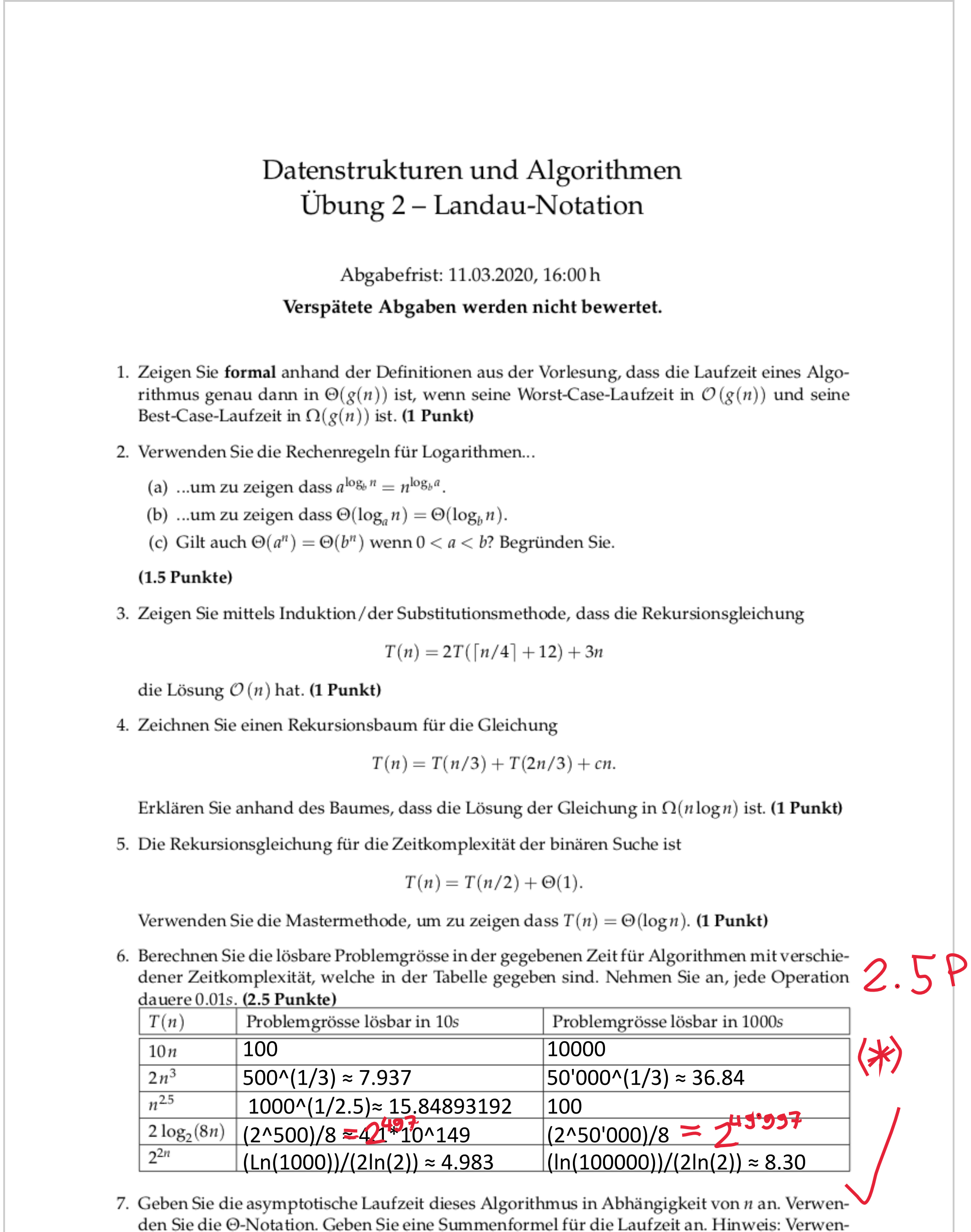
DA-2

Mittwoch, 3. März 2021

Lorenz Pfäffli 18-925-230
Lukas Ingold 20-123-998

TK

8.25/10



Datenstrukturen und Algorithmen
Übung 2 – Landau-Notation

Abgabefrist: 11.03.2020, 16:00 h

Verspätete Abgaben werden nicht bewertet.

1. Zeigen Sie **formal** anhand der Definitionen aus der Vorlesung, dass die Laufzeit eines Algorithmus genau dann in $\Theta(g(n))$ ist, wenn seine Worst-Case-Laufzeit in $\mathcal{O}(g(n))$ und seine Best-Case-Laufzeit in $\Omega(g(n))$ ist. **(1 Punkt)**
2. Verwenden Sie die Rechenregeln für Logarithmen...
   - (a) ...um zu zeigen dass $a^{\log_b n} = n^{\log_b a}$.
   - (b) ...um zu zeigen dass $\Theta(\log_a n) = \Theta(\log_b n)$.
   - (c) Gilt auch $\Theta(a^n) = \Theta(b^n)$ wenn $0 < a < b$? Begründen Sie.

   (1.5 Punkte)
3. Zeigen Sie mittels Induktion/der Substitutionsmethode, dass die Rekursionsgleichung
   $$T(n) = 2T(\lceil n/4 \rceil + 12) + 3n$$
   die Lösung $\mathcal{O}(n)$ hat. **(1 Punkt)**
4. Zeichnen Sie einen Rekursionsbaum für die Gleichung
   $$T(n) = T(n/3) + T(2n/3) + cn.$$
   Erklären Sie anhand des Baumes, dass die Lösung der Gleichung in $\Omega(n \log n)$ ist. **(1 Punkt)**
5. Die Rekursionsgleichung für die Zeitkomplexität der binären Suche ist
   $$T(n) = T(n/2) + \Theta(1).$$
   Verwenden Sie die Mastermethode, um zu zeigen dass $T(n) = \Theta(\log n)$. **(1 Punkt)**
6. Berechnen Sie die lösbare Problemgrösse in der gegebenen Zeit für Algorithmen mit verschiedener Zeitkomplexität, welche in der Tabelle gegeben sind. Nehmen Sie an, jede Operation dauere 0.01s. **(2.5 Punkte)**

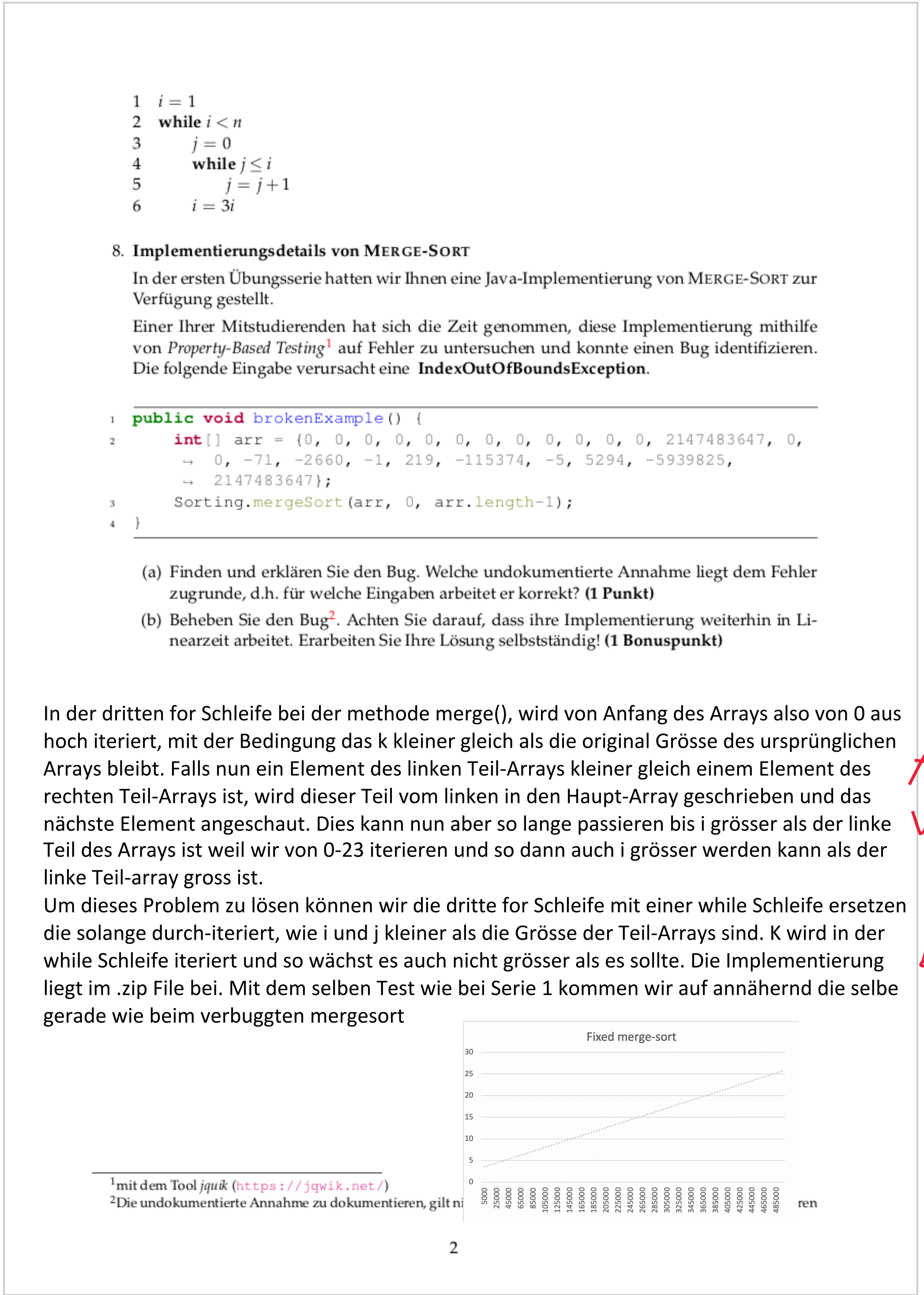
2.5P

$T(n)$	Problemgrösse lösbar in 10s	Problemgrösse lösbar in 1000s
$10n$	100	10000
$2n^3$	500^(1/3) ≈ 7.937	50'000^(1/3) ≈ 36.84
$n^{2.5}$	1000^(1/2.5)≈ 15.84893192	100
$2\log_2(8n)$	(2^500)/8 ≈4.1*10^149 $= 2^{497}$	(2^50'000)/8 $= 2^{49'997}$
2^{2n}	(Ln(1000))/(2ln(2)) ≈ 4.983	(ln(100000))/(2ln(2)) ≈ 8.30

(*) ✓

7. Geben Sie die asymptotische Laufzeit dieses Algorithmus in Abhängigkeit von n an. Verwenden Sie die Θ-Notation. Geben Sie eine Summenformel für die Laufzeit an. Hinweis: Verwenden Sie die Partialsummenformel für geometrische Reihen ($\sum_{k=0}^{n} a_0 q^k = a_0 \frac{q^{n+1}-1}{q-1}$). **(1 Punkt)**

(*) Problemgrössen sind ganzzahlig zB. 4.983 => $n \leq 4$



```
1  i = 1
2  while i < n
3      j = 0
4      while j ≤ i
5          j = j + 1
6      i = 3i
```

8. **Implementierungsdetails von MERGE-SORT**

   In der ersten Übungsserie hatten wir Ihnen eine Java-Implementierung von MERGE-SORT zur Verfügung gestellt.

   Einer Ihrer Mitstudierenden hat sich die Zeit genommen, diese Implementierung mithilfe von *Property-Based Testing*[1] auf Fehler zu untersuchen und konnte einen Bug identifizieren. Die folgende Eingabe verursacht eine **IndexOutOfBoundsException**.

```java
public void brokenExample() {
    int[] arr = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2147483647, 0,
      ↪ 0, -71, -2660, -1, 219, -115374, -5, 5294, -5939825,
      ↪ 2147483647};
    Sorting.mergeSort(arr, 0, arr.length-1);
}
```

   - (a) Finden und erklären Sie den Bug. Welche undokumentierte Annahme liegt dem Fehler zugrunde, d.h. für welche Eingaben arbeitet er korrekt? **(1 Punkt)**
   - (b) Beheben Sie den Bug[2]. Achten Sie darauf, dass ihre Implementierung weiterhin in Linearzeit arbeitet. Erarbeiten Sie Ihre Lösung selbstständig! **(1 Bonuspunkt)**

In der dritten for Schleife bei der methode merge(), wird von Anfang des Arrays also von 0 aus hoch iteriert, mit der Bedingung das k kleiner gleich als die original Grösse des ursprünglichen Arrays bleibt. Falls nun ein Element des linken Teil-Arrays kleiner gleich einem Element des rechten Teil-Arrays ist, wird dieser Teil vom linken in den Haupt-Array geschrieben und das nächste Element angeschaut. Dies kann nun aber so lange passieren bis i grösser als der linke Teil des Arrays ist weil wir von 0-23 iterieren und so dann auch i grösser werden kann als der linke Teil-array gross ist.

Um dieses Problem zu lösen können wir die dritte for Schleife mit einer while Schleife ersetzen die solange durch-iteriert, wie i und j kleiner als die Grösse der Teil-Arrays sind. K wird in der while Schleife iteriert und so wächst es auch nicht grösser als es sollte. Die Implementierung liegt im .zip File bei. Mit dem selben Test wie bei Serie 1 kommen wir auf annähernd die selbe gerade wie beim verbuggten mergesort

0.5P

Aber warum wird i zu gross?

Die Arrays werden nicht mehr korrekt sortiert

---

[1] mit dem Tool *jquik* (https://jqwik.net/)

[2] Die undokumentierte Annahme zu dokumentieren, gilt ni ren

---

1.) Worst-Case : $O(g(n))$

Best-Case : $\Omega(g(n))$

WC: $O(g(n)) = \{ f(n) : \exists$ positive konst. c und n_0, so dass $0 \le f(n) \le c \cdot g(n)$ für alle $n \ge n_0 \}$

BC: $\Omega(g(n)) = \{ f(n) : \exists$ positive konst. c und n_0, so dass $0 \le cg(n) \le f(n$ für alle $n \ge n_0 \}$

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ Sollte man zeigen

$\Theta(g(n)) = \{ f(n) : \exists$ positive Konstante c_1 und c_2 sowie n_0, so dass $c_1 g(n) \le f(n) \le c_2 g(n)$ für alle $n \ge n_0 \}$

0.5P

2a) $a^{\log_b n} = n^{\log_b a}$

$b^x = n$ & $b^y = a$

$a^{\log_b n} = n^{\log_b a} \quad | \cdot \log_a$

$\log_a a^{\log_b n} = \log_a n^{\log_b a}$ ✓

$\log_b n = \log_b a \cdot \log_a n$

$\frac{\log n}{\log b} = \frac{\log a}{\log b} = \frac{\log n}{\log a}$

$\frac{\log n}{\log b} = \frac{\log n}{\log b}$ (w)

2b) $\Theta(\log_a n) = \Theta(\log_b n)$

sei $f(n)$ eine funktion so dass

$f(n) \in \Theta(\log_a n) \Rightarrow c_1, c_2, n_0 > 0$ und $n \ge n_0$

$0 \le c_1 \log_a n \le f(n) \le c_2 \log_a n$

für $a,b \Rightarrow \log_a n = \frac{\log_b n}{\log_b a} \Rightarrow \forall n \ge n_0$

$0 \le c_1 \frac{\log_b n}{\log_b a} \le f(n) \le c_2 \frac{\log_b n}{\log_b a}$

$c_3 = \frac{c_1}{\log_b a}$, $c_4 = \frac{c_2}{\log_b a} \Rightarrow \forall n \ge n_0$

$0 \le c_3 \log_b n \le f(n) \le c_4 \log_b n \Rightarrow f(n) \in \Theta(\log_b n)$ ✓ □

Einfacher $c = \log_b(a) > 0$

$\Rightarrow \log_b(n) = c \log_a(n)$

1P

2c.) $\Theta(a^n) = \Theta(b^n)$ wenn $0 < a < b$

sei $f(n) \in \Theta(a^n)$ mit $c_1, c_2, n_0 > 0$ und $n \ge n_0$

$0 \le c_1 \cdot a^n \le f(n) \le c_2 \cdot b^n$

wenn $a < b$ und $a, b \neq 0$

dann

. . .

3.) $T(n) = 2T(\lceil n/4 \rceil + 12) + 3n$

Aufrunden Klammern

Erraten: $O(n)$

Induktions annahme : $T(k) \le c \cdot k$ für $k < n$

$T(\lceil n/4 \rceil + 12) \le c \cdot (\lceil n/4 \rceil + 12)$ ✓

zu beweisen: $T(n) \le c \cdot n$

aus Annahme : $T(\lceil n/4 \rceil + 12) \le c \cdot (\lceil n/4 \rceil + 12)$

$T(n) = 2T(\lceil n/4 \rceil + 12) + 3n \le c \cdot n$

$\le 2 \cdot c \cdot (\lceil n/4 \rceil + 12) + 3n$

$\frac{c}{2} \cdot n + 24c + 3n$ ~

$= c \cdot n - (\frac{c}{2} \cdot n - 24c - 3n) < c \cdot n$

$c \ge 1$ ~ ✓

$n \ge 24c$

$\Rightarrow T(n) \in O(n)$

Randbedingung : $T(n) \begin{cases} c & \text{wenn } n < n_0 \\ 2T(\lceil n/4 \rceil + 12) + 3n & \text{wenn } n \ge n_0 \end{cases}$ ✓

$\Rightarrow T(16) = C \le c \cdot 16$, da nun c beliebig gewählt werden kann $\Rightarrow$ liegt in $O(n)$

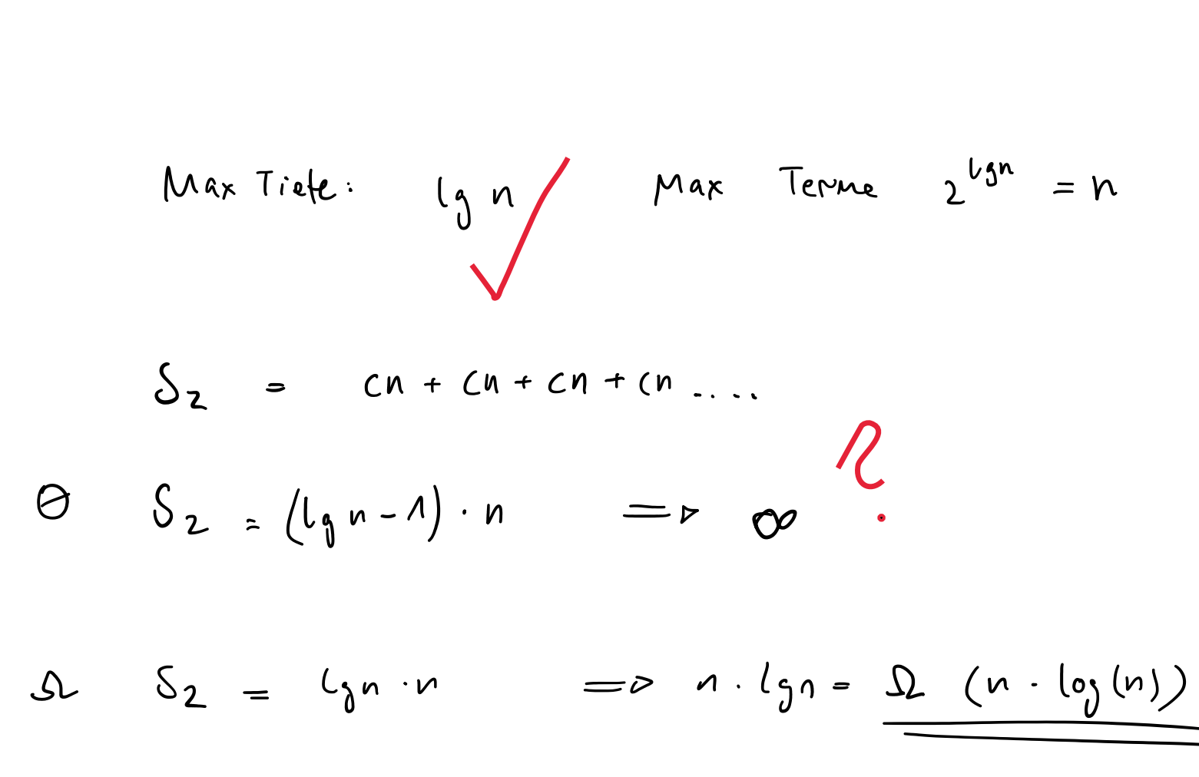
0.75 P

$\lceil \frac{n}{4} \rceil \le \frac{n}{4} + 1$

aber

$\lceil \frac{n}{4} \rceil \not\le \frac{n}{4}$ zB $n=3$

4.) $T(n) = T(n/3) + T(2n/3) + cn$ $\Omega(n \log n)$



$c \cdot n$

S_2

1

$\frac{cn}{3} + \frac{2cn}{3} = \frac{3cn}{3} = cn$

1

$\frac{cn}{9} + \frac{2cn}{9} + \frac{2cn}{9} + \frac{4cn}{9} = cn$ ✓

1

$cn + 2cn + 2cn + 4cn + 2cn + 4cn + 4cn + 8cn = cn$

Max Tiefe: $\lg n$ ✓ Max Terme $2^{\lg n} = n$

1P

$S_2 = cn + cn + cn + cn \ldots$

Θ $S_2 = (\lg n - 1) \cdot n \Rightarrow \infty$?

Ω $S_2 = \lg n \cdot n \Rightarrow n \cdot \lg n = \underline{\underline{\Omega(n \cdot \log(n))}}$ ✓

A5) $T(n) = T(n/2) + \Theta(1)$ wähle $f(n) := 1$

$= T(n/2) + 1$

1P

$\Rightarrow a = 1, b = 2, f(n) = 1$ ✓

Wir überprüfen Fall 2

$f(n) \overset{?}{=} \Theta(n^{\log_2(1)}) = \Theta(n^0) = \Theta(1)$ ✓

$f(n) = 1 = \Theta(1)$

$\Rightarrow$ Fall 2 des Mastertheorems kann angewendet werden. ✓

$\Rightarrow T(n) = \Theta(n^{\log_2 a} \cdot \lg n)$

$= \Theta(\lg n)$

$= \underline{\underline{\Theta(\log(n))}}$ ✓

A7)

1P

```
0  Algo(n) {
1    i = 1
2      while i < n :
3          j = 0
4          while j ≤ i :
5              j = j + 1
6          i = 3i
7  }
```

Die Operationen der Zeilen 1, 3, 5, 6 haben eine konstante Laufzeit von $\Theta(1)$.
Nun betrachten wir die äussere Scheife (z. 2 - 6).
Diese wird m Mal durchlaufen, wobei $m \le \log_3(n)$. Dies ist so, da i am Ende jeder Schleifeniteration verdreifacht wird. (Wenn i immer verdoppelt werden würde, so wäre $m \le \log_2(n)$) ✓

Nun betrachten wir die innere Schleife (z. 4 - 5).
Die Anzahl p ihrer Iterationen schätzen wir nach oben ab : $p \le i < n$

Die Worst-Case Laufzeit beträgt somit $O(\log_3(n) \cdot n) = O(n \log(n))$

Bei genaueren Betrachtung des Algorithmus können Schärfere Schranken bestimmt werden.

$\sum_{k=0}^{m} a_0 q^k = a_0 \frac{q^{m+1} - 1}{q - 1} = 1 \cdot \frac{3^{m+1} - 1}{3 - 1} = 1 \cdot \frac{3^{\log_3(n)+1} - 1}{3 - 1} = \frac{3^{\log_3(n)+1} - 1}{2}$ ✓

$\Rightarrow$ Die Laufzeit des Algorithmus beträgt also $\underline{\underline{\Theta\left(\frac{3^{\log_3(n)+1} - 1}{2}\right)}} = \Theta(n)$ ~✓

$3^{\log_3(n)} = n$

1	3	9	27	81	243
0 1	0 1 2 3	0 1 2 3 … 9	0 … 27	0 … 81	0 … 243
2	4	10	28	82	243

2 → 2.5 → 2.8 → 2.92 → 2.96