JOHANNES KEPLER
UNIVERSITÄT LINZ

Algorithms and Data Structures 1
*Summer term 2021*
*Dari Trendafilov*
*Stefan Grünberger*

# Assignment 6

Deadline: **Thu 17.6.2021, 23:59**
Submission via: **Moodle**

### Elaboration time

*Remember the time you need for the elaboration of this assignment and document it in the file **time.txt** according to the structure illustrated in the right box. Please do not pack this file into an archive but upload it as a **separate file**.*

```
#Student ID
K12345678
#Assignment number
06
#Time in minutes
190
```

# Sorting

Stick to the given interfaces and skeletons and don't forget to test your code!

## 1. HeapSort — 10 points

Implement the **HeapSort** algorithm based on the skeleton in **MaxHeap.py**. Create the heap with the **in-place bottom-up** construction approach without using temporary data structures.

The following methods shall be implemented:

```
class MaxHeap:
    def __init__(self, list):
        # Creates a bottom-up maxheap in-place from the input list of numbers.

    def contains(self, val):
        # Tests if val is in the heap. Do not search the array sequentially, but use the heap properties.

    def is_empty(self):
        # return True if the heap is empty, False otherwise.

    def remove_max(self):
        # Removes and returns the maximum element of the heap.

    def sort(self):
        # This method sorts the numbers in-place using the maxheap data structure in ascending order.
```

a) The **MaxHeap()** constructor should create a valid maxheap using **in-place *bottom-up*** *construction*.

b) The method **contains()** should return *true*, if the element *val* is found in the heap (duplicates allowed). Use the properties of the maxheap for efficient implementation and do not search the array sequentially!

c) The **sort** method should implement the **HeapSort** algorithm in ***ascending*** order, using the created maxheap by repeated removal of the top element **in-place**. **Don't** use intermediate data structures!

You can use private (help) methods for your implementation. For this task submit your **MaxHeap.py**.

## 2. RadixSort — 14 points

Implement the **direct RadixSort** (base 7) for ***ascending*** sorting of **positive** Integer numbers.

- Describe the runtime complexity of your algorithm in the function comment of the class `RadixSort`.
- Implement the method **`sort()`**, which takes a list of **positive** Integer numbers and sorts them in ascending order using the RadixSort algorithm.
  - after every iteration (when all numbers are assigned to buckets) append the buckets content into a list of 7 lists and
  - use the provided method **`_add_bucket_list_to_history`** `(...)` to add the new bucket list to history for testing purposes
- Test your implementation by using (and extending) the provided unit tests. For your support we provide the method `print_bucket_list_history` in the unit test class, that can be used after the sort() method has been executed, to print the content of the bucket lists in a formatted way.

JOHANNES KEPLER
UNIVERSITÄT LINZ

Algorithms and Data Structures 1
*Summer term 2021*
*Dari Trendafilov*
*Stefan Grünberger*

# Assignment 6

Deadline: **Thu 17.6.2021, 23:59**
Submission via: **Moodle**

```python
# Runtime Complexity O(...)
class RadixSort:
    def __init__(self):

    def get_bucket_list_history(self):
        return self.bucket_list_history

    def sort(self, list):
        # Sorts a given list using radixsort in ascending order and returns the sorted list

    def _add_bucket_list_to_history(self, bucket_list):
        # This method creates a snapshot (clone) of the bucketlist and adds it to the bucketlistHistory.
        @param bucket_list is your current bucketlist, after assigning all elements to be sorted to the buckets.
```

For this task submit your **RadixSort.py**.