1. Recursive functions

What is the base case of the recursion?

Consider the following recursive program:

```python
def rec(n):
    if n == 1:
        s = 0
    else:
        if n % 2 == 0:
            s = n + rec(n // 2)
        else:
            s = n + rec(n + 1)
    return s
print(rec(22))
```

Enter this program in your IDE and run the debugger: in Thonny select "Run" → "Debug current script (nicer)" and step through the program with "Step into (F7)". You see that the function calls itself several times. Write below all arguments with which the function calls itself. (For example, the first argument is 22.)

Fill the blanks in such a way that the function works as shown in the right:

```python
def fun(x, depth):
    if ________________:
        fun(______________, ____________)
    print(x)
```

```plaintext
>>> fun(6, 4)
9
8
7
6
```
The following program is similar to the previous program but there is one crucial difference. Fill the blanks in such a way that the function works as shown in the right:

```python
def gun(x, depth):
    if ______________:
        print(x)
        gun(____________, ____________)
```

```python
>>> gun(6, 4)
3
4
5
6
```

2. Gold

Try to run the solution of the gold coin homework problem (home1.py) with a large argument, such as 1000000. Probably the computer aborts the program with an error message. Explain why this happens?

Rewrite the function `number_of_coins` in such a way that it doesn't use recursion.

Run both the recursive version and nonrecursive version of the function with varying values of the argument. Which of these functions runs faster and approximately how many times?

In some situations recursion can be very beneficial, in other cases not so much. Give one solid example of a situation where using recursion to implement a function is useful. Justify your answer.

Give an example of a situation where implementing a function using recursion makes things very much worse, compared to nonrecursive implementation.
3. Extravert square

Write a recursive function that uses Tkinter (or other GUI library) to draw a fractal of a square that has three smaller similar squares at three corners which have three smaller squares at three corners and so on, until the depth given to the function as an argument.

The following figure shows fractals that are created at depths 1, 2, 3, and 4, respectively:

![Fractals](image)

*Hint:* additional info (for example, starting point, direction, etc.) can be given to the function in the form of additional parameters. Think what is the base case, where the recursion should start.

4. Flight plan

Write a recursive function `flight_plan` which has three parameters: database of flights, city of departure and city of arrival, and returns a list of cities that lead from the city of departure to the city of arrival. If flying is not possible, then the function returns an empty list.

The database of flights is given as a dictionary, where keys are cities and the values associated with keys are sets of cities where it is possible to fly directly from that city.

```python
>>> flights = {'Berlin': {'London'}, 'Tallinn': {'Berlin'}}
>>> flight_plan(flights, 'Tallinn', 'Berlin')
['Tallinn', 'Berlin']
```

```python
>>> flights = {'Berlin': {'Tallinn'}, 'Tallinn': {'Berlin'}}
>>> flight_plan(flights, 'Tallinn', 'Nice')
[]
```

```python
>>> flight_plan(flights, 'Tallinn', 'Nice')
['Tallinn', 'Berlin', 'Paris', 'Nice']
```

*Hints.*

- What is the base case of recursion?
- The route from starting city to destination city must go through a city that has a direct link with the starting city.
- To avoid cycles, cities that already belong to the set of visited cities, should be skipped.
5. Maze

A maze is represented by a multi-line string, where # denotes the wall and space denotes blank space. For example,

```python
maze = '''
#############
#   ## #  # #
### ## ## # #
###    ## # #
#   ##### # #
###         #
#############
'''
```

Write a recursive function that takes a maze and coordinates of the starting point and destination point as its arguments, and returns a list of steps that lead from the starting point to the destination point in the maze.

**Question:** what situation is the base case?

**Hint:** take one step in the direction not explored yet and solve the problem again starting from that point. To avoid infinite repetition, the program should remember the set of positions that have already been explored.

**Additional task:** try to make the Pykkar (see worksheet of Week 4) to find the way out of the maze.