Exercise 1: Bucket Sort

_Bucketsort_ 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 _Bucketsort_ based on this description. You can use the template _BucketSort.py_ which uses an implementation of FIFO queues that are available in _Queue.py_ and _ListElement.py_.

An example of usage of this template is the following:

```python
from Queue import Queue
from ListElement import ListElement
q = Queue()
q.enqueue(ListElement(5))
q.enqueue(ListElement(17))
q.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)
```
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 $m = \lfloor \log_b k \rfloor$. 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) \(\text{def } \text{radix_sort}(\text{array}, k):\)

\[\text{Implements the radix sort algorithm to sort data elements with keys in range}(k+1)\]

\[\text{Args:}\]

\[\text{array: array of data elements}\]

\[k: \text{largest key}\]

\[\text{>>> } \text{radix_sort}([123, 1111, 789, 456, 0, 12, 13, 247], 2000)\]

\[[0, 12, 13, 123, 247, 456, 789, 1111]\]

\[\text{>>> } \text{radix_sort}([1000 - i \text{ for } i \text{ in range}(0, 1000)], 1000) == \\]

\[[i \text{ for } i \text{ in range}(1, 1001)]\]

\[\text{True}\]

\[\text{>>> } \text{radix_sort}([10 - i \text{ for } i \text{ in range}(10)], 10)\]

\[[]\]

\[\text{>>> } \text{bucket_sort}([203, 307, 210, 121, 420], 2, \lambda x: \text{int}(x / 10) \% 10)\]

\[[]\]

\[\text{>>> } \text{bucket_sort}([], 10)\]

\[[]\]

\[\text{>>> } \text{bucket_sort}([10 - i \text{ for } i \text{ in range}(10)], 10)\]

\[[]\]

\[\text{Exercise 2: Radix Sort}\]

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

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

(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) \(\text{def } \text{radix_sort}(\text{array}, k):\)

\[\text{Implements the radix sort algorithm to sort data elements with keys in range}(k+1)\]

\[\text{Args:}\]

\[\text{array: array of data elements}\]

\[k: \text{largest key}\]

\[\text{>>> } \text{radix_sort}([123, 1111, 789, 456, 0, 12, 13, 247], 2000)\]

\[[0, 12, 13, 123, 247, 456, 789, 1111]\]

\[\text{>>> } \text{radix_sort}([1000 - i \text{ for } i \text{ in range}(0, 1000)], 1000) == \\]

\[[i \text{ for } i \text{ in range}(1, 1001)]\]

\[\text{True}\]

\[\text{>>> } \text{radix_sort}([10 - i \text{ for } i \text{ in range}(10)], 10)\]

\[[]\]

\[\text{>>> } \text{bucket_sort}([203, 307, 210, 121, 420], 2, \lambda x: \text{int}(x / 10) \% 10)\]

\[[]\]

\[\text{>>> } \text{bucket_sort}([], 10)\]

\[[]\]

\[\text{>>> } \text{bucket_sort}([10 - i \text{ for } i \text{ in range}(10)], 10)\]

\[[]\]

\[\text{Exercise 2: Radix Sort}\]

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 $m = \lfloor \log_b k \rfloor$. 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) \(\text{def } \text{radix_sort}(\text{array}, k):\)

\[\text{Implements the radix sort algorithm to sort data elements with keys in range}(k+1)\]

\[\text{Args:}\]

\[\text{array: array of data elements}\]

\[k: \text{largest key}\]

\[\text{>>> } \text{radix_sort}([123, 1111, 789, 456, 0, 12, 13, 247], 2000)\]

\[[0, 12, 13, 123, 247, 456, 789, 1111]\]

\[\text{>>> } \text{radix_sort}([1000 - i \text{ for } i \text{ in range}(0, 1000)], 1000) == \\]

\[[i \text{ for } i \text{ in range}(1, 1001)]\]

\[\text{True}\]

\[\text{>>> } \text{radix_sort}([10 - i \text{ for } i \text{ in range}(10)], 10)\]

\[[]\]

\[\text{The } i\text{-th digit } c_i \text{ of a number } x \in \mathbb{N} \text{ in base-} b \text{ representation (i.e, } x = c_0 \cdot b^0 + c_1 \cdot b^1 + c_2 \cdot b^2 + \ldots), \text{ can be obtained via the formula } c_i = (x \text{ mod } b^{i+1}) \text{ div } b^i, \text{ where mod is the modulo operation and div the integer division.}\]
\[ m = \text{math.ceil}(\text{math.log}(k, 10)) \]

```python
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 \textit{Bucketsort} is linear in \( k \). For \textit{Radixsort} 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 \textit{Radixsort} calls \textit{Bucketsort} for each digit in the input and the number of these digits (and therefore the calls of \textit{Bucketsort}) is increased from 5 to 6 at \( k = 10^5 \).

(c) \textit{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, \textit{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 \( O(n + k) \).

\textit{RadixSort} calls \textit{Bucketsort} for each digit. The keys have \( m = O(\log k) \) digits, so we call \textit{Bucketsort} \( O(\log k) \) times. One run of \textit{Bucketsort} takes \( O(n) \) here as the keys according to which \textit{Bucketsort} 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).
Figure 2: Considering a larger range of keys to visualize the second step at $10^6$. 