Prof. Dr. Thomas Schultz
Olivier Morelle (morelle@uni-bonn.de)

B-IT
Bonn-Aachen International
Center for Information Technology
December 8, 2024

Winter term 2024/25

Computer Science for Life Scientists

Assignment Sheet 10

Solution has to be uploaded by December 18, 2024, 10:00 a.m., via eCampus

- This exercise can be submitted in **small groups** of 2-3 students. Submit each solution only once, but clearly indicate who contributed to it by forming a team in eCampus. Remember that all team members have to be able to explain all answers.
- Remember to include proper **documentation** in all your code, in particular, docstrings for functions.
- Please submit your answers as a single file in .ipynb or .pdf format.

If you have questions concerning the exercises, please use the forum on eCampus.

Exercise 1 (Hash Tables, *9 Points*)

a) You have a universe of 30 integers $\{0, 1, 2, ..., 29\}$ and a hash-table of size 10. Which hash-function is better, $h_1(k) = k \bmod 10$, or $h_2(k) = \lfloor k/10 \rfloor$? Why? (2P)

b) When using strings as keys in a hash table, it is common to first convert them to integers, and then apply a hash function that works for integers. To preserve computational efficiency, integers should be in $[0, M-1]$, where M is a power of two (e.g., on a 64 bit system, $M = 2^{64}$). Following are two methods to map a string S to such an integer. They make use of the helper function *ord*, which maps an individual character to its integer representation, which you can assume to be in $[0, 255]$.

```
def H1(S):
  val = 0
  for i in range(0,len(S)):
        val = (val + ord(S[i])) % M
  return val
```

```
def H2(S):
  val = 0
  for i in range(0,len(S)):
        val = (val*31 + ord(S[i])) % M
  return val
```

Why might one prefer the second algorithm over the first one? (1P) In the second algorithm, what might be a reason for using 31 as the multiplying factor (rather than, say, 32)? (2P)

c) Chaining and open addressing are two methods that are used to handle collisions in hash tables. Which one would you prefer for a use case in which most items that are added to the hash table will be deleted again shortly afterwards? Why? (2P)

d) The implementation of insertion with probing in the lecture did not check if the key is already present. In such a case, we could either raise an error, or update the value corresponding to the existing key. Specify suitable pseudo code for both of these two options. (2P)

Exercise 2 (Checking a directed graph for cycles, *12 Points*)

In the lecture, we discussed how we can check whether or not a graph contains any cycles. In this exercise, you will first implement a graph class in Python, and learn about a simple way of visualizing graphs. You will then write code for detecting cycles in graph, and highlighting them in the visualization.

a) Write a Python class that represents directed graphs with an adjacency list implementation. Your class should provide the following functionality: (6P)
   - Adding nodes
   - Adding edges
   - Deleting nodes
   - Deleting edges
   - For a given node, iterating over all successor nodes

b) Modify the `__init__` function of your graph class so that it can optionally initialize an instance based on the information given in a text file, which has the following format:
   - The first line contains a single nonnegative integer number, which specifies the overall number of nodes in the graph.
   - Each of the remaining lines contains two nonnegative integers i and j, separated by a space. They represent an edge (i, j), where node indices start at zero.

   We provide two example files in this format, `graph-06.txt` and `graph-20.txt`. (2P)

c) Add a function to your graph class that detects whether or not the graph is acyclic. If the graph is not acyclic, your function should prove this by returning a cycle, as a list of nodes. Try this out on the two graphs `graph-06.txt` and `graph-20.txt`. (4P)

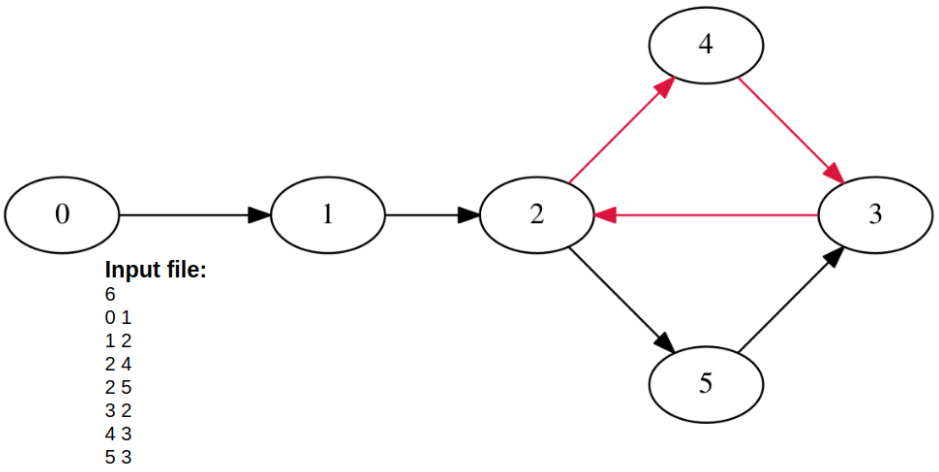

Figure 1: A visualization of a directed graph in which a cycle is highlighted.

Exercise 3 (Breadth First Trees, *4 Points*)

Draw two breadth-first trees of the graph shown in Figure 2. The first tree should have node 1 as its root, the second one node 3.

Good Luck!

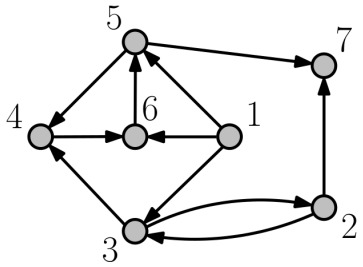

Figure 2: The graph on which to perform breadth first search.