Sebastian Kodek, 1507066

Beispiel 1: Stable Matching

Schülerinnen

	1.	**2.**	**3.**	**4.**
W	B	A	D	C
X	B	C	D	A
Y	A	B	D	C
Z	D	C	B	A

Gastfamilien

	1.	**2.**	**3.**	**4.**
A	Z	Y	X	W
B	X	Y	W	Z
C	Z	X	Y	W
D	Z	W	Y	X

1. Zuordnung W-B
2. Zuordnung X-B. B bevorzugt X gegenüber W, letzterer ist also wieder frei.
3. X-B, W-A
4. X-B, Y-A. A bevorzugt Y gegenüber W, letzterer ist wieder frei.
5. X-B, Y-A, W-D
6. X-B, Y-A, Z-D. D bevorzugt Z gegenüber W, letzterer ist wieder frei.
7. **X-B, Y-A, Z-D, W-C** ist die finale Zuordnung.

Beispiel 2: Schranken von Laufzeitfunktionen

$f(n)$ **ist in**	$\Theta(.)$	$O(.)$	$\Omega(.)$	**keines**
$g_1(n)$	X	X	X	
$g_2(n)$		X		
$g_3(n)$				X

$$f(n) = 2^n(100n^2 + 20n + 7)$$

$$g_1(n) = \begin{cases} 100n^2 + 2^n \cdot n^2 + 7n + (\frac{3}{2})^n & \text{falls } (n < 10^3) \text{ oder } (n \geq 10^5) \\ 5^n + 2^{\frac{3n}{2}} + 100n & \text{sonst} \end{cases}$$

$$g_2(n) = \frac{2^{2^n}}{10000}$$

$$g_3(n) = \begin{cases} 60 & \text{falls } n \text{ eine Primzahl ist} \\ n^{n^n} & \text{sonst} \end{cases}$$

Beispiel 3: Laufzeit

```
i ← n
j ← n
k ← 0
while i > 0                          } Θ (n)
    j ← j · n
    i ← i − 1
while j > 0 ⟶ j = n^n                } Θ(log_2 n^n) = Θ(n log_2 n)
    if j = ⌊j/2⌋ · 2 then
        k ← k + 1
    j ← ⌊j/2⌋
```

Laufzeit: $\Theta(n \log n)$

Beispiel 4: Laufzeit

```
max ← 0
ind ← 0
for i ← 1, ..., n                    } Θ (n)
    A[i] ← n
if n = 1 then
    while A[1] > 0                   } Θ (1)
        A[1] ← A[1] − 1
else
    while A[n] > 0                                          } Θ (n^3)
        max ← 0
        for j ← 1, ..., n                                  } Θ (n)
            if A[j] > max then
                max ← A[j]
                ind ← j
                if A[1] > A[2] then   ← nie erfüllt!
                    for k ← 1, ..., j
                        max ← max + 1
        A[ind] ← A[ind] − 1
```

Laufzeit: $\Theta\ (n^3)$

Sebastian Kodek, 1507066

Aufgabe 5: Beweis

Gegeben sind $f_1(n) = \mathrm{O}(g_1(n))$ und $f_2(n) = \mathrm{O}(g_2(n))$; es soll bewiesen werden, dass gilt: $f_1(n) \cdot f_2(n) = \mathrm{O}(g_1(n) \cdot g_2(n))$.

- Für die Konstanten c_1 und n_{01} gilt für alle $n \geq n_{01}$: $f_1(n) \leq c_1 \cdot g_1(n)$
- Analog gilt für die Konstanten c_2 und n_{02} für alle $n \geq n_{02}$: $f_2(n) \leq c_2 \cdot g_2(n)$
- Definieren: $n_0^* = \max(n_{01}, n_{02})$ und $c^* = c_1 \cdot c_2$.
- Daraus ergibt sich:

  $f_1(n) \cdot f_2(n) \leq (c_1 \cdot g_1(n)) \cdot (c_2 \cdot g_2(n))$

  $f_1(n) \cdot f_2(n) \leq c^* \cdot g_1(n) \cdot g_2(n)$

  $\boldsymbol{f_1(n) \cdot f_2(n) = \mathrm{O}(g_1(n) \cdot g_2(n))}$ **für alle** $\boldsymbol{n \geq n_0^*}$

- Beispiel**:**

  $f_1(n) = 2n^2 + 1 = \mathrm{O}(n^2)$

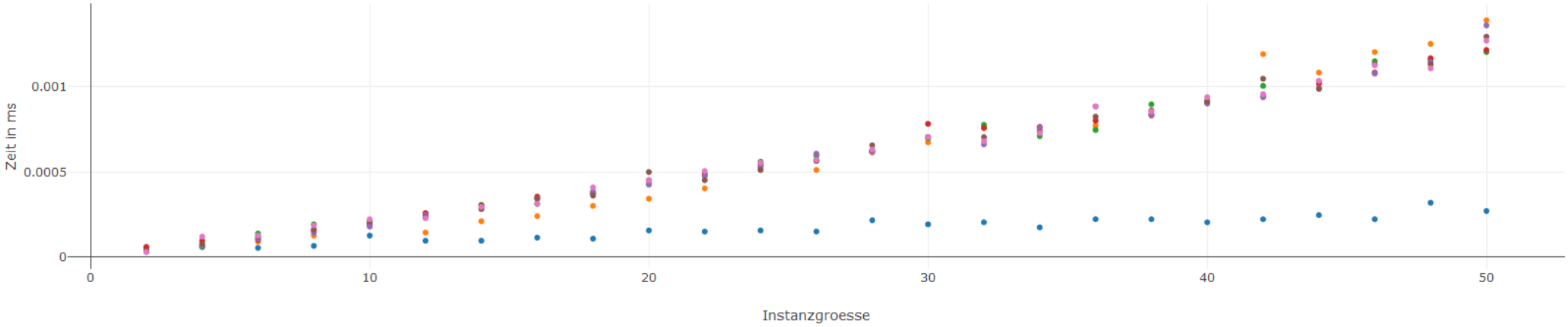
  $f_2(n) = n^3 + 3n^2 = \mathrm{O}(\mathrm{n}^3)$

  $f_1(n) \cdot f_2(n) = 2n^5 + 6n^4 + n^3 + 3n^2 = \mathrm{O}(n^5)$

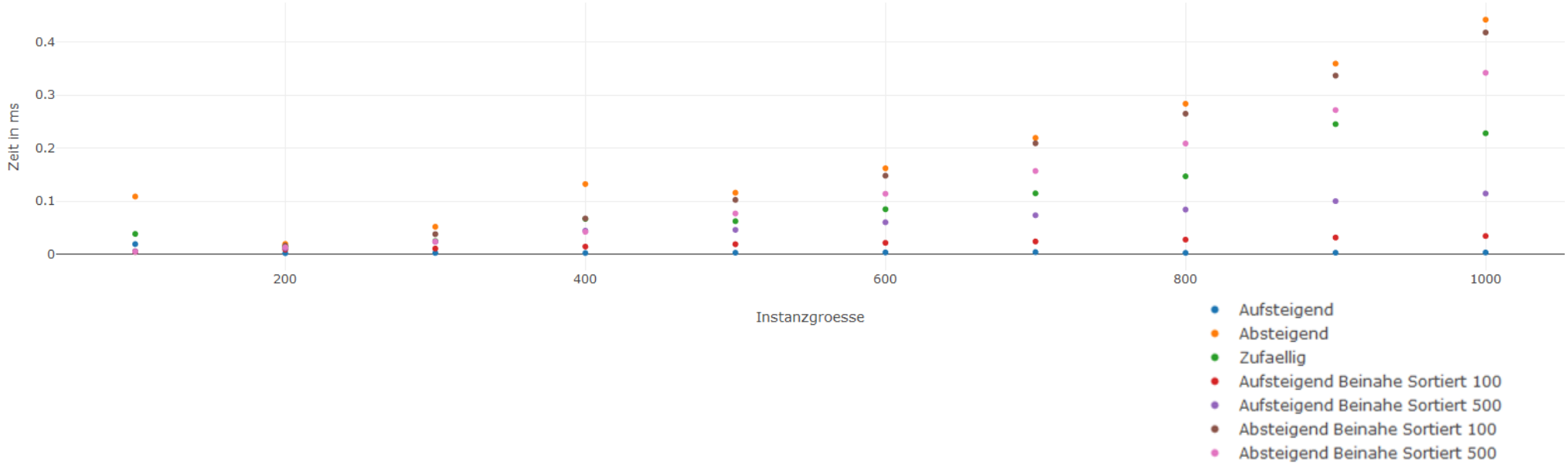
Sebastian Kodek, 1507066

Beispiel 6: Programmieraufgabe

Test Korrektheit

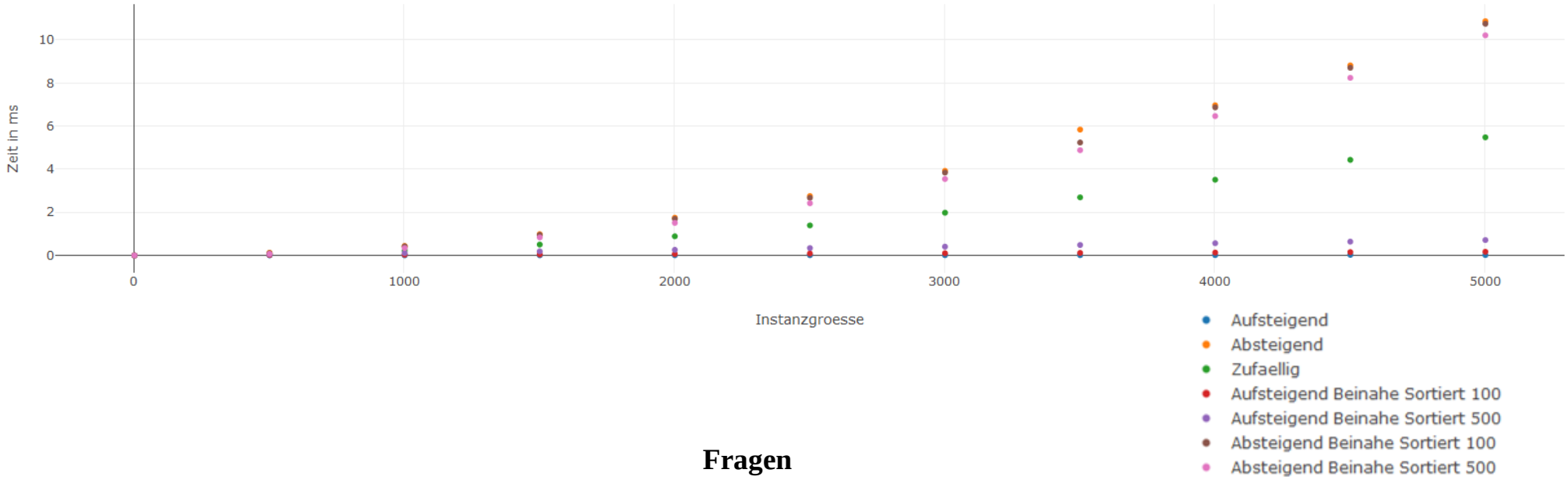


Test Mittel



Sebastian Kodek, 1507066

Test Abgabe



Fragen

1) **Welchen Einfluss haben die unterschiedlich sortierten Arrays?** Bei einem schon aufsteigend sortierten Array – dem Best Case – und auch bei einem aufsteigend beinahe sortierten Array läuft der Algorithmus am schnellsten durch, bei einem absteigend sortierten Array – dem Worst Case – braucht der Insertionsort-Algorithmus am längsten, ähnlich lange benötigt er bei einem beinahe absteigend sortierten Array. Die Laufzeit für Arrays mit zufälligen Werten liegt dazwischen; je größer die Arrays sind, umso mehr nähert sie sich dem Durchschnitt zwischen Worst- und Best-Case-Laufzeit an (empirisches Gesetz der großen Zahlen).
2) **Warum ergeben sich die Laufzeitunterschiede?** Der Algorithmus sortiert das Array aufsteigend. Dabei wird innerhalb des Arrays eine sortierte Teilliste gebildet, die mit jeder Iteration der äußeren Schleife um ein Element aus dem unsortierten Array vergrößert wird, das mithilfe der inneren Schleife an der richtigen Stelle der Teilliste eingefügt wird. Wenn das letzte Element in die sortierte Liste eingefügt wurde, entspricht diese dem sortierten Array. Bei einem schon aufsteigend sortierten Array läuft die innere Schleife, die die Werte der Teilliste verschiebt, sodass der ausgewählte Wert an der richtigen Stelle eingefügt werden kann, kein einziges Mal durch, da der Vergleich $numbers[y-1] >$

$einzufuegenderWert$ nie zutrifft; es muss ja nichts umsortiert und daher auch nichts im Array verschoben werden. Bei einem beinahe sortierten Array wird der Vergleich oft nicht zutreffen und die innere Schleife läuft daher nur wenige Male durch. Wenn das Array dagegen absteigend sortiert ist, muss es komplett „umsortiert“ werden, wobei die innere Schleife jedes Mal bis ans Ende der aktuellen Teilliste durchläuft, um das entsprechende Element von seiner Ursprungsposition bis an den Anfang des Arrays zu verschieben. Genau dieses Verschieben der Elemente der Teilliste ist es auch, was den Algorithmus im Vergleich zu anderen Sortieralgorithmen so ineffizient macht.

3) **Laufzeitabschätzung**:
   a) Best Case, aufsteigend sortiertes Array: $O(n)$ – es läuft die äußere Schleife bei einem Array mit n Elementen genau n -mal durch. Der Insertionsort-Algorithmus läuft also bei einem schon richtig sortierten Eingabearray schneller durch als andere Sortieralgorithmen wie Mergesort oder Quicksort.
   b) Worst Case, absteigend sortiertes Array: $\frac{n(n-1)}{2} \in O(n^2)$ – es müsen ja alle Elemente jeweils an den Anfang des Arrays verschoben werden, die innere Schleife läuft $i-1$ mal durch, wobei i die Zählvariable der äußeren Schleife und damit die Position des einzufügenden Elements im Array ist.
   c) Average Case: $\frac{n(n-1)}{4} \in O(n^2)$ – entspricht der durchschnittlichen Anzahl an nötigen Verschiebeoperationen auf einem zufällig gewählten Array.
4) **In welchen Situationen ist Insertionsort in der Praxis gut geeignet?** Wenn davon auszugehen ist, dass die zu sortierenden Arrays erstens nicht zu groß werden und zweitens schon eine gewisse Vorsortierung aufweisen.

Quellcode

```
static void sort(Integer[] numbers) {
    for (int i = 0; i < numbers.length; i++) {
        int x = numbers[i];
        int y = i;
        while (y > 0 && numbers[y-1] > x) {
            numbers[y] = numbers[--y];
        }
        numbers[y] = x;
    }
}
```