufruf ist dann Sortiere(A,1,n). Warum sortiert der Algorithmus nicht jede Folge korrekt?
Ändern Sie den Algorithmus so, dass er funktioniert. Ihre Änderung sollte die zentrale Idee des Algorithmus beibehalten, also das kleinste Element nach vorne bringen, das größte nach hinten und dann den Mittelteil rekursiv sortieren. Geben Sie den Pseudocode Ihres neuen Algorithmus an.

Lösung:

Ist nach der Ermittlung des minimalen und des maximalen Elements in $A[l \dots r]$ $r = min$, so tritt ein Problem auf, so wird das zunächst korrekt an die Stelle ganz links positionierte Element der zu sortierenden Folge ganz nach rechts verschoben; das größte Element der zu sortierenden Folge wurde jedoch im ersten Vertauschungsschritt an eine andere Stelle verschoben und wird nicht korrekt einsortiert:

5	4	0	1	2	$min = 3$	$max = 4$
0	4	5	1	2		
2	4	5	1	0		
2	4	5	1	0	$min = 4$	$max = 3$
2	1	5	4	0		
2	1	4	5	0		

Durch folgende Korrektur funktioniert der Algorithmus einwandfrei:

```
Sortiere (var A,l,r) {
  falls r - l > 1 dann {
    min = Position des minimalen Elementes in A[l . . . r];
    Vertausche A[l] und A[min];
    max = Position des maximalen Elementes in A[l+1 . . . r];
    Vertausche A[r] und A[max];
    Sortiere (A, l + 1, r - 1);
  }
  sonst {
    falls r - l = 1 dann {
      falls A[l] > A[r] dann {
        Vertausche A[l] und A[r];
      }
    }
  }
}
```

Der Algorithmus sortiert anschließend wie folgt:

5	4	0	1	2	$min = 3$
0	4	5	1	2	$max = 3$
0	4	2	1	5	
0	4	2	1	5	$min = 4$
0	1	2	4	5	$max = 4$
0	1	2	4	5	