University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
G. Schmid

Algorithms and Data Structures – Conditional Course Winter Term 2026/27

Sample Solution Exercise Sheet 3

Due: Wednesday, 11 November 2026, 12:00

Exercise 1: Bucket Sort *(7 Points)*

Bucket Sort is a simple algorithm for *stably* sorting an array $A[0 \dots n-1]$ of n data items whose sorting keys lie in $\{0, \dots, k\}$. A function `key` assigns each item $x \in A$ a key $\texttt{key}(x) \in \{0, \dots, k\}$.
First, create an array $B[0 \dots k]$ of FIFO queues,[1] where $B[i]$ is the ith queue. Iterate through A. If the current item $A[j]$ has key $i := \texttt{key}(A[j])$, append it to $B[i]$ using `enqueue`.
Finally, empty the queues $B[0], \dots, B[k]$ using `dequeue`, writing the returned items back to A in that order. The array is now sorted by `key`. Items x, y with equal keys retain their relative order: this is what makes the sort *stable*.
Implement Bucket Sort from this description using `BucketSort.py`. FIFO queue implementations are supplied in `Queue.py` and `ListElement.py`.[2]

Sample Solution

The idea is to collect all items with the same key in one queue. We then empty these queues in increasing key order. For example, all items with key 2 are written back before any item with key 3. Items with equal keys leave their queue in the order in which they entered it, so their relative order is preserved.
The implementation is in `BucketSort.py` in the solution ZIP. It creates $k+1$ queues, visits the input from left to right, and puts each item into the queue selected by `key`. When emptying the queues, a single index keeps track of the next position to fill in the input array. The supplied queue stores `ListElement` objects, so we wrap each item when inserting it and retrieve its value with `get_key()` when removing it.

[1] In a FIFO (First In, First Out) queue, items are removed in the order in which they were inserted.
[2] You may instead use a library queue, but note that its method names may differ.

Exercise 2: Radix Sort *(13 Points)*

Let $A[0 \dots n-1]$ contain integers from $\{0, 1, \dots, k\}$ for some $k \in \mathbb{N}$. Our implementation of *Radix Sort* uses the stable Bucket Sort algorithm as a subroutine.
Let $m = \lfloor \log_b k \rfloor$. Represent every key $x \in A$ in base b as $x = \sum_{i=0}^{m} c_i b^i$. You may use $b = 10$, so that the c_i are decimal digits. First use Bucket Sort to sort by the least significant digit c_0, then by c_1, and so on up to c_m.[3] The resulting array is sorted.
Example: Let $A = [329, 457, 657, 839, 436, 720, 355, 009]$ and $b = 10$. Sorting by c_0 gives

$$A = [720, 355, 436, 457, 657, 329, 839, 009].$$

Stability preserves the relative order of items with the same digit: for example, 839 still precedes 009. Sorting by c_1 gives

$$A = [009, 720, 329, 436, 839, 355, 457, 657].$$

Finally, sorting by c_2 gives

$$A = [009, 329, 355, 436, 457, 657, 720, 839].$$

(a) Implement Radix Sort from this description, using your Bucket Sort implementation as a subroutine. *(7 Points)*

(b) Compare the running times of your Bucket Sort and Radix Sort implementations. Use arrays of size $n = 10^4$ containing random keys from $\{0, 1, \dots, k\}$. Plot the running times for $k \in \{2i \cdot 10^4 \mid i = 1, \dots, 60\}$; helper functions are provided in the templates. Briefly discuss the results in `experience.txt`. *(3 Points)*

(c) Give and justify the asymptotic running times of both implementations as functions of n and k. *(3 Points)*

[3] The ith digit of x in base b is $c_i = (x \bmod b^{i+1}) \texttt{ div } b^i$, where `div` denotes integer division.

Sample Solution

(a) Implementation. The implementation is in `RadixSort.py` in the solution ZIP. We use a variable `place` with values $1, 10, 100, \ldots$. The expression `(x // place) % 10` extracts the corresponding digit. For each digit, we call Bucket Sort with keys from 0 to 9. We stop after processing the highest digit of k. For $k = 0$, one pass is enough.
Why start with the units digit? After that pass, the units are in order. Sorting by the tens digit groups equal tens together; stability keeps the units in order within each group. Thus the last two digits are now in order. The same idea applies to each further digit, until the whole numbers are sorted.

(b) Measurements. Figure 1 shows the mean of three runs for each requested value of k, using $n = 10^4$ random integers. Both algorithms receive copies of the same input in each run; input generation is not timed.
Bucket Sort becomes slower as k grows because it creates and visits more queues, even though the number of input items stays the same. Radix Sort grows much more slowly: it still uses only ten queues per pass. It needs an extra pass when k reaches 10^5 and again at 10^6. Timing noise can obscure these small changes. The exact times depend on the computer and implementation.

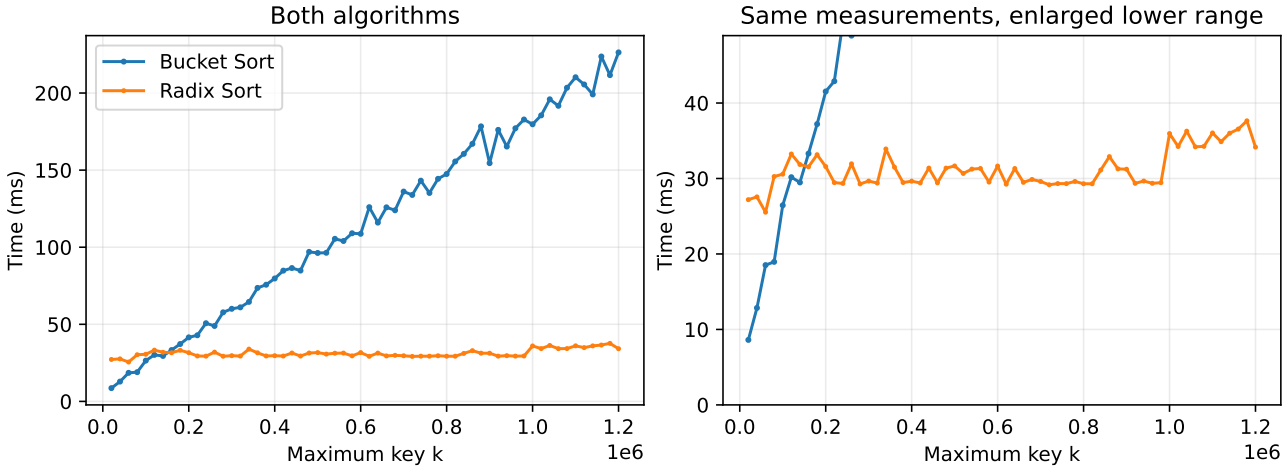


Figure 1: Measured running times; the right panel enlarges the lower range.

(c) Running times. Bucket Sort creates and visits $k + 1$ queues. Each of the n items is inserted and removed once, and each queue operation takes constant time. Its total running time is therefore $O(n + k + 1)$, usually written $O(n + k)$.
For Radix Sort, one pass costs $O(n + 10)$: there are n items and ten possible digits. For $k \geq 1$, there are $d = \lfloor \log_{10} k \rfloor + 1$ passes, so the total is $O((n + 10)d)$. For $n \geq 1$ and $k \geq 2$, this is $O(n \log k)$. If $k = 0$, we use one pass. As in the lecture, this analysis counts arithmetic on a key as a constant-time operation.