Algorithms and Datastructures
Summer Term 2020
Sample Solution Exercise Sheet 3
Due: Wednesday, 3rd of June, 4 pm.

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 $key$ assigning a key $key(x) \in \{0, \ldots, k\}$ 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 $B[key(A[j])]$ using the function $enqueue$.

Finally we empty all queues $B[0], \ldots, B[k]$ using $dequeue$ and write the returned values back to $A$, one after the other. After that, $A$ is sorted with respect to $key$ and elements $x, y \in A$ with $key(x) = key(y)$ 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 $Queue.py$ und $ListElement.py$.

Sample Solution

Cf. $BucketSort.py$ in the public repository.

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 $RadixSort$ which uses $BucketSort$ 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 $BucketSort$, afterwards we sort according to $c_1$ and so on.

(a) Implement $RadixSort$ based on this description. You may assume $b = 10$, i.e., your algorithm should work for arrays containing numbers in base-10 representation. Use $BucketSort$ as a subroutine. If you did not solve task 1, you may use a library function (e.g., sorted) as alternative to $BucketSort$. (6 Points)

(b) Compare the runtimes of $BucketSort$ and $RadixSort$. For both algorithms and each $k \in \{i \cdot 10^4 | i = 1, \ldots, 50\}$, 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 in $erfahrungen.txt$. (4 Points)

(c) Explain the asymptotic runtime of your implementations of $BucketSort$ und $RadixSort$ depending on $n$ and $k$. (3 Points)

1Remember to make unit-tests and to add comments to your source code.

2The $i$-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. 2. 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 a step at \( k = 10^5 \). 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 \( k = 10^5 \).

(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 \( \mathcal{O}(n) \) as writing a value into a bucket and from a bucket back to \( A \) costs \( \mathcal{O}(1) \). Additionally, Bucketsort needs to allocate \( k \) empty lists and write it into an array of size \( k \) which takes time \( \mathcal{O}(k) \). Hence, the runtime is \( \mathcal{O}(n + k) \).

RadixSort calls Bucketsort for each digit. The keys have \( m = \mathcal{O}(\log k) \) digits, so we call Bucketsort \( \mathcal{O}(\log k) \) times. One run of Bucketsort takes \( \mathcal{O}(n) \) here as the keys according to which Bucketsort sorts the elements are from the range \( \{0, \ldots, 9\} \). The overall runtime is therefore \( \mathcal{O}(n \log k) \).
Abb. 2: Considering a larger range of keys to visualize the second step at $10^6$. 