Datenstrukturen und Algorithmen: Blatt 2

Bernhard Dietrich (6256800)
Lars Fernhomberg (6256030)
Sebastian Kniesburges (6257120)
Marcus Köthenbürger (6258550)

Aufgabe 1
Wir betrachten den folgenden Sortieralgorithmus, genannt BUBBLE-SORT.

BUBBLE-SORT(A)
```
1     for i←1 to length(a)
2           do for j←length(A) downto i+1
3                 do if A[j]<A[j-1]
5                       then A[j]↔A[j-1]
```

1. Formulieren und beweisen Sie zunächst eine Invariante für die **for**-Schleife in den Zeilen 2-5. Formulieren und beweisen Sie dann eine Invariante für die **for**-Schleife in den Zeilen 1-5. Beweisen Sie die Korrektheit von BUBBLE-SORT mit Hilfe dieser Invarianten.

   Invariante für die Zeilen 2-5: Im Bereich des Arrays von j bis length(A) befindet sich kein kleineres Element als das Element mit dem Index j.

   Beweis:
   Initialisierung: Für j=length(A) kann es oberhalb von j keine kleineren Elemente geben, da das betrachtete Teilarray (j bis length(A) mit j=length(A)) aus nur einem einzigen Element besteht.
   Erhaltung: Ist das Element mit dem Index j kleiner als sein Vorgänger (mit dem Index j-1) (Zeile 3) wechseln beide Elemente den Platz (Zeile 5), so dass es oberhalb von j-1 keine kleineren Elemente mehr geben kann.
   Terminierung: Nach dem letzten Durchlauf der Schleife enthält das Element i das kleinste Element, welches sich im Teilarray (i bis length(A)), welches von der zweiten for-Schleife bearbeitet wird, befindet. Die for-Schleife arbeitet somit korrekt.

   Invariante für die Zeilen 1-5: Das Array ist im Bereich zwischen 1 und i korrekt (aufsteigend) sortiert.

   Beweis:
   Initialisierung: Vor Beginn der Schleife hat die Variable i keinen Wert. Somit existiert auch kein Teilarray, auf dem die Sortierung gemäß der Invariante korrekt sein muss, so dass die Invariante erfüllt ist.
   Erhaltung: Die zweite for-schleife liefert, wie oben gezeigt, das kleinste Element des Arrays im Bereich zwischen i und length(A). Dieses kleinste Element befindet sich nach Durchlauf der zweiten for-Schleife an der Stelle mit dem Index i, so dass die korrekte Sortierung des Arrays zwischen dem 1. und dem i-ten Element gewährleistet ist.
   Terminierung: Nach dem letzten Schleifendurchlauf ist das Array aufsteigend sortiert, da die zweite for-Schleife für jedes Teilarray die kleinsten Elemente geliefert und an die jeweilige Stelle i gesetzt hat. Der komplette Algorithmus arbeitet somit wie spezifiziert.
2. Analysieren Sie die worst-case Laufzeit von BUBBLE-SORT.
   Der worst-Case des BUBBLE-SORT –Algorithmus liegt bei einer umgekehrten Sortierung vor (statt aufsteigend somit absteigend), da sich das kleinste Element jeweils an der Stelle length(A) und somit am Ende des Arrays befindet.

BUBBLE-SORT(A)

1	for i←1 to length(a)	c_1	$n+1$
2	do for j←length(A) downto i+1	c_2	$\frac{n(n+1)}{2}$
3	do if A[j]<A[j-1]	c_3	$\frac{n(n-1)}{2}$
5	then A[j]↔A[j-1]	c_5	$\frac{n(n-1)}{2}$

Es ergibt sich somit:

$$c_1 \cdot (n+1) + c_2 \cdot \left(\frac{n \cdot (n+1)}{2} \right) + (c_3 + c_5) \cdot \left(\frac{n \cdot (n-1)}{2} \right)$$

$$= \left(\frac{c_2 + c_3 + c_5}{2} \right) \cdot n^2 + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_5}{2} \right) \cdot n + c_1$$

$$= a \cdot n^2 + b \cdot n + c$$

Der Algorithmus gehört somit zu $O(n^2)$.

Aufgabe 2

Wir betrachten wieder das Suchproblem. Wir nehmen aber an, dass die Eingabefolge A bereits sortiert ist. Also

Eingabe: Eine sortierte Folge von n Zahlen $(a_1, ..., a_n)$, die in einem Array A gespeichert ist und eine weitere Zahl v.

Ausgabe: Ein Index i, sodass $v = A[i]$ oder ein spezieller Wert NIL, falls v nicht in der Folge A auftaucht.

Schreiben Sie Pseudocode für die so genannte binäre Suche: Bei der binären Suchen wird zunächst geprüft, ob v in der oberen oder unteren Hälfte des Arrays auftauchen kann. Dann wird rekursiv in der passenden Hälfte des Arrays weitergesucht.

1. Schreiben Sie Pseudocode für die binäre Suche.

BINSEARCH(A, START, ENDE, V)

```
1  if ende-start=1
2       then if A[start]=v
3                  return start
4            if A[ende]=v
5                  return ende
6            else
7                  return NIL
8  i ← start + Abrunden((ende-start)/2)
9  ► Überprüfen, ob das aktuelle Element evtl. schon das gesuchte
     ist
10 if A[i] = v
11      then return i
12 ► Überprüfen, in welcher Hälfte sich das geuschte Element
     befinden müsste
13 if A[i] > v
14      then return BinSearch(A, start, i, v)
15 else
16      return BinSearch(A, i, ende, v)
```

In Zeilen 1 bis 7 wird überprüft, ob die Rekursion noch weiter ausgeführt werden kann oder ob das Start- und Endelement direkt nebeneinander liegen. Wenn das Start- und das Endelement direkt nebeneinander liegen sollten, wird überprüft, ob eins dieser Elemente das gesuchte ist (Zeilen 2-5). Sollte dies zutreffen, wird der Index dieses Elements zurückgeben, ansonsten der Wert für die Erfolglose Suche (NIL).
Falls die Rekursion aufrecht erhalten werden kann (sich also noch mindestens ein Element zwischen Start und Ende befindet), wird der Mittelpunkt des durch „Start“ und „Ende“ spezifizierten Teilarrays bestimmt (Zeile 8) und anschließend überprüft, ob dieser Mittelpunkt das gesuchte Element enthält (Zeilen 10-11).
Sollte das gesuchte Element nicht gefunden werden, wird durch den Vergleich in Zeile 13 bestimmt, in welcher Teilhälfte sich das gesuchte Element befindet und diese neue Teilhälfte wird durch eine Rekursion durchsucht: Sollte das Element am Mittelpunkt kleiner als das gesuchte sein, wird die rechte Teilhälfte durchsucht, andernfalls die linke Teilhälfte.
Die vorliegende Pseudocode-Implementation setzt voraus, dass das Array aufsteigend sortiert ist. Sollte es absteigend sortiert sein, müssen die Vergleichsoperatoren dementsprechend angepasst werden.

2. Analysieren Sie die worst-case Laufzeit ihres Algorithmus.
   Der worst-case des vorliegenden Sortieralgorithmus tritt ein, wenn das gesuchte Element sich nicht im Array befindet.
   Das Problem gilt spätestens dann als gelöst, wenn zwischen dem Start- und dem Endelement sich kein Element mehr befindet, so dass keine Rekursion mehr sinnvoll ausgeführt werden kann.

   Wie häufig werden die einzelnen Zeilen des Suchalgorithmus im worst case durchlaufen?
   Alle Zeilen werden fast konstant (zwischen null- und einmal) durchlaufen, so dass die Laufzeit eines Rekursionsaufrufs praktisch konstant ist (in Abhängigkeit von den Vergleichen kann die Laufzeit geringfügig variieren), sofern man die Rekursionsanweisungen in Zeile 14 und 16 außen vor lässt. Bezieht man diese mit ein ergibt sich die Laufzeit eines Rekursionsaufrufs durch: $T\left(n\right)=T\left(\frac{n}{2}\right)+c$

   $$T\left(n\right)=T\left(\frac{n}{2}\right)+c=T\left(\frac{n}{4}\right)+c+c=T\left(\frac{n}{8}\right)+c+c+c=...=T\left(\frac{n}{2^l}\right)+l\cdot c$$

   Sei nun $n=2^k$, so ergibt sich:

   $$T\left(\frac{n}{2^k}\right)+k\cdot c=T\left(1\right)+k\cdot c=c+k\cdot c$$

   Da $n=2^k$ folgt: $T\left(n\right)=c+\log\left(n\right)\cdot c$