Sarah Ertel
Patrick Greher
Eugen Ljavin

1	2	3	Σ
10	15	8	33

Übungsblatt Nr. 4

(Abgabetermin 17.05.2018)

Aufgabe 1

a)

Es sei das Array $A =$ | 7 | 1 | 4 | 1 | 2 | 5 | 4 | 7 | 1 | 5 | 2 | mit der Länge $n = 11$ gegeben.
Annahme: $max\{A\} = 7$ und $min\{A\} = 0 \Rightarrow k = 8$

1. Schritt: Initialisiere das Array C der Länge $k = 8$ mit den Werten 0 (Z. 1).

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

2. Schritt: Iteriere von vorne beginnend über das Array A und summiere die Anzahl der Werte von A an dem zugehörigen Index des Arrays C auf (Z. 2 - 4). Der Wert an jedem Index im Array C gibt nun an, wie häufig ein Wert in A vorkommt. Der Index von C entspricht dabei dem Wert von A.

0	1	2	3	4	5	6	7
0	3	2	0	2	2	0	2

3. Schritt: Summiere alle aufeinanderfolgenden Werte im Array C auf (Z. 5 - 7). Der Wert am letzten Index von C muss n entsprechen.

0	1	2	3	4	5	6	7
0	$3+0=3$	$3+2=5$	$5+0=5$	7	9	$9+0=9$	$9+2=11$

4. Schritt: Initialisiere das Array R mit der Länge $n = 11$ (Z. 8)

1	2	3	4	5	6	7	8	9	10	11

5. Schritt: Sortieren. Iteriere über das Array A von hinten beginnend. Schreibe an den Index des Arrays R (der Index des Arrays R entspricht dem Wert des Arrays C und der Index des Arrays C entspricht dem aktuell betrachteten Wert im Array A) den aktuell betrachteten Wer von A entspricht. Dekrementiere Anschließend den Wert in dem Array C. (Z. 9 - 12). Nach n Wiederholungen ist das Array A im Array R sortiert.

1	2	3	4	5	6	7	8	9	10	11
				2						

0	1	2	3	4	5	6	7
0	3	4	5	7	9	9	11

1	2	3	4	5	6	7	8	9	10	11
				2				5		

0	1	2	3	4	5	6	7
0	3	4	5	7	8	9	11

1	2	3	4	5	6	7	8	9	10	11
		1		2				5		

0	1	2	3	4	5	6	7
0	2	4	5	7	8	9	11

⋮

1	2	3	4	5	6	7	8	9	10	11
1	1	1	2	2	4	4	5	5	7	7

0	1	2	3	4	5	6	7
0	0	3	5	5	7	9	9

b)

Nachdem das Array C aufsummiert wurde (siehe vorherige Aufgabe), lässt sich das Array C folgendermaßen interpretieren: "Wie viele Elemente existieren, die kleiner-gleich dem betrachteten Element sind"(das betrachtete Element entspricht dabei dem Index von C). Beispielsweise gibt es 11 Elemente, die kleiner-gleich dem Element 7 sind. Da C von Links nach rechts aufsummiert wird, handelt es sich bei kleiner-gleich Relation. Dies impliziert, dass ein betrachtetes Element immer an einem Index platziert wird, der größer/gleich der Anzahl kleinerer Elemente in Abängigkeit des betrachteten Elements entspricht. Der Algorithmus sortiert folglich korrekt.

c)

Der Algorithmus arbeitet mit drei Schleifen:

1. Die erste Schleife (Zeile 2 - 4) iteriert über $n \Rightarrow \mathcal{O}(n)$.
2. Die zweite Schleife (Zeile 5 - 7) iteriert über $k \Rightarrow \mathcal{O}(k)$.
3. Die dritte Schleie (Zeile 9 - 12) iteriert über $n \Rightarrow \mathcal{O}(n)$.

Es ergibt sich eine Komplexität von $\mathcal{O}(n) + \mathcal{O}(k) + \mathcal{O}(n) = \mathcal{O}(2 \cdot n + k) = \mathcal{O}(n + k)$

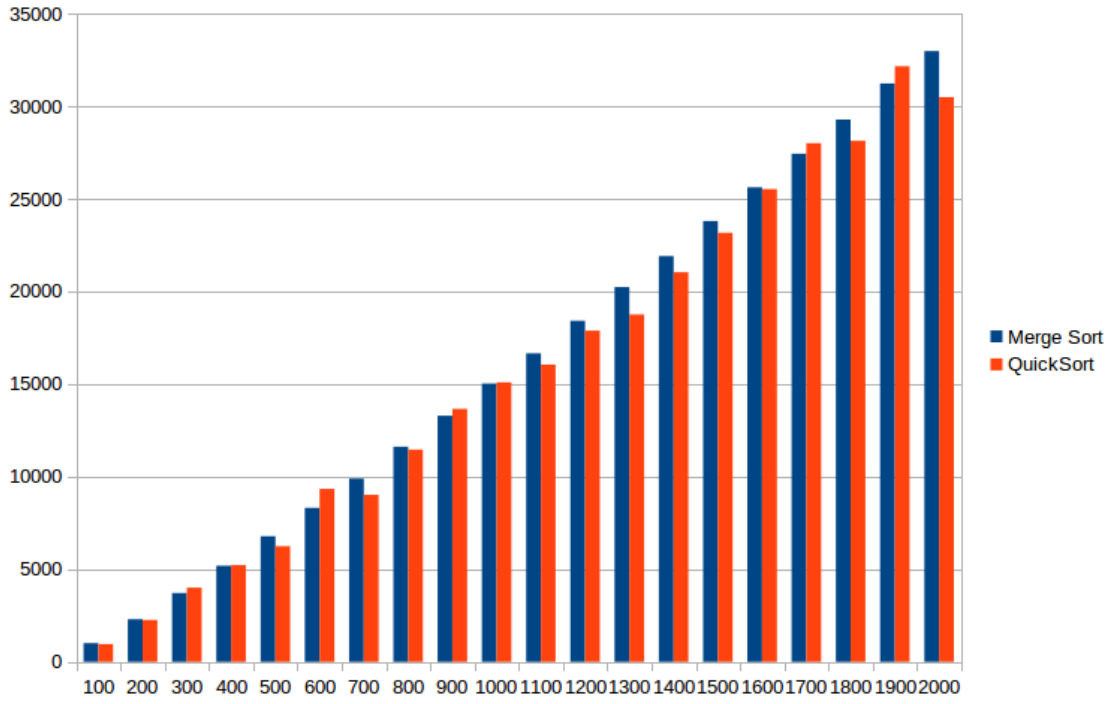
d)

Entscheident sind die Zeilen 9 - 2 des Pseudocodes. Nach dem dritten Schritt (siehe Aufgabe 1 a)) kann dem Array C entnommen werden, welche zu sortierende Zahl an welchen Index von R gesetzt werden muss. Da nach jeder Sortieroperation der Wert in C dekrementiert wird, wird auch der Index von R verkleinert, an den eine äquivalente zu sortierende Zahl in einem späteren Schritt in R gesetzt wird. Da von hinten beginnend sortiert wird, vertauscht sich die Reihenfolge nicht. Der Algorithmus arbeitet somit stabil.

e)

Diese Modifikation hat weder Auswirkungen auf das Array A, das Array C noch auf das Array R. Lediglich die Reigenfolge, in der sortiert wird, wird vertauscht ⇒ Alle Elemente werden weiterhin korrekt sortiert, jedoch werden die vorderen Elemente von A von hinten in R platziert, da das Array A von vorne beginnend sortiert wird. Folglich arbeitet der Algorithmus weiterhin korrekt, ist aber nicht mehr stabil.

Aufgabe 2



*Countingsort nicht aufgeführt, da nicht vergleichsbasiert

Aufgabe 3

a)

Algorithm 1: Algorithmus, der ein aufeinanderfolgendes Zahlenpaar aus einem Array mit der kleinsten Differenz sucht

```
1  function findMinDifferenceFigurePair(array[ ])
2  mergeSort(array);
3  minDifference ← array[1] − array[0];
4  figurePair[ ];
5  for i ← 1; i < array.length − 1; i ← i + 1 do
6  |   difference ← array[i] − array[i − 1];
7  |   if difference ≤ minDifference then
8  |   |   minDifference ← difference;
9  |   |   figurePair[0] ← array[i − 1];
10 |   |   figurePair[1] ← array[i];
11 |   end
12 end
13 return figurePair;
```

Zwei Zahlen aus einem Array sind per vorgegebener Definition $aufeinanderfolgend$, wenn das Array sortiert ist. Deswegen wird zunächst das zu durchsuchende Array der Länge n mit einem Sortieralgorithmus der Komplexität $\mathcal{O}(n \log n)$ aufsteigend sortiert

(hier: Mergesort). Anschließend wird die Differenz zweier aufeinanderfolgender Elemente berechnet und geprüft, ob diese die neue minimale Differenz darstellt. Da das gesamte Array durchlaufen werden muss, ist die Komplexität der Operation $\mathcal{O}(n)$.
Es ergibt sich somit eine Gesamtlaufzeit von $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n + n) = \mathcal{O}(n \log n)$.

b)

Algorithm 2: Algorithmus, der das Element in einem Array sucht, das am häufigsten vorkommt

```
1  function findMostFrequentElement(array[ ])
2  mergeSort(array);
3  n ← 0;
4  maxCount ← 0;
5  for i ← 0; i < array.length − 1; i ← i + n do
6  |  n ← i;
7  |  currentCount ← 0;
8  |  repeat
9  |  |  currentCount ← currentCount + 1;
10 |  |  n ← n + 1;
11 |  until array[i] = array[n];
12 |  if currentCount > maxCount then
13 |  |  maxCount ← currentCount;
14 |  |  maxFrequencyElement ← array[i];
15 |  end
16 end
17 return maxFrequencyElement;
```

Zunächst wird das zu durchsuchende Array der Länge n mit einem Sortieralgorithmus der Komplexität $\mathcal{O}(n \log n)$ aufsteigend sortiert (hier: Mergesort). Anschließend wird über das gesamte sortierte Array mit der Komplexität $\mathcal{O}(n)$ iteriert und gezählt, wie häufig ein selbes Element aufeinanderfolgt.
Es ergibt sich somit eine Gesamtlaufzeit von $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n + n) = \mathcal{O}(n \log n)$.

c)

Wird A nur auf natürliche Zahlen beschränkt, ändert sich weder was an der Laufzeit des Sortieralgorithmus noch an der Laufzeit der Operation, die in a) zur Bestimmung des Minimums benötigt wird und in b) zur Bestimmung des am häufigsten vorkommenden Elements genutzt wird. Die Einschränkung hat somit keine Auswirkung auf die Laufzeit.