National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2024/2025

Mission 2
Cyclic Runes and Maze

Release date: 2nd September 2024
Due: 15th September 2024, 23:59

General Restrictions

- No importing of additional packages (e.g. `math`). Necessary packages will be stipulated in the template file.
- Do not use any compound data structures, such as `tuple`, `list`, `dict`, `set`, etc.

Required Files

- mission02-template.py
- runes.py
- graphics.py
- PyGif.py

Background

After passing through the two doors in your previous mission, you arrive in a large spacious hall with no visible exits except for a large hole in the ceiling, from which a bright beam of light illuminates the entire hall. In one corner, you find two raised terminals with an interface similar to the empty spaces you found on the doors. You can barely see the first letters on their interfaces - R and I.

This mission has **three** tasks.

RESTRICTIONS: In the template file, all functions are defined as `some_func(params)`. You should replace `params` with the appropriate number and order of parameters in accordance to the given examples.

Usage of default and optional arguments is **not** allowed. For all assignments, tests, and exams, functions should be written with the same number of parameters as the examples given for that function, unless otherwise stated.

Task 1: Dual Fractals (8 marks)

This question is a simple modification of `fractal` encountered in Recitation 2. Once again, there are two subtasks, one for a recursive implementation and one for an iterative implementation.

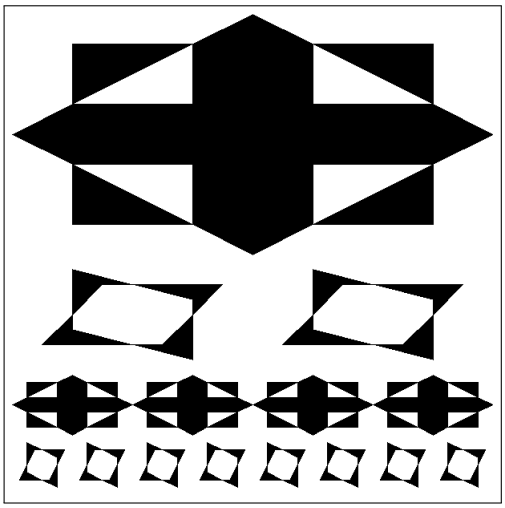
RESTRICTIONS: Your solution to subtask 1a must be **recursive** (i.e. `dual_fractal` must be recursive in nature), and your solution to subtask 1b must be **iterative**. Failure to adhere to these restrictions will result in 0 marks awarded. If you submit a working iterative solution for subtask 1a and a working recursive solution for subtask 1b, **you will receive 0 marks**.

Task 1a: Recursive Dual Fractal (5 marks)

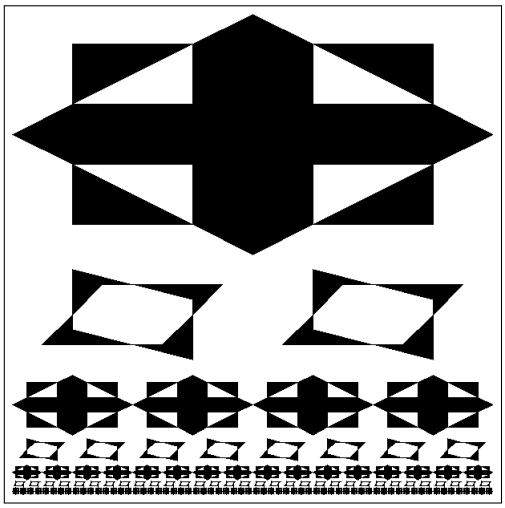
Write a **recursive** function `dual_fractal` that takes as input two `runes` and a integer $n \geq 1$ and returns a `rune` corresponding to the sample execution.

Observe the following sample execution of `dual_fractal`.

```
>>> df1 = dual_fractal(make_cross(rcross_bb), make_cross(nova_bb), 4)
>>> df2 = dual_fractal(make_cross(rcross_bb), make_cross(nova_bb), 7)
```



(a) `show(df1)`



(b) `show(df2)`

Notice that the first rune is used for every odd layer in the fractal starting from the largest, topmost rune, whereas the second rune is used for every even layer in the fractal. Just like in the recitation, for both cases shown here, the topmost rune occupies exactly half of the height of the full rune.

Task 1b: Iterative Dual Fractal (3 marks)

Write a **iterative** function `dual_fractal_iter` that takes as input two `runes` and a integer $n \geq 1$ and returns a `rune` corresponding to the sample execution.

`dual_fractal_iter` has identical functionality as `dual_fractal`, just implemented iteratively.

Task 2: Palindrome (7 marks)

After successfully completing the rigorous training, your prowess has garnered the attention of none other than Pharaoh Tyro, the esteemed ruler of Egypt. Recognizing your exceptional skills and strategic acumen, Pharaoh Tyro entrusts you with further critical tasks that demands your unique abilities.

A **palindromic number** is a number that is the same when read from front to back and back to front. For example, 12321 is a palindromic number, but 110 is not. Interestingly, every positive number can be expressed as a sum of three palindromic numbers (e.g. 877 = 191 + 232 + 454).

Task 2a: Using Reverse (3 marks)

By definition, the reverse of a palindrome is the same as the original palindrome. Therefore, we can check if a number is palindromic or not by checking if the reverse of a number is same as the original number.

Write the functions `reverse` and `is_palindrome_using_reverse`.

- `reverse` takes as input a number and returns another number, formed from reversing the digits of the input.
- `is_palindrome_using_reverse` takes as input a number and returns `True` if the number is palindromic, and `False` otherwise.

The input number is always a **non-negative** `int`.

RESTRICTIONS: `reverse` must be implemented **iteratively** and only make use of arithmetic operations `+, -, *, /, //, %, **` that `int` supports. Conversion to other data types (e.g. `str`) is not allowed. `is_palindrome_using_reverse` must make use of `reverse`.

Sample execution:

```
>>> reverse(12321)
12321
>>> reverse(110)
11
>>> reverse(0)
0
>>> is_palindrome_using_reverse(12321)
True
>>> is_palindrome_using_reverse(110)
False
>>> is_palindrome_using_reverse(0)
True
```

Task 2b: Using Symmetry (4 marks)

We would like to also check if a string is also a palindrome. However, instead of constructing the reverse of the string, we take a different approach here.

Consider a somewhat long string, such as `"today is a really long day"`. We can almost instantly tell that it is not a palindrome, without generating the reverse of this string. This is because it is not *symmetrical*. By comparing the first (`'t'`) and last (`'y'`) characters, we see that they are different, and can therefore immediately say that the string is not a palindrome.

Symmetry refers to the fact that the i-th character from the front is the same as the i-th character from the back. Consider the string `"saippuakivikauppias"`, where the first and last characters are identical, the second and second-to-last characters are also identical, and so on, we eventually conclude that this string is a palindrome.

Write an **iterative** function `is_palindrome_using_symmetry` which takes as input a string and returns `True` if the string is a palindrome and `False` otherwise. Empty strings are the same, no matter which direction they are read from.

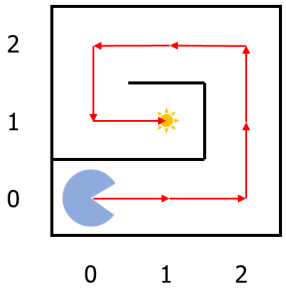
RESTRICTIONS: Your solution must be based on the process described above, using the symmetry property. If you generate or use the reverse of the input string, or solve the problem based on other methods, **you will receive 0 marks**.

Sample execution:

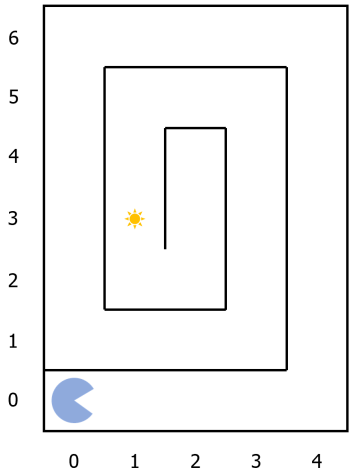
```
>>> is_palindrome_using_symmetry("today is a really long day")
    False
>>> is_palindrome_using_symmetry("saippuakivikauppias")
    True
>>> is_palindrome_using_symmetry("2002") # string not int
    True
>>> is_palindrome_using_symmetry("A man, a plan, a canal: Panama")
    False
>>> is_palindrome_using_symmetry("amanaplanacanalpanama")
    True
```

Task 3: The Spiral Maze (5 marks)

CircleMan is at the bottom left corner of a spiral maze, (coordinates $(0, 0)$), and wants to get to a powerup at some coordinate (x, y). The maze has width W and height H. Consider the following two mazes, which illustrate how the spiral maze is constructed.



(a) $W = 3, H = 3$



(b) $W = 5, H = 7$

CircleMan moves in steps of 1 unit length. In each step, he can move in one direction: up, down, left or right. Since there is only one path to the powerup in this "maze", CircleMan wonders how many steps it would take for him to reach the powerup.

Write an **iterative** function `num_of_steps` that takes as input in 4 arguments, the x and y coordinates of powerup (`x`, `y`), the width and height of the maze (`W`, `H`). By simulating the movement of CircleMan within the maze, `num_of_steps` returns the number of steps needed to reach the powerup from the origin.

You are given that $0 \leq x < W$ and $0 \leq y < H$, i.e. there are no invalid inputs.

RESTRICTIONS: You must use iteration to solve this problem. If a closed-form solution is given, **you will receive 0 marks**.

Sample execution:

```
>>> num_of_steps(1, 1, 3, 3) # (a)
    8
>>> num_of_steps(0, 0, 3, 3)
    0
>>> num_of_steps(1, 3, 5, 7) # (b)
    30
>>> num_of_steps(1, 1, 3, 2)
    4
```