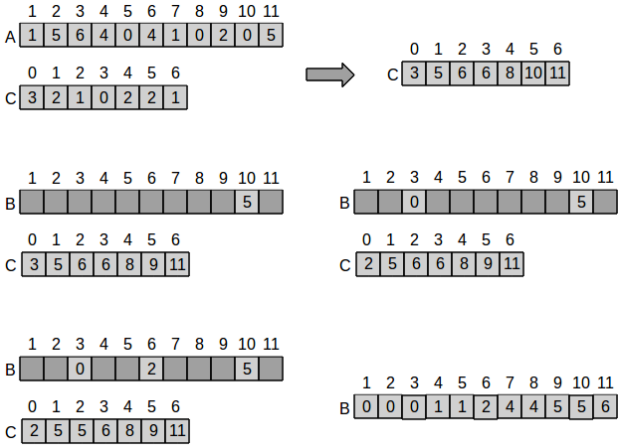
Theoretische Aufgaben

Aufgabe 1

Der CountingSort-Algorithmus ist relativ leicht zu erklären, dies kann in 4 Schleifen geschehen:

1. Die erste Schleife stellt ein leeres Feld C bereit, von 0 bis zur höchsten vorkommenden Zahl.
2. Die zweite Schleife zählt iterierend, wieviel mal j vorkommt und speichert dies in $C[j]$.
3. Die dritte Schleife zählt die für die Elemente 0 bis i von C fortlaufend zusammen und speichert den Wert bei jedem Durchgang in $C[i]$.
4. Die vierte Schleife nimmt den obersten Wert von der Eingabe A, setzt ihn als Index in C ein. Dort steht der Index von B, wo der oberste Wert von A abgespeichert werden soll. Dies wiederholt sich, bis alle Elemente eingefügt wurden.



Aufgabe 2

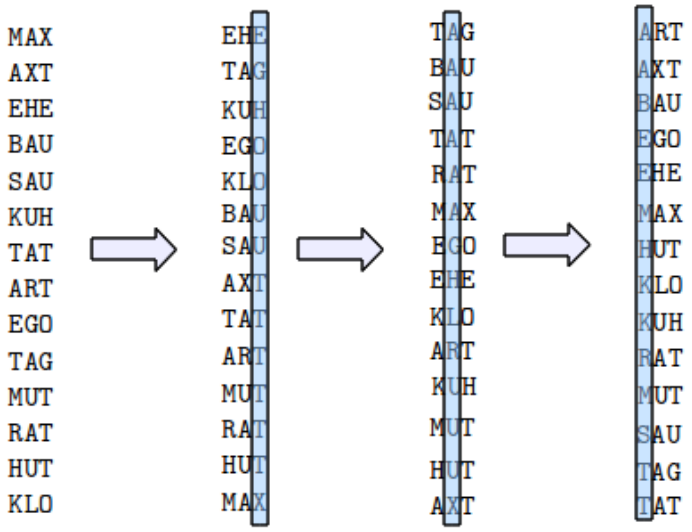
Listing 1: Aufgabe 2

```
// bereitet die Abfrage vor
void prepare(A, k)
int C[k]
for i = 0 to k
  C[i] = 0
for j = 1 to A.length
  C[A[j]] = C[A[j]] + 1
for i = 1 to k
  C[i] = C[i] + C[i - 1]

// Abfrage
int query(a, b)
if a = 0
  return C[b]
else
  return C[b] - C[a-1]
```

Aufgabe 3

Die einzelnen Elemente werden mit dem hintersten Buchstaben beginnend buchstabenweise sortiert. Das heisst, es wird zuerst bei allen der hinterste, dann der mittlere, und zum Schluss nach dem vordersten Buchstaben sortiert.

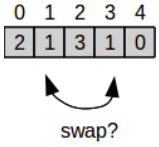


Aufgabe 4

Stabilität der Sortieralgorithmen:

- InsertionSort: Ja. Elemente werden nicht mehr ausgeatuscht, wenn nächstvorderes Element gleichgross ist.
- MergeSort: Ja. bei gleichen Elementen wird zuerst aus dem ersten Array das (ursprünglich vordere) Element eingesetzt, somit wird die Reihenfolge erhalten.
- HeapSort: Nein. Aufgrund der Verteilung auf verschiedene Teilbäume kommt es auf die anderen Zahlen an, ob zwei gleiche Elemente vertauscht werden oder nicht.
- QuickSort: Nein. Durch den Austausch rund um die fortschreitende Zählvariable (und wo bestimmt wird, ob die folgende Zahl grösser oder kleiner dem Pivot ist) werden die Elemente so 'herumgeschoben', dass es nach einem Durchlauf möglich ist, dass zwei gleiche Elemente in umgekehrter Reihenfolge anzutreffen sind.

Aufgabe 5



Bei einem Austausch kann bei Elementen zusätzlich noch der Index selbst verwendet werden, um sicherzustellen, dass der Algorithmus stabil läuft. Dabei wird bei einem Vergleich ein einfacher Test $i_1 < i_2$ gemacht, um zu verhindern, dass ein `swap` mit gleichen Elementen stattfindet.

Aufgabe 6

Angenommen, es gibt unter den Zahlen k unterschiedliche Längen. Für jede Länge k_x gilt, dass sie zuerst sortiert werden muss, und die verschiedenen Teilergebnisse für die Längen $k_{1,2,3,\ldots}$ können dann aneinandergereiht werden. Die Sortierdauer für alle Elemente der Länge k dauert der Sortiervorgang $\frac{n}{k_1}$.
Zusammengezählt ergeben die Laufzeiten für die verschieden langen Elemente die Gesamtlaufzeit:

$$O\left(\frac{n}{k_1}\right) + O\left(\frac{n}{k_2}\right) + O\left(\frac{n}{k_3}\right) + \cdots + O\left(\frac{n}{k_n}\right) \tag{1}$$

$$\Rightarrow O\left(\frac{n}{k_1 + k_2 + \cdots + k_n}\right) \tag{2}$$

$$\Rightarrow O\left(\frac{n}{1}\right) = O(n) \tag{3}$$

Praktische Aufgaben

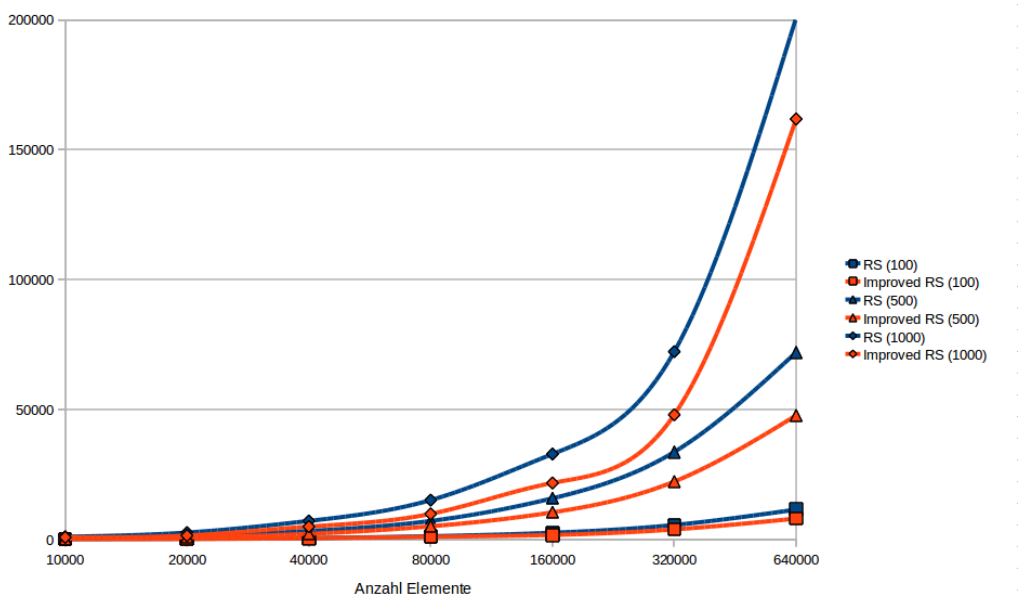
Aufgabe 1

Äussere Schleife: $\Theta(d)$. Innere Schleife: $n \cdot \Theta(1)$. Gesamter Algorithmus: $\Theta(d * n)$

Aufgabe 2

Siehe Beilagen `RadixSort.Java`, `RadixMiniTest.out`, `RadixSortTester.out`.

Aufgabe 3



Die Parameter in der Grafik geben geben den Wert von d aus, der auf 100, 500 und 1000 eingegeben wurde.
Die Laufzeit ist um 20-30% verbessert worden (siehe `RadixSortTester.out` für die exakten Laufzeiten), scheint jedoch noch nicht linear zu sein, sondern um einen Faktor kleiner als die Zeit für den 'normalen' radixSort. Evtl. spielen auch die gewählten Datenstrukturen eine Rolle, welche noch optimaler gewählt werden könnten.