Algorithms & Data Structures Solutions - SoSe 24

Igor Dimitrov

2024-04-22

Table of contents

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)

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

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

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

b)

Zuerst bemerken wir die folgende Eigenschaft

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

Wir definieren die Folge:

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

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

$$c(n) \le c(m), \quad \forall n \ge m$$

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

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

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:

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

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

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

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.:

$$a_1 f_1(n) \leq g_1(n) \leq a_2 f_1(n), \forall n \geq n_1$$
$$b_1 f_2(n) \leq g_2(n) \leq b_2 f_2(n), \forall n \geq n_2$$

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} waechst. (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:

$$f \in \mathcal{O}(f_1) \qquad (n^2 \in \mathcal{O}(n^3))$$
$$f_1 \in \Omega(f_2) \qquad (n^3 \in \Omega(n))$$

aber

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

1.2.6 f)

Es gilt:

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

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

1.3 Auafgabe 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' binaeres Zaehlen:

```
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.

2 Blatt 2

2.1 Aufgabe 1

2.1.1 1

Rekursionsbaum des Aufrufs `sum(<1, 2, 3, 4, 5, 6, 7, 8>)`

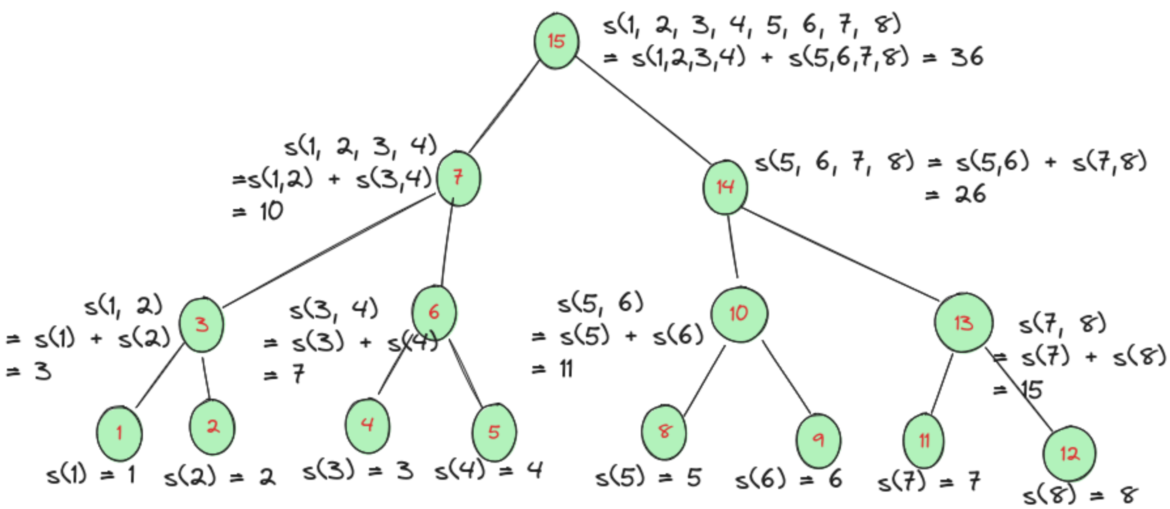

Figure 2.1: Rekursionsbaum

Die Nummerierung der Knoten entspricht der Berechnungsreihenfolge.

2.1.2 2

Eine nicht-konstante Laufzeit ensteht, falls uebergebene arrays auf den Stack des Funktionsaufrufs kopiert werden muessen.

Wenn eine gegebene Implementierung der Programmiersprache folgende zwei Eigenschaften aufweist, kann dies vermieden werden:

- Die Groesse eines Arrays ist immer als zusaetzliche Information beinhaltet.

- Die Funktionsaufrufe werden per-default als **call by reference** realisiert statt **call by value**.

So wuerde fuer einen existierenden Array A : Array$[0..n-1]$ of $\mathbb{N}$ der allgeimeiner Ausdruck $A[l..k]$ einen Array liefern, dessen Anfang-position im Speicher und Groesse durch Pointer-arithmetik, bzw durch den Ausdruck $k-l+1$ bestimmt werden koennen. Das sind nur zwei Grundoperationen, und somit $\mathcal{O}(1)$

Da die Uebergabe der Arrays per Referenz stattfindet, wuerden die Aufrufe `sum(A[0..m-1])` und `sum(A[m..n-1])` nur konstante Zeit bei der Initialisuerungen auf ihren Function call-stacks benoetigen.

2.1.3 3

a) Die Laufzeit erfuellt die Rekurrenzgleichung:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 1 + 2 \cdot T(\frac{n}{2}) && \text{(fuer } n = 2^k > 1) \end{aligned}$$

b) Da, die Eingabe bei jedem Aufruf halbiert wird ist die Tiefe des Rekurrenzbaums (Figure 2.1) $k = \log_2(n)$. Dieser Baum ist vollstaendig binaer, deshalb enthaelt jede Tiefe i genau 2^i Knoten, fuer $i = 0 \dots k$. Somit betreagt die Gesamtzahl der Knoten:

$$\begin{aligned} N &= \sum_{i=0}^{\log_2(n)} 2^i \\ &= 2^{\log_2(n)+1} - 1 && \text{(Geom Reihe)} \\ &= 2n - 1 \end{aligned}$$

Bei jedem Knoten wird eine konstante Anzahl von Additions- & Zuweisungsoperationen durchgefuehrt, und das Ergebnis zur aufrufenden Funktion zurueckgegeben. Somit ist die Laufzeit proportional zur Anzahl der Knoten, die wir in der vorangehenden Diskussion berechnet haben, d.h. $T(n) = c_1 n + c_2$. Dann gilt offensichtlich $T(n) = \Theta(n)$

2.2 Aufgabe 2

2.2.1 a)

$$
\begin{aligned}
& a = 1, \\
& c = \tilde{c}, \\
& d = 1 < 2 = b \\
\Rightarrow & T(n) \in \Theta(n) && \text{(Fall } d < b \text{ des MT)}
\end{aligned}
$$

2.2.2 b)

$$
\begin{aligned}
& a = 1, \\
& c = 4, \\
& d = 9 > 3 = b \\
\Rightarrow & T(n) \in \Theta(n^{\log_3(9)}) = \Theta(n^2) && \text{(Fall } d > b \text{ des MT)}
\end{aligned}
$$

2.2.3 c)

Der Ausdruck $C(n/4) + n + 6$ kann asymptotisch als $C(n/4) + n$ kann vereinfacht werden, da Addition mit konstante vernachlaessigt werden kann. Somit:

$$
\begin{aligned}
& a = 1, \\
& c = 1, \\
& d = 1 < 4 = b \\
\Rightarrow & T(n) \in \Theta(n) && \text{(Fall } d < b \text{ des MT)}
\end{aligned}
$$

2.2.4 d)

In c) wurde gezeigt, dass $C(n) \in \Theta(n)$. Somit kann $C(n)$ fuer asymptotische Zwecke durch $c \cdot n$ erzetzt werden. Dann gilt:

$$T(n) = c \cdot n + 4D(\frac{n}{4})$$

und somit:

$$
\begin{aligned}
&a = 1, \\
&d = 4 = 4 = b \\
\Rightarrow &T(n) \in \Theta(n \log n) && \text{(Fall } d = b \text{ des MT)}
\end{aligned}
$$

2.3 Aufgabe 3