Übungsblatt 1

Abgabe via Moodle.
Deadline Fr. 12th Mai

Aufgabe 1 **(**8 ***Punkte:*** $1+1+1+2+1+2$**)**

Beweisen oder widerlegen Sie:

a) $n^3 = \mathrm{O}\left(n^3 - 2n^2\right), \quad n^{\frac{3}{2}} = \mathrm{O}\left(n^2\right), \quad n^3 = \mathrm{O}\left(10^6 n^2\right)$.

b) $2^{n+1} = \mathrm{O}\left(2^n\right), \quad 5\cos(n) + n = \mathrm{O}\left(1\right), \quad (n+1)! = \mathrm{O}\left(n!\right).$

c) $\log_2 n = \Theta(\log_{10} n), \quad 2^{\log_a(n)} = \Theta\left(2^{\log_b(n)}\right)$ für $a \neq b$

d) $f(n)+g(n) = \mathrm{O}\left(\min(f(n), g(n))\right), \quad f(n)+g(n) = \mathrm{O}\left(\max(f(n), g(n))\right), \quad f(n)g(n) = \mathrm{O}\left(f(n)g(n)\right)$

e) $f(n) = \mathrm{O}\left(n\right) \Rightarrow 2^{f(n)} = \mathrm{O}\left(2^n\right), \quad f(n) = \mathrm{O}\left(n\right) \Rightarrow f(n)^2 = \mathrm{O}\left(n^2\right)$

f) Auf der Menge der asymptotisch positiven Funktionen ist

$$f \sim_\Theta g \text{ gdw. } f \in \Theta(g)$$

eine Äquivalenzrelation.

Lösung:

a)
- Die Behauptung $n^3 = \mathrm{O}\left(n^3 - 2n^2\right)$ gilt: wähle $c = 2, n_0 = 4$.
- Die Behauptung $n^{\frac{3}{2}} = \mathrm{O}\left(n^2\right)$ gilt: wähle $c = 1, n_0 = 1$
- Die Behauptung $n^3 = \mathrm{O}\left(10^6 n^2\right)$ gilt nicht. Beweis durch Widerspruch. Angenommen, es gilt $n^3 = \mathrm{O}\left(10^6 n^2\right)$. Dann $\exists c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : n^3 \leq c \cdot 10^6 n^2$. Im Anschluss gilt nach teilen durch n^2: $n \leq 10^6 c \forall n \geq n_0$. ↯

b)
- Die Behauptung $2^{n+1} = \mathrm{O}\left(2^n\right)$ gilt: $2^{n+1} = 2 \cdot 2^n$, wähle also $n_0 = 1, c = 2$
- Die Behauptung $5\cos(n) + n = \mathrm{O}\left(1\right)$ gilt nicht: $5\cos(n) \leq 5 \ \forall n \in \mathbb{N}$, womit gilt $5\cos(n) = \mathrm{O}\left(1\right)$, jedoch ist der zweite Teil nicht beschränkt und somit der Ausdruck insgesamt nicht in $\mathrm{O}\left(1\right)$.
- Die Behauptung $(n+1)! = \mathrm{O}\left(n!\right)$ gilt nicht. Beweis durch Widerspruch. Angenommen, es gilt $(n+1)! = \mathrm{O}\left(n!\right)$. Dann $\exists c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : (n+1)! \leq c \cdot n!$. Also gilt nach teilen durch $n!$: $n + 1 \leq c \forall n \geq n_0$ ↯

c)
- Die Behauptung $\log_2 n = \Theta(\log_{10} n)$ gilt, da $\log_2 n = \frac{\log_{10} n}{\log_{10} 2}$
- Die Behauptung $2^{\log_a n} = \Theta\left(2^{\log_b n}\right)$ für $a \neq b$ gilt nicht: wähle $a = 2$, $b = 10$. Dann ist $2^{\log_a n} = n$ und $2^{\log_b n} = 2^{\log_{10} n} = 2^{\log_2 n * \log_{10} 2} = n^{0.30\ldots}$. Analog zu a) oder b) zeigt man $n \notin \Theta\left(n^{\log_{10} 2}\right)$

d)
- Die Behauptung $f(n) + g(n) = \mathrm{O}\left(\min(f(n), g(n))\right)$ gilt nicht. Gegenbeispiel: $f(n) := n$, $g(n) := \sqrt{n}$.
- Die Behauptung $f(n)+g(n) = \mathrm{O}\left(\max(f(n), g(n))\right)$ gilt, da $f(n)+g(n) \leq 2\max(f(n), g(n))$.
- Die Behauptung $f(n) \cdot g(n) = \mathrm{O}\left(f(n) \cdot g(n)\right)$ gilt: wähle $c = 1, n_0 = 1$.

e)
- Die Behauptung $f(n) = \mathrm{O}(n) \Rightarrow 2^{f(n)} = \mathrm{O}(2^n)$ gilt nicht. Gegenbeispiel: $f(n) = 2n$. Dann $f(n) \in \mathrm{O}(n)$, aber $4^n \notin \mathrm{O}(2^n)$
- Die Behauptung $f(n) = \mathrm{O}(n) \Rightarrow f(n)^2 \in \mathrm{O}(n^2)$ gilt. Beweis: $f(n) = \mathrm{O}(n) \Rightarrow \exists c > 0 \exists n_0 > 0 : f(n) \leq cn \quad \forall n \geq n_0$. Quadrieren liefert: $\exists c > 0 \exists n_0 > 0 : f(n)^2 \leq c^2 n^2 \quad \forall n \geq n_0$. Wähle also $\tilde{n_0} = n_0$ und $\tilde{c} = c^2$ und erhalte mit diesen Konstanten die Behauptung.

f) Behauptung gilt. Beweis:

- *Reflexivität*: zu zeigen $f \sim_\Theta f$.
  Beweis: $c_1 f(n) \leq f(n) \leq c_2 f(n)$ ist trivialerweise erfüllt für $c_1 = c_2 = 1$, und n_0 beliebig.
- *Symmetrie*: zu zeigen $f \sim_\Theta g$ gdw. $g \sim_\Theta f$.
  Beweis: Es sei $f \sim_\Theta g$ mit den Konstanten c_1, c_2, n_0. D.h. es gilt $c_1 g(n) \leq f(n) \leq c_2 g(n)$. Umstellen liefert uns: $\frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$. Also ist $g(n) = \Theta(f(n))$ mit den Konstanten $\frac{1}{c_2}, \frac{1}{c_1}, n_0$. Also $g \sim_\Theta f$
- *Transitivität*: zu zeigen $f \sim_\Theta g$ und $g \sim_\Theta h$ impliziert $f \sim_\Theta h$
  Beweis: Es sei $f \sim_\Theta g$ und $g \sim_\Theta h$.
  Also gilt $c_1 g(n) \leq f(n) \leq c_2 g(n)$ und $d_1 h(n) \leq g(n) \leq d_2 h(n)$ für entsprechende Konstanten und n_0. Daraus folgt: $c_1 d_1 h(n) \leq f(n) \leq c_2 d_2 h(n) \, \forall n \geq n_0$, was zu zeigen war.

Aufgabe 2 (*Zeitaufwand von Algorithmen,* $2 + 1 + 1 + 2$ *Punkte*)

Gehen Sie davon aus, dass Sie mit einem Rechner arbeiten, der nur die Rechenoperationen + und −, sowie die Vergleiche $<, \leq, =, >, \geq$ in Zeit $\Theta(1)$ ausführen kann. Zuweisungen an Variablen oder an Einträge von Arrays sind ebenfalls möglich und benötigen ebenfalls $\Theta(1)$ Zeit; ebenso auch das Auslesen von Variablen und Array-Einträgen. Ansonsten unterstützt der Rechner nur die übliche Ablaufsteuerung (Schleifen, Fallunterscheidung, Unterprogrammaufruf) und *keine* weiteren Operationen. Der *einzige* verfügbare Basisdatentyp ist zudem die Menge $\mathbb{N}_{\geq 0}$. Im folgenden müssen Sie nun einige Algorithmen für diesen Rechner angeben. Dabei spielt nur die Laufzeit eine Rolle. Ihr Algorithmus könnte z.B. immer den gleichen Wert als Ergebnis zurückgeben. Wichtig ist nur, dass er dabei das gewünschte Laufzeitverhalten aufweist.

1. Geben Sie einen Algorithmus an, der als Eingabe eine Zahl $n \in \mathbb{N}_{\geq 0}$ hat und $\Theta((\log n)^2)$ Zeit benötigt.
2. Geben Sie einen Algorithmus an, der als Eingabe eine Zahl $n \in \mathbb{N}_{\geq 0}$ hat und $\Theta(n^2 \log n)$ Zeit benötigt.
3. Geben Sie einen Algorithmus an, der als Eingabe eine Zahl $n \in \mathbb{N}_{\geq 0}$ hat und $\Theta(2^n)$ Zeit benötigt.
4. Geben Sie einen Algorithmus an, der als Eingabe eine Zahl $n \in \mathbb{N}_{\geq 0}$ hat und $\Theta(n^n)$ Zeit benötigt.

Zeigen Sie jeweils, dass Ihr Algorithmus tatsächlich die geforderte Laufzeit hat.
Hinweis: Soweit sinnvoll können Sie dabei die Erkenntnisse aus Aufgabe 1 verwenden.

Lösung:

1. Ein möglicher Algorithmus:

```
1: procedure A(n: ℕ≥0)
2: a := 1 : ℕ≥0
3: b := 1 : ℕ≥0
4: while a < n do
5:     a := a + a
6:     b = 1
```

7: **while** $b < n$ **do** $b := b + b$
8: **return**

In der While-Schleife hat a nach i Schritten den Wert 2^i. Also gilt nach $i \geq \log n$ Schritten $a = 2^i \geq 2^{\log n} = n$, was zum Abbruch der While-Schleife führt. Das analoge Argument gilt für b in der inneren While-Schleife. Insgesamt werden also $\Theta((\log n)^2)$ Additionen, Zuweisungen und Vergleiche ausgeführt. Die Gesamtlaufzeit ist also in $\Theta((\log n)^2)$.

2. Ein möglicher Algorithmus:

1: **procedure** $B(n\colon \mathbb{N}_{\geq 0})$
2: **for** $i_1 := 1, \ldots, n$ **do**
3: **for** $i_2 := 1, \ldots, n$ **do**
4: $a := 1 : \mathbb{N}_{\geq 0}$
5: **while** $a < n$ **do** $a := a + a$
6: **end for**
7: **end for**
8: **return**

Die Schachtelung von vier For-Schleife bewirkt, dass die innere While-Schleife n^2-mal ausgeführt wird. Die For-Schleifen selbst verursachen dabei einen Zeitaufwand von $\Theta(n^2)$ Die Zeitkomplexität der While-Schleife liegt, wie bereits in **??** gezeigt, in $\Theta(\log n)$. Insgesamt ergibt sich die gewünschte Laufzeit von $\Theta(n^2 \log n)$.

3. Ein möglicher Algorithmus:

1: **procedure** $C(n\colon \mathbb{N}_{\geq 0})$
2: **if** $n = 0$ **then return**
3: $C(n-1)$
4: $C(n-1)$
5: **return**

Ein Aufruf von C verursacht einen gewissen eigenen Aufwand zuzuzüglich dem Aufwand der Kindaufrufe, also insgesamt den Zeitaufwand

$$T_{\text{gesamt}}(0) = T_{\text{eigen}}, \quad T_{\text{gesamt}}(n) = T_{\text{eigen}} + 2T_{\text{gesamt}}(n-1)$$

Der Eigenaufwand ist aber offenbar in $\Theta(1)$, da er nicht von n abhängt. D.h. es gibt eine Konstante $c > 0$ mit $T_{\text{eigen}} \leq c$. Dann gilt

$$\begin{aligned} T_{\text{gesamt}}(n) &\leq c + 2T_{\text{gesamt}}(n-1) \\ &\leq c + 2(c + 2T_{\text{gesamt}}(n-2)) \\ &= c + 2c + 4T_{\text{gesamt}}(n-2)) \\ &\leq \cdots \leq c + 2c + \ldots 2^n c \\ &= c\sum_{i=0}^{n} 2^i = c2^{n+1} - c \end{aligned}$$

Also gilt $T_{\text{gesamt}}(n) = \mathrm{O}\left(2^{n+1}\right) = \mathrm{O}\left(2^n\right)$. Analog zeigt man $T_{\text{gesamt}}(n) = \Omega(2^n)$, in diesem Fall mit einer Abschätzung nach unten. Allerdings bekommt man dabei

$$T_{\text{gesamt}}(n) \geq c' \sum_{i=0}^{n} 2^i = c'2^{n+1} - c'$$

für ein geeignetes $c' > 0$, was aber nicht ganz zur Definition des Ω-Symbols passt. Allerdings gibt es eine Konstante $c'' > 0$ mit $c'2^{n+1} - c' \geq c''2^{n+1}$, was dieses Problem löst. Ein Aufruf von $C(n)$ hat also einen Zeitaufwand von $\Theta(2^n)$.

4. Ein möglicher Algorithmus (rekursiver Algorithmus mit "‘Wrapper"’-Prozedur, wird benutzt weil die Rekursion eine "‘hässliche"’ Parameterliste erfordert):

```
1: procedure D(n: ℕ_≥0)
2: D_rec(n, n)
3: return
```

Hilfsprozedur, in der die Rekursion stattfindet:

```
1: procedure D_rec(n, k: ℕ_≥0)
2: if n ≤ 1 then return
3: if k = 0 then return
4: for i := 1, ..., n do D_rec(n, k − 1)
5: return
```

Abgesehen von dem Aufwand, der durch rekursive Aufrufe verursacht wird, entsteht in *D_rec* höchstens konstanter Zeitaufwand. Sei $n > 1, k > 0$. Dann gilt für den Zeitaufwand $T(n,k)$ eines Aufrufes *D_rec*(n,k) also $T(n,k) \leq c + nT(n,k-1)$ und $T(n,0) \leq c$ für ein geeignetes $c > 0$. Damit gilt

$$\begin{aligned} T(n,n) &\leq c + nT(n,n-1) \\ &\leq c + n\big(c + nT(n,n-2)\big) \\ &= c + nc + n^2 T(n,n-2) \\ &\leq \cdots \leq c + nc + n^2 c + \cdots + n^{n-1}c + n^n c \\ &= c\sum_{i=0}^{n} n^i = c\,\frac{n^{n+1}-1}{n-1} \leq 2c\,\frac{n^{n+1}-1}{n} \leq 2cn^n \end{aligned}$$

für eine Konstante $c > 0$ und ausreichend große n. Analog zeigt man

$$T(n,n) \geq c'\,\frac{n^{n+1}-1}{n-1}$$

für eine Konstante $c' > 0$ und schätzt ab

$$T(n,n) \geq c'\,\frac{n^{n+1}-1}{n-1} \geq c'\,\frac{n^{n+1}-1}{n} = c'n^n - c'\frac{1}{n} \geq c'n^n - c'\,.$$

Es gibt aber eine weitere Konstante $c'' > 0$ mit $c'n^n - c' > c''n$. Also ist insgesamt $T(n,n) = \Theta(n^n)$, was der Zeitaufwand ist, den ein Aufruf von $D(n)$ verursacht.

Aufgabe 3 (*Finden einer fehlenden ganzen Zahl,* $4+1+1$ *Punkte*)

Ein Feld $A[0..n-2]$ enthält alle ganzen Zahlen von 0 bis $n-1$ außer einer. In der vorliegenden Problemstellung können wir auf die vollständige ganze Zahl nicht mit einer einzigen Operation zugreifen. Die Elemente von A sind binär dargestellt und die einzige Operation, die wir benutzen können, um auf sie zuzugreifen, ist "rufe das j-te Bit von $A[i]$ auf". Dies benötigt konstante Zeit.
Gehen Sie dabei davon aus, dass $n = 2^k$ für ein $k \in \mathbb{N}$ und die Codierung pro Zahl k Bits verwendet, also führende Nullen vorhanden sind. Weiter können Sie davon vereinfachend davon ausgehen, dass die Zahlen aufsteigend sortiert sind.

1. Geben Sie einen rekursiven Algorithmus an, der die fehlende ganze Zahl immer noch in Zeit $\mathrm{O}(n)$ findet.
2. Geben Sie für den Algorithmus eine Rekursion für die Laufzeitabschätzung an.
3. Lösen Sie die Rekursion und beweisen Sie damit, dass ihr Algorithmus in Zeit $\mathrm{O}(n)$ läuft.

Lösung:

1. Man startet mit dem k-ten Bit und zählt für alle Wörter wie oft dieses Bit 0 bzw. 1 ist. Da eine Zahl fehlt, muss entweder die 0 oder die 1 häufiger als k-tes Bit vorkommen. Die fehlende Zahl ist unter den Zahlen enthalten, dessen k-tes Bit weniger häufig war. Damit muss die fehlende Zahl mit diesem Bit beginnen. Nun macht man das ganze rekursiv auf den Zahlen, dessen k-tes Bit weniger häufig war ($\leq n/2$ Elemente) und zwar mit dem $k-1$-ten Bit und so weiter. So bekommt man Stück für Stück die Bits der fehlenden Zahl, während die Anzahl der zu untersuchenden Elemente sich immer halbiert.

```
1: procedure search(A : Array [0..n − 2] of ℕ≥0)
2:    left := 0; right := n − 2, bit = k
3:    call findMissingNumber(A, left, right, bit)
4: return
```

```
1: procedure findMissingNumber(A : Array [0..n − 2] of ℕ≥0, left, right, bit : ℕ≥0)
2:    zeros := 0, ones := 0;
3:    for i :=left to right do
4:       if Bit bit of A[i] is 0 then zeros++ else ones++
5:    if zeros > ones then
6:       print 1, left := ⌊(left+right)/2⌋ + 1
7:    else
8:       print 0, right := ⌊(left+right)/2⌋ − 1
9:    if left ! = right then call findMissingNumber(A, left, right, bit−−)
10: return
```

2. Aus der Beschreibung des Algorithmus ergibt sich sofort die folgende Abschätzung für die Laufzeit $T(n) \leq T(\frac{n}{2}) + n$.

3. Das Mastertheorem liefert uns mit $d = 1, b = 2, c = 1$ eine Gesamtlaufzeit von $\Theta(n)$. Alternativ kann man die Rekursion auch folgendermaßen lösen: $T(n) \leq T(\frac{n}{2}) + n = T(\frac{n}{4}) + \frac{n}{2} + n = T(\frac{n}{8}) + \frac{n}{4} + \frac{n}{2} + n = \cdots \leq n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n$, wobei die letzte Reihe gerade die geometrische Reihe ist.

Aufgabe 4 (*Karatsuba-Ofman Multiplikation,* 4 *Punkte*)

Multiplizieren Sie 1231 mit 1622 (das sind Dezimalzahlen!) nach dem Algorithmus von Karatsuba und Ofman.
Hinweis: Führen Sie genau die einstelligen Operationen direkt aus.

Lösung:
Der Algorithmus:

```
1: Function recMult(a, b)
2:    assert a und b haben n = 2k Ziffern
3:    if n = 1 then return a · b
4:    Schreibe a als a1 · B^k + a0
5:    Schreibe b als b1 · B^k + b0
6:    c11:= recMult(a1, b1)
7:    c00:= recMult(a0, b0)
8:    c0110:= recMult((a1 + a0), (b1 + b0))
9:    e:= c11 · B^2k + (c0110 − c11 − c00)B^k + c00
10: return e
```

Schema zum Aufschrieb eines Multiplikationsaufrufs:

$a \cdot b$	$a_1 \cdot b_1 = \text{recMult}(a_1, b_1)$
	$a_0 \cdot b_0 = \text{recMult}(a_0, b_0)$
e	$(a_1 + a_0) \cdot (b_1 + b_0) = \text{recMult}((a_1 + a_0), (b_1 + b_0))$

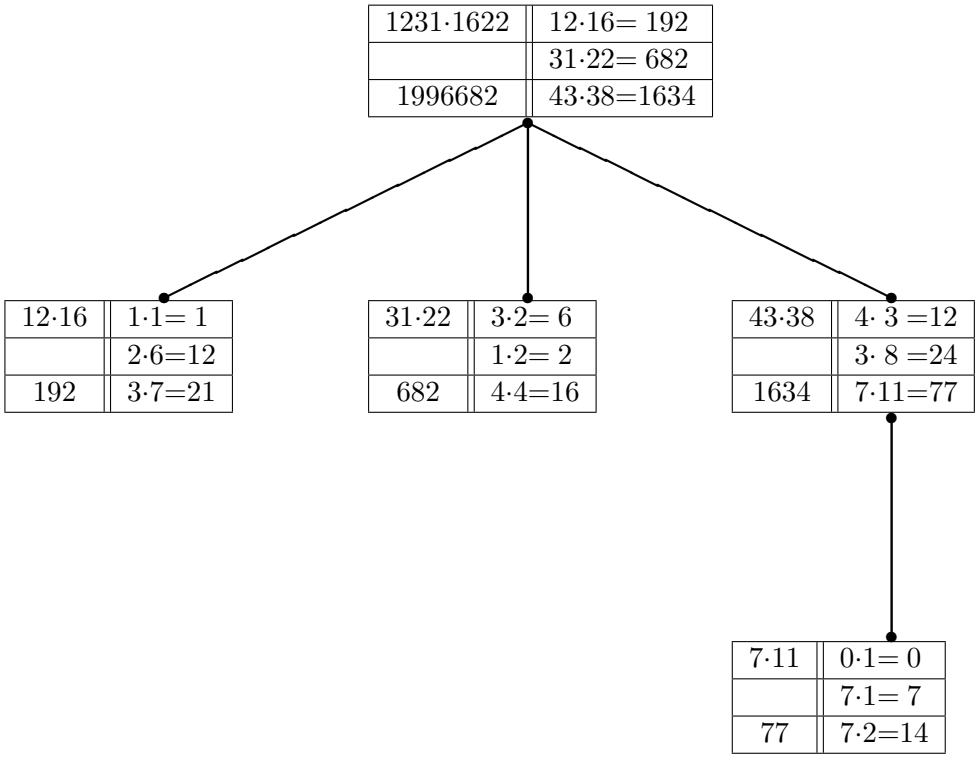

Figure 1: Berechnungsbaum

Berechnung:
Aufbau als Rekursionsbaum.

Aufgabe P1 (*Praktische Aufgabe, optional*)

Für die praktischen Übungen verwenden wir die Plattform `www.hackerrank.com`. Hier müssen Sie sich registrieren um an den Übungen teilzunehmen. Unter dem Link

`https://www.hackerrank.com/adsi-2023`

finden die praktischen Übungen in der Form eines Programmierwettbewerbs statt.

Die erste Challenge dient vor allem dazu, die Plattform HackerRank kennen zu lernen.
Die Aufgabe besteht darin ein Programm zu schreiben, das als Eingabe einen Namen bekommt. Die Ausgabe des Programms soll sein: **Hello out there, 'Name'!**. Wobei die Person begrüßt wird, dessen Name das Programm als Eingabe erhält.

Der fertige Code kann direkt über die Plattform eingereicht werden.