Exercise 4.1  Lower bounds / Algorithm design.

Consider a set of $n$ coins that contains exactly one false one. This false coin is heavier than all other ones. To find the false coin you can only use a balance scale. Using this you can only determine whether the coins on the left are lighter, heavier, or have exactly the same weight as the coins on the right.

a) Design a strategy for $n = 9$ coins that uses as few weighings as possible.

b) For a general $n$, provide an algorithm that uses in the worst case exactly $\log_3(n)$ many weighings.

c) Show that even the best algorithm will use at least $\log_3(n) - 1$ many weighings in the worst case.

Note: You can assume that $n$ is a power of 3.

Exercise 4.2  Natural Search Trees.

a) Draw the resulting tree if you insert the keys 6, 2, 1, 4, 7, 9, 3, 8, 10, 11, 5 in this order into an initially empty natural search tree.

b) Give the preorder, postorder and inorder traversal of the tree in a).

c) Remove the key 11 from the tree in a) and after that the key 2 from the resulting tree. Draw both trees.

d) Draw the binary search tree with the preorder traversal 9, 2, 7, 5, 8, 11, 20, 15.

Exercise 4.3  Advanced Search Trees.

In this exercise, we wish to extend the functionality of a natural search tree. We assume that the tree maintains integers. Modify the search tree such that we can not only find a number, but are also able to answer the following queries.

a) How many elements in the tree are even and smaller than a given number $k$?

b) How many even elements in the tree are (strictly) between two given numbers $k_1$ and $k_2$ with $k_1 < k_2$?

Discuss which additional information every node has to store and how the insertion and deletion operations have to be changed such that the above queries can be answered efficiently. Provide the running times of all operations.

Please turn over.
Exercise 4.4  Programming Exercise: Natural Search Trees.

In this exercise we are going to implement two operations for natural search trees.

The operation **Insert** works recursively: Inserting an element into an empty tree results in a tree whose root is the element itself. To insert the element \( k \) into a non-empty tree, we compare the key of the root with \( k \).

If \( k \) is smaller than the key of the root, then we continue recursively on the subtree rooted at the left child. If \( k \) is larger, then we continue recursively on the subtree rooted at the right child. In case of equality, we do not insert further keys (of the corresponding testcase) and simply output the error message **DUPLICATE-KEY**. The insertion of the keys 9, 7, 5, 11, 8, 15 in this order results in the search tree depicted above.

The operation **Visit** performs a post-order traversal of the tree (we first visit the left subtree, then the right subtree and finally the root). For the above tree, the nodes are considered in the order 5, 8, 7, 15, 11, 9. For each node that is visited, we print the key stored in that node and the depths of the left and the right subtree of the node (in exactly this order).

**Input**  The first line contains only the number \( t \) of test instances. After that, we have exactly one line per test instance containing the numbers \( n, k_1, \ldots, k_n \). While \( n \in \mathbb{N}, 1 \leq n \leq 1000 \), describes the number of following integers, \( k_i \in \mathbb{Z}, -10^8 \leq k_i \leq 10^8 \) is the \( i \)-th key to be inserted into the search tree.

**Output**  For every test instance we output only one line. It contains the result of the **Visit** operation after every element of the corresponding sequence has been inserted into the natural search tree. If the testcase contains some key multiple times, we simply want to output the text **DUPLICATE-KEY**.

**Example**

\begin{verbatim}
Eingabe:
3
2 1 1
5 1 2 3 4 5
6 9 7 5 11 8 15

Ausgabe:
DUPLICATE-KEY
5 0 0 4 0 1 3 0 2 2 0 3 1 0 4
5 0 0 8 0 0 7 1 1 5 0 0 11 0 1 9 2 2
\end{verbatim}

**Hand-in:** until Wednesday, 18th March 2015.