

Ausarbeitung des 1. Übungstests WS 2003/04 Gruppe A

1a) Beweisen oder widerlegen Sie die folgenden Behauptungen:

$$f(n) = \Theta(g(n)) \Rightarrow g(n) = \Omega(f(n))$$

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O\left(g\left(\frac{1}{2} \cdot n\right)\right)$$

Lösungsweg I zur ersten Aussage:

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n_0 > 0$$

$$\text{sodass } \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\text{Ziel: } \exists c, n_0 > 0 \text{ sodass } \forall n \geq n_0 : 0 \leq c f(n) \leq g(n)$$

$$\Rightarrow \exists c, n_0 > 0 \text{ sodass } \forall n \geq n_0 : 0 \leq c f(n) \leq g(n) \text{ mit } c = \frac{1}{c_2} \Leftrightarrow$$

$$\Leftrightarrow g(n) = \Omega(f(n))$$

Lösungsweg II zur ersten Aussage:

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

$$\Rightarrow g(n) = \Omega(f(n)) \wedge g(n) = O(f(n))$$

zweite Aussage:

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O\left(g\left(\frac{1}{2} \cdot n\right)\right)$$

Widerlegt durch:

$$f(n), g(n) = 4^n$$

$$f(n) = \Theta(g(n)), 4^n = \Theta(4^n)$$

$$f(n) = O\left(g\left(\frac{1}{2} \cdot n\right)\right)$$

$$4^n = O\left(4^{\frac{1}{2} \cdot n}\right)$$

$$4^n = O(2^n) \rightarrow \text{nicht möglich}$$

Bsp:

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n-1)) \rightarrow \text{Widerlegt durch doppelten}$$

$$\text{Exponent zB: } 4^{4^n}$$

$$\text{Ann.: } g(n-1)$$

$$4^n = \Theta(4^n)$$

$$4^n = O\left(\frac{4^n}{4}\right)$$

$$4^n = O\left(\frac{1}{4} \cdot 4^n\right) \rightarrow \text{wäre konstanter Faktor, damit nicht widerlegt}$$

Widerlegt durch:

$$4^{(4^n)} = O(4^{4^{n-1}})$$

$$= O(4^{(4^n)} \cdot \frac{1}{4}) \rightarrow \text{andere Ordnung}$$

$$= O(4^{\text{sqrt}(4^{4^n})}) \rightarrow \text{bleibt im Exponent und beeinflusst damit das Funktions - Argument)}$$

1b) Zusätzlich zu den in der Vorlesung kennen gelernten Notationen ist eine andere in der Informatik häufig benutzte Notation die **o-Notation**. Sie ist wie folgt definiert:

$$o(g(n)) = \{f(n) \mid (\forall c > 0), (\exists n_0 \geq n_0) : 0 \leq f(n) < c \cdot g(n)\}$$

Beweisen oder widerlegen Sie $f(n) = o(g(n))$ für die folgenden Funktionspaare:

$$f(n) = \frac{1}{2} \cdot \sqrt{n}$$

$$f(n) = 7n$$

$$g(n) = \sqrt{n}$$

$$g(n) = 3n^2$$

klein o bedeutet soviel, wie: $g(n)$ muss an irgendeiner Stelle $f(n)$ „durchstoßen“ bzw. überholen.

Folgende Tabelle bekommt man als **Lösung**:

| f(n) | g(n) | O | o |
|-------------|-------------|----------|----------|
| n | \sqrt{n} | - | - |
| n | n | + | - |
| n | n^2 | + | + |

$$f(n) = \frac{1}{2} \cdot \sqrt{n}$$

$$f(n) = 7n$$

$$g(n) = \sqrt{n}$$

$$g(n) = 3n^2$$

$$f(n) \neq o(g(n))$$

$$f(n) = o(g(n))$$

Beweis für die rechte Behauptung:

$$\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq 7n \leq c \cdot 3n^2 / 3n^2$$

$$\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq \frac{7}{3n} \leq c$$

2a) Verändern Sie den Quicksort-Algorithmus so, dass bei einer Folge mit weniger als 3 Elementen Insertion-Sort verwendet wird. Geben Sie den veränderten Pseudocode für Quick-Insertion-Sort an. Sie können die Funktionen Partition(A,l,r,x) (der aus der Vorlesung bekannte Pseudocode) und Insertion-Sort(A,i,j), die das Verfahren Insertion-Sort im Feld A von einschließlich Index i bis einschließlich Index j durchführt, verwenden, ohne dafür Pseudocode anzugeben.

Lösung:

```
Quicksort(A, l, r):
1: falls r=l+1 {
2:   InsertionSort(A, l, r);
3: }
4: sonst {
5:   x=A[r].key;
6:   p=Partition(A, l, r, x);
7:   Quicksort(A, l, p-1);
8:   Quicksort(A, p+1, r);
9: }
```

2b) Sortieren Sie die Folge <1,2,3,8,6,7,4> mit Ihrem Verfahren. Geben Sie die Zahlenfolge nach jedem kompletten Aufteilungsschritt an.

Lösung: (unterstrichen = Pivot-Element)

1,2,3,8,6,7,4
 1,2,3,4,6,7,8

| | | |
|---------------------|---------------|---------------------|
| | 1,2, <u>3</u> | 6,7, <u>8</u> |
| 1,2 (mit Insertion) | | 6,7 (mit Insertion) |

2c) Ist das Verfahren für die obige Folge schneller oder langsamer als der Original-Quicksort? Wie ist das im allgemeinen Fall? Begründen Sie Ihre Antwort.

Lösung:

Der modifizierte Sortieralgorithmus ist geringfügig schneller, da Quicksort rekursiv ist, dh. bei kleinen Zahlenfolgen benötigt er mehr Zeit als InsertionSort. InsertionSort ist bei kurzen Zahlenfolgen immer schneller als Quicksort.

3) Der abstrakte Datentyp „Stack“ stellt die folgenden Operationen bereit:**S.size()**

gibt die Anzahl der Elemente auf dem Stack zurück

S.push(x)

fügt Element x zum Stack hinzu

S.pop()

gibt das zuletzt eingefügte Element zurück und löscht es vom Stack

Der abstrakte Datentyp „Queue“ funktioniert ähnlich. Die Queue zeigt das gleiche Verhalten bei den Operationen size und push; lediglich bei Q.pop() unterscheiden sich die beiden Datentypen: eine Queue gibt unter den noch enthaltenen Elementen jenes zurück, das als Erstes eingefügt wurde, und löscht es.

Zeigen Sie, wie Sie eine Queue Q mit zwei Stacks S1 und S2 simulieren können. Hinweis: Wenn Sie Elemente von einem Stack „poppen“, erscheinen diese in umgekehrter Reihenfolge. Wenn Sie das wiederholen, haben Sie wieder die ursprüngliche Reihenfolge. Geben Sie Pseudocode für die Operationen Q.size(), Q.push(x) und Q.pop() an.

Lösung:**Stack:**

| |
|---|
| 3 |
| 2 |
| 1 |

push(x) legt ein Element auf den Stack
 pop() liefert das erste Element d. Stacks, bzw. das zuletzt eingefügte
 1,2,3 -> in den Stack, mit 3x pop() -> 3,2,1

Queue: 1,2,3 ->

| | | |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

 -> 1,2,3
Queue mit Stacks simuliert:

| |
|---|
| 3 |
| 2 |
| 1 |

| |
|---|
| 1 |
| 2 |
| 3 |

umfüllen der Zahlen von Stack1 nach Stack2. Dann wird das erste
 Element von Stack2 ausgegeben, was dem letzten von Stack1 entspricht
 -> Eigenschaft des Queues ist erfüllt.

Pseudocode:

```

Q.push(x) {
    s1.push(x);
}
Q.pop() {
    solange(s1.size()>1) {                               // umfüllen des 1. Stack in den 2.
Stack      s2.push(s1.pop());                             // bis auf das letzte Element
    }
    x=s1.pop()                                           // speichern d. letzten Elements in
S1
    solange(s2.size()>0) {                               // zurückfüllen von S2 auf S1
        s1.push(s2.pop());
    }
    retourniere x;
}
  
```

Ausarbeitung des 1. Übungstests SS 2003

1a) Gegen sei die Funktion f wie folgt:

$$f(n) = \begin{cases} 5n^3 + \frac{2n}{3} & n \text{ gerade} \\ \frac{25n^3}{14} + n^2 + \sqrt{8n} & n \text{ ungerade} \end{cases}$$

Kreuzen Sie in der folgenden Tabelle die zutreffenden Felder an:

| $f(n)$ ist | $O(\cdot)$ | $\Omega(\cdot)$ | keines davon |
|-----------------|-------------------------------------|-------------------------------------|--------------------------|
| n^4 | <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| $\frac{1}{n^2}$ | <input type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| n^3 | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| $n^3 \log n$ | <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Lösung:

$$f(n) = \Theta(n^3)$$

da $\left(\frac{2n}{3}\right)$ und $(n^2 + \sqrt{8n})$ schwach wachsend \rightarrow vernachlässigbar

$$\Rightarrow \text{es bleibt } n^3 \Rightarrow \Theta(n^3)$$

Bsp:

$$f(n) = \begin{cases} n \\ n^3 \end{cases}$$

$$g(n) = n^2 \Rightarrow \text{weder } O \text{ noch } \Omega$$

1b) Welche Laufzeiten in Θ -Notation haben die folgenden Algorithmen abhängig von n ?

(i)

$$t = n^2$$

wiederhole // $\frac{n^2}{2}$ Durchläufe

$$k = nk$$

$$t = t - 2$$

bis $t \leq 3$

$$\underline{\underline{\Theta(n^2)}}$$

(ii)

```

s = 1
solange s < 2n - 3 {
    s = 2 * s
    t = n - 5
}

```

$\Theta(\log n)$ // durch verdoppeln \Rightarrow 2er Potenzen \Rightarrow log Dualis

(iii)

```

x = 2
z = n - 1
wiederhole
    für y = 2, 3, ..., z {
        v = 3z
    }
    x = 3x
bis x ≥ n

```

innere für - Schleife läuft ca. n - mal $\Rightarrow \Theta(n)$
 äußere wiederhole - Schleife läuft $\Theta(\log n)$,
 da der Wert x durch eine
 Multiplikation verändert wird

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

(iv)

```

m = 2n - 3
solange m > 0 {
    a = n + 26
    wiederhole
        a = ⌊ a / 2 ⌋
    bis a < 1
    m = ⌊ m / 3 ⌋
}

```

innere wiederhole - Schleife halbiert a bis zu einem Grenzwert
 $\Rightarrow \Theta(\log n)$
 äußere solange - Schleife drittelt den Wert bis zu einem Grenzwert
 $\Rightarrow \Theta(\log n)$
 äußere Schleife ruft die innere Schleife $\log n$ - mal auf, welche
 $\log n$ Iterationen durchläuft $\Rightarrow \log n \cdot \log n$

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

Faustregeln für die Laufzeitfunktion:

| Konstante verändert n durch | die Laufzeitfunktion ist |
|---|--------------------------|
| Addition, Subtraktion | linear |
| Division, Multiplikation zu einen Grenzwert | logarithmisch |
| Exponent | exponentiell |

2a) Geben Sie ein Beispiel für eine Zahlenfolge mit 7 Elementen an, für welche die Laufzeit des einfachen Quicksort-Verfahrens, bei dem das Trennelement jeweils am Ende der Teilfolge entnommen wird, für *absteigendes* Sortieren zum Worst Case entartet.

Lösung:

| | |
|---------------------------|---------------|
| 1,2,3,4,7,8,13, 42 | n |
| 1,2,3,4,7,8, 13 | n-1 |
| 1,2,3,4,7, 8 | n-2 |
| 1,2,3,4, 7 | n-3 |
| 1,2,3, 4 | n-4 |
| 1,2, 3 | n-5 |
| 1, 2 | n-6 |
| 1 | 1 |
| | <hr/> |
| | $\Theta(n^2)$ |

Worst-Case: Zahlenfolge, die absteigend sortiert ist, zB 7,6,5,4,3,2,1

2b) Gegen sei die Zahlenfolge (5,8,10,2,6,3,12,11).

Sortieren Sie diese Zahlenfolge *aufsteigend* mittels Quicksort, wobei das Trennelement jeweils am Ende einer Teilfolge entnommen wird. Geben Sie die Zahlenfolge nach jedem kompletten Aufteilungsschritt an.

Lösung:

5,8,10,2,6,3,12,**11**

5,8,10,2,6,**3**

2,**3**,10,5,6,8

12

10,5,6,**8**

6,5,10,**8**

6,5,**8**,10

6,5

5,6

10

2

3a) Schreiben Sie einen Algorithmus in Pseudo-Code, der einen gegebenen binären Baum rekursiv durchmustert und die Anzahl Blätter zurückliefert.

Lösung:

```
blätterzahl(w)
  blätter = blätterzahl(w.links)
  blätter = blätter + blätterzahl(w.rechts)
  falls (blätterzahl = 0) {
    retourniere 1;
  }
  retourniere blätter;
```

andere Lösung:

```
c=0
blätterzahl(w)
  falls w != NIL {
    blätterzahl(w.links);
    blätterzahl(w.rechts);
    falls (w.links = NIL und w.rechts = NIL) {
      c=c+1;
    }
  }
  retourniere c;
```

Lösung von nex aus dem Forum: (Danke an dieser Stelle)

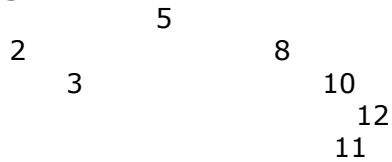
```
blätterzahl(w) {
  falls w == null { // tolle abfrage, die nützen wir im folgenden gleich
  aus
    retourniere 0;
  }
  blätter = blätterzahl(w.links) + blätterzahl(w.rechts);
  retourniere min(1, blätterzahl); // *)
}
```

*) falls wir in den kindern 0 blätter gezählt haben, ist w selbst ein blatt, d.h. wir geben dann nicht 0 zurück, sondern 1. wenn wir mehr als 1 blatt gezählt haben, geben wir diese zahl zurück. also minimum von 1 und blätter.

3b) Gegeben sei die Zahlenfolge (5,8,10,2,3,12,11).

Zeichnen Sie den natürlichen binären Suchbaum, der sich ergibt, wenn die Zahlen in der angegebenen Reihenfolge in einen anfangs leeren Baum eingefügt werden. Eine Zeichnung des Endergebnisses reicht hier.

Lösung:



3c) Nun wird aus der gleichen Zahlenfolge ein AVL-Baum aufgebaut. Zeichnen Sie diesen nach jedem Einfügeschritt.

