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 $\text{BucketSort}$ based on this description. You can use the template $\text{BucketSort.py}$ which uses an implementation of FIFO queues that are available in $\text{Queue.py}$ and $\text{ListElement.py}$.

An example of usage of this template is the following:

```python
from Queue import Queue
from ListElement import ListElement
q = Queue()
qu.enqueue(ListElement(5))
qu.enqueue(ListElement(17))
qu.enqueue(ListElement(8))
while not q.is_empty():
    print(q.dequeue().get_key())
```

This would print the numbers 5, 17, 8 on three separate lines.

Solution:

```python
def bucket_sort(array, k, key=lambda x: x):
    
    Implements the bucket sort algorithm to sort
    data elements using a key function to
    assign sorting keys to data elements

    Args:
    array: array of data elements
    k: largest key
    key: a function mapping data elements to values
    in range(k+1) (identity function as default)
```

1Remember to make unit-tests and to add comments to your source code.
Exercise 2: Radix Sort

Assume we want to sort an array $A[0..n−1]$ of size $n$ containing integer values from $\{0,\ldots,k\}$ for some $k \in \mathbb{N}$. We describe the algorithm $\text{Radixsort}$ which uses $\text{BucketSort}$ as a subroutine.

Let $m = \lceil \log_b k \rceil$. We assume each key $x \in A$ is given in base-$b$ representation, i.e., $x = \sum_{i=0}^{m} c_i \cdot b^i$ for some $c_i \in \{0,\ldots,b−1\}$. First we sort the keys according to $c_0$ using $\text{BucketSort}$, afterwards we sort according to $c_1$ and so on.\(^2\)

(a) Implement $\text{Radixsort}$ based on this description. You may assume $b = 10$, i.e., your algorithm should work for arrays containing numbers in base-10 representation. Use $\text{BucketSort}$ as a subroutine.

(b) Compare the runtimes of $\text{Bucketsort}$ and $\text{Radixsort}$. For both algorithms and each $k \in \{i \cdot 10^4 \mid i = 1,\ldots,50\}$, use an array of size $10^4$ with randomly chosen keys from $\{0,\ldots,k\}$ as input and plot the runtimes. Shortly discuss your results.

(c) Explain the asymptotic runtime of your implementations of $\text{Bucketsort}$ and $\text{Radixsort}$ depending on $n$ and $k$.

Solution:

(a) \textbf{def} \texttt{radix\_sort} (\texttt{array}, \texttt{k}):
\begin{verbatim}
  
  # add your code here
  bucket = [Queue() for i in range(k+1)]
  for i in range(len(array)):
    bucket[key(array[i])].enqueue(ListElement(array[i]))
  i = 0
  for j in range(k+1):
    while not bucket[j].is_empty():
      array[i] = bucket[j].dequeue().get_key()
      i += 1
  return array

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 \texttt{mod} is the modulo operation and \texttt{div} the integer division.

\(2\)
for i in range(m+1):
    key = lambda x: (x % 10**(i+1)) // 10**i
BucketSort.bucket_sort(array, 10, key)
return array

(b) See Figure 1. We see that Bucket sort is linear in \(k\). For Radix sort the situation is not that clear. At the first sight, the runtime could be constant, but upon closer examination (see Figure 2) we see a step at \(k = 10^5\). The reason is that Radix sort calls Bucket sort for each digit in the input and the number of these digits (and therefore the calls of Bucket sort) is increased from 5 to 6 at \(k = 10^5\).

(c) Bucket sort 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, Bucket sort needs to allocate \(k\) empty lists and write it into an array of size \(k\) which takes time \(O(k)\). Hence, the runtime is \(O(n + k)\).

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

![Figure 1: Plot for exercise 2 b)](image-url)
Figure 2: Considering a larger range of keys to visualize the second step at $10^6$. 