Exercise 1: Bucket Sort

*Bucket sort* is an algorithm to stably sort an array $A[0..n-1]$ of $n$ elements where the sorting keys of the elements take values in $\{0, \ldots, k\}$. That is, we have a function $\text{key}$ assigning a key $\text{key}(x) \in \{0, \ldots, k\}$ to each $x \in A$.

The algorithm works as follows. First we construct an array $B[0..k]$ consisting of (initially empty) FIFO queues. That is, for each $i \in \{0, \ldots, k\}$, $B[i]$ is a FIFO queue. Then we iterate through $A$ and for each $j \in \{0, \ldots, n-1\}$ we attach $A[j]$ to the queue $B[\text{key}(A[j])]$ using the function $\text{enqueue}$.

Finally we empty all queues $B[0], \ldots, B[k]$ using $\text{dequeue}$ and write the returned values back to $A$, one after the other. After that, $A$ is sorted with respect to $\text{key}$ and elements $x, y \in A$ with $\text{key}(x) = \text{key}(y)$ are in the same order as before.

Implement *Bucket sort* based on this description. You can use the template *BucketSort.py* which uses an implementation of FIFO queues that are available in *Queue.py* und *ListElement.py*.\(^1\)

Sample Solution

Cf. BucketSort.py in the public repository.

Exercise 2: Radix Sort

(a) Implement *Radix sort* based on this description. You may assume $b = 10$, i.e., your algorithm should work for arrays containing numbers in base-10 representation. Use *Bucket sort* as a subroutine. If you did not solve task 1, you may use a library function (e.g., *sorted*) as alternative to *Bucket sort*.\(^2\)

(b) Compare the runtimes of *Bucket sort* and *Radix sort*. For both algorithms and each $k \in \{2 \cdot i \cdot 10^4 \mid i = 1, \ldots, 60\}$, use an array of fixed size $n = 10^4$ with randomly chosen keys from $\{0, \ldots, k\}$ as input and plot the runtimes. Shortly discuss your results in experiences.txt.\(^3\)

(c) Explain the asymptotic runtime of your implementations of *Bucket sort* und *Radix sort* depending on $n$ and $k$.\(^3\)

---

\(^1\)Remember to make unit-tests and to add comments to your source code.

\(^2\)You are allowed to use libraries, but note that the names of the methods may differ.

\(^3\)The $i$-th digit $c_i$ of a number $x \in \mathbb{N}$ in base-$b$ representation (i.e., $x = c_0 \cdot b^0 + c_1 \cdot b^1 + c_2 \cdot b^2 + \ldots$), can be obtained via the formula $c_i = (x \mod b^{i+1}) \div b^i$, where $\mod$ is the modulo operation and $\div$ the integer division.
Sample Solution

(a) Cf. RadixSort.py in the public repository.

(b) Cf. 1. We see that BucketSort is linear in $k$. For Radixsort the situation is not that clear. At the first sight, the runtime could be constant, but upon closer examination we see steps at $k = 10^5$ and $k = 10^6$. The reason is that Radixsort calls BucketSort for each digit in the input and the number of these digits (and therefore the calls of BucketSort) is increased from 5 to 6 at $k = 10^5$ (respectively 6 to 7 at $k = 10^6$). This is also the reason why BucketSort is faster for small $k$ (the runtimes are roughly even when $n \log_{10}(k) = n + k$ holds).

(c) BucketSort goes through $A$ twice, once to write all values from $A$ into the buckets and another time to write the values back to $A$. This takes time $O(n)$ as writing a value into a bucket and from a bucket back to $A$ costs $O(1)$. Additionally, BucketSort needs to allocate $k$ empty lists and write it into an array of size $k$ which takes time $O(k)$. Hence, the runtime is $O(n + k)$.

RadixSort calls BucketSort for each digit. The keys have $m = O(\log k)$ digits, so we call BucketSort $O(\log k)$ times. One run of BucketSort takes $O(n)$ here as the keys according to which BucketSort sorts the elements are from the range \{0, \ldots, 9\}. The overall runtime is therefore $O(n \log k)$.

Abb. 1: Plot for exercise 2 b).