

Algorithms & Data Structures Solutions - SoSe 24

Igor Dimitrov

2024-04-22

Table of contents

Preface	3
1 Blatt 01	4
1.1 Aufgabe 2	4
1.2 Aufgabe 3	5
1.2.1 a)	5
1.2.2 b)	6
1.2.3 c) Falsch:	6
1.2.4 d) Falsch:	6
1.2.5 e) Falsch:	6
1.2.6 f)	7
1.3 Auafgabe 4	7
1.3.1 a)	7
1.3.2 b)	7
1.3.3 c)	8
1.3.4 d)	9

Preface

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

1 Blatt 01

1.1 Aufgabe 2

a)

$$\begin{aligned}\log(n!) &= \log\left(\prod_{i=1}^n i\right) && \text{(Def } n!) \\ &= \sum_{i=1}^n \log(i) && \text{(Eig } \log(\bullet)) \\ &\leq \sum_{i=1}^n \log(n) && \text{(Eig } \log(\bullet)) \\ &= n \log(n)\end{aligned}$$

Wähle nun $c_0 := 1$ und $n_0 := 1$. Es folgt somit:

$$\begin{aligned}\log(n!) &\leq 1 \cdot n \log(n), \quad \forall n \geq 1 \\ \Leftrightarrow \log(n!) &\in \mathcal{O}(n \log(n))\end{aligned} \quad \text{(Def } \mathcal{O})$$

b)

Zuerst bemerken wir die folgende Eigenschaft

$$\begin{aligned}n \log(n) &\leq c \log(n!) \\ \Leftrightarrow c &\geq \frac{n \log(n)}{\log(n!)} \\ &= \frac{n \log(n)}{\sum_{i=1}^n \log(i)}\end{aligned}$$

Wir definieren die Folge:

$$\begin{aligned}
c(n) &:= \frac{n \log(n)}{\sum_{i=1}^n \log(i)} \\
&= \frac{\overbrace{\log(n) + \dots + \log(n)}^{n\text{-mal}}}{\log(1) + \dots + \log(n)}
\end{aligned}$$

Wir behaupten ohne Beweis, dass $c(n)$ eine monoton fallende Folge ist. D.h. es gilt:

$$c(n) \leq c(m), \quad \forall n \geq m$$

Setze nun $n_0 := 10, c_0 := c(10) = \frac{10 \log(10)}{\sum_{i=1}^{10} \log(i)}$. Somit folgt:

$$\begin{aligned}
n \log(n) &\leq \left(\frac{n \log(n)}{\sum_{i=1}^n \log(i)} \right) \log(n!) \\
&= c(n) \log(n!) \\
&\leq c_0 \log(n!), \quad \forall n \geq n_0 = 10
\end{aligned}
\tag{c(n) monoton fallend}$$

1.2 Aufgabe 3

1.2.1 a)

Da $f_1 \in \mathcal{O}(g_1)$ und $f_2 \in \mathcal{O}(g_2)$ existieren n_1, n_2, c_1, c_2 s.d:

$$\begin{aligned}
f_1(n) &\leq c_1 g_1(n), \quad \forall n \geq n_1 \\
f_2(n) &\leq c_2 g_2(n), \quad \forall n \geq n_2
\end{aligned}$$

Setze $c_0 := \max\{c_1, c_2\}, n_0 := \max\{n_1, n_2\}$. Dann gilt

$$\begin{aligned}
(f_1 + f_2)(n) &= f_1(n) + f_2(n) \\
&\leq c_1 g_1(n) + c_2 g_2(n), \quad \forall n \geq n_0 & (n \geq n_0 \Rightarrow n \geq n_1 \wedge n \geq n_2) \\
&\leq c_0 g_1(n) + c_0 g_2(n), \quad \forall n \geq n_0 \\
&= c_0 (g_1 + g_2)(n) \quad \forall n \geq n_0 \\
\iff f_1 + f_2 &\in \mathcal{O}(g_1 + g_2) & (\text{Def } \mathcal{O})
\end{aligned}$$

1.2.2 b)

mit $f_1 \in \Theta(g_1)$, $f_2 \in \Theta(g_2)$ existieren $a_1, a_2, b_1, b_2, n_1, n_2$, s.d.:

$$\begin{aligned}a_1 f_1(n) &\leq g_1(n) \leq a_2 f_1(n), \forall n \geq n_1 \\b_1 f_2(n) &\leq g_2(n) \leq b_2 f_2(n), \forall n \geq n_2\end{aligned}$$

Setze $n_0 := \max\{n_1, n_2\}$, $c_1 := a_1 b_1$, $c_2 := a_2 b_2$. Dann gilt:

$$c_1(f_1 f_2)(n) = a_1 f_1(n) b_1 f_2(n) \leq (g_1 g_2)(n) \leq a_2 f_1(n) b_2 f_2(n) = c_2(f_1 f_2)(n), \quad \forall n \geq n_0$$

Somit $f_1 f_2 \in \Theta(g_1 g_2)$.

1.2.3 c) Falsch:

Betrachte $f(n) := n$ und $g(n) := 10n$. Offensichtlich gilt $f \in \Omega(g)$ mit $c_0 := 1/10, n_0 := 1$. Aber $2^n \notin \Omega(2^{10n})$, da 2^n langsamer als 2^{10n} wächst. (Setze z.B. $2^n := x$. Dann $2^{10n} = (2^n)^{10} = x^{10}$, und x^{10} ist offensichtlich schneller als x)

1.2.4 d) Falsch:

Sei $g(n) := 2^n$. Dann $f(n) = g(2n) = 2^{2n} = (2^n)^2$. $(2^n)^2$ ist offensichtlich schneller als 2^n

1.2.5 e) Falsch:

Seien $f(n) := n^2$, $f_1(n) := n^3$, $f_2(n) := n$. Es gilt:

$$\begin{aligned}f &\in \mathcal{O}(f_1) & (n^2 \in \mathcal{O}(n^3)) \\f_1 &\in \Omega(f_2) & (n^3 \in \Omega(n))\end{aligned}$$

aber

$$f \notin \mathcal{O}(f_2) \qquad (n^2 \notin \mathcal{O}(n))$$

1.2.6 f)

Es gilt:

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f(n)}{f_2(n)} &= \lim_{n \rightarrow \infty} \left(\frac{f(n)}{f_1(n)} \cdot \frac{f_1(n)}{f_2(n)} \right) \\ &= \lim_{n \rightarrow \infty} \left(\frac{f(n)}{f_1(n)} \right) \cdot \lim_{n \rightarrow \infty} \left(\frac{f_1(n)}{f_2(n)} \right) \\ &= 0 \cdot c, \text{ fuer ein } c \quad (f \in o(f_1), f_1 \in \mathcal{O}(f_2)) \\ &= 0 \\ \Leftrightarrow f &\in o(f_2) \quad (\text{Def } o)\end{aligned}$$

Wobei wir die alternativen Definitionen von $o(\bullet)$ und $\mathcal{O}(\bullet)$ benutzt haben.

1.3 Aufgabe 4

1.3.1 a)

$\mathcal{O}(n^2 \log(n))$:

```
read(n) //input
for i := 1 to n :
  for j := 1 to n:
    k := 1
    // O(log(n))
    while (k < n) :
      k := 2 * k
```

1.3.2 b)

$\mathcal{O}((\log(n))^2)$:

```
read(n) //input
i := 1
while (i < n) :
  j := 1
  while (j < n) :
    j := 2 * j
```

```
i := 2 * i
```

1.3.3 c)

Wir ‘simulieren’ Exponentiation durch einzelne Additionsoperationen. Somit ist n^n in n^n Additionen berechnet - Python Implementierung:

```
def add(n, m) :  
    if m == 0 : return n  
    return 1 + add(n, m - 1)  
  
def mult(n, m) :  
    if m == 0 : return 0  
    return add(n, mult(n, m - 1))  
  
def exp(n, m) :  
    if m == 0 : return 1  
    return mult(n, exp(n, m - 1))  
  
def f(n) : return exp(n, n)
```

Wir testen diese Funktion fuer einige Werte:

```
for i in range(5) :  
    print(f(i))
```

```
1  
1  
4  
27  
256
```

Alternativ betrachte folgende rekursive Funktionsdefinition:

```
function recursiveLoops(n : Nat, m : Nat) :  
    if m > 0 then :  
        for i = 1 ... n do :  
            recursiveLoops(n, m - 1)
```

Dann erzeugt der Aufruf `recursiveLoops(n, n)` eine Anzahl von $\mathcal{O}(n^n)$ rekursiven Aufrufe.

1.3.4 d)

$\Theta 2^n$ - Wir 'simulieren' binäres Zählen:

```
read(n)
base := 0
count := 0
k := 1
// invariant: k == 2^b, count < k
while (base < n) :
    k := 2 * k
    base := base + 1
    while (count < k) :
        count := count + 1
    // post-condition: count == k
//post-condition b == n => count == 2^n
```

Python Implementierung:

```
def binary_count(n) :
    base = count = 0
    k = 1
    while (base < n) :
        k = 2 * k
        base = base + 1
        while (count < k) :
            count = count + 1
    return count
```

Wir testen diese Funktion fuer einige Werte. Das Ergebniss ist die Anzahl der Schritte fuer die jeweilige Eingabe:

```
for i in range(11) :
    print(binary_count(i))
```

```
0
2
4
8
16
32
64
```

128
256
512
1024

Alternativ:

```
function f(n) :  
    if n == 0 : return 1  
    return f(n - 1) + f(n - 1)
```

Diese rekursive Funktion ruft sich selbst zweimal fuer jeden Wert von n auf, was zu einer Laufzeit von 2^n fuehrt.