Exercises

Set 2

DM857 Introduction to Programming
DS830 Introduction to Programming

1 Conditionals

1. Define a function `sign(n: float) -> int` that returns the sign of the number n using the following algorithm:

$$sign(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ -1 & \text{if } n < 0 \end{cases}$$

2. Define a function `convert_to_meters(length: float, unit: str) -> float` that converts `length` given in `unit` to meters using the following table (the unit is expressed using strings in its extended or abbreviated form).

	unit		meters
1	`'inch'`, `'in'`	=	0.0254
1	`'hand'`, `'h'`	=	0.1016
1	`'foot'`, `'ft'`	=	0.3048
1	`'yard'`, `'yd'`	=	0.9144

For instance `convert_to_meters(30,'in')` must return `0.762`.

3. Define a function `convert(length: float, from_unit: str, to_unit: str) -> float` that converts `length` given in `from_unit` units to `to_unit` units where units are expressed as `'m'`, `'in'`, `'h'`, `'ft'`, and `'yd'`.

(`from` and `to` could better names, however `from` is a keyword i.e., and, like `def` and `return` cannot be used as an identifier/name).

4. Define a function `classify_triangle` that takes the length of three segments and returns

- `0` if the three segments do not form a triangle;
- `1` if the three segments form a scalene triangle;
- `2` if the three segments form an isosceles triangle;
- `3` if the three segments form an equilateral triangle.

(The triangle inequality for testing if segments with lengths a,b,c form a triangle can be succinctly expressed as $\max(a, b, c) < a + b + c - \max(a, b, c)$.)

2 Recursion on numbers

Implement the functions and don't forget to use type hinting, docstrings and doctests.

1. Define a recursive function `count_down(n: int)` that prints all natural numbers from `n` down to `0` (included), one per line.

```
>>> count_down(2)
2
1
0
>>> count_down(-5)
>>> count_down(0)
0
```

2. Define a recursive function `count_up(n: int)` that prints all natural numbers up to `n` (included), one per line.

```
>>> count_up(2)
0
1
2
>>> count_up(-5)
>>> count_up(0)
0
```

3. Define a recursive function `count_down_up(n: int)` that prints all natural numbers from `n` to 0 and back to `n` (included), one per line.

```
>>> count_down_up(2)
2
1
0
1
2
>>> count_up(-5)
>>> count_up(0)
0
```

4. Define a recursive function `sum_up_to(n: int) -> int` that returns the sum of all natural numbers smaller than or equal to `n`.

5. Define a recursive function `power(b: float, n: int) -> float` that returns b^n where n is a natural number.

6. Define a recursive function `int_log(x: float, b: float) -> int` that returns the integer logarithm in base b of x (both positive) i.e., the natural number n such that $b^n \leq x < b^{(n+1)}$. (Hint: use repeated division)

7. Define a recursive function `factorial(n: int) -> int` that returns $n!$, the factorial of n ($n! = 1 \cdot 2 \cdot \ldots \cdot n$) using the algorithm:

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

8. Define a function `double_factorial(n: int) -> int` that returns $n!!$ ($n!! = 1 \cdot 3 \cdot 5 \cdot \ldots \cdot n$ if n is odd and $n!! = 2 \cdot 4 \cdot 6 \cdot \ldots \cdot n$ if n is even).

9. Define a recursive function `fib(n: int) -> int` computes f_n, the $(n+1)$-th number of the Fibonacci series (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...) using the algorithm:

$$f_n = \begin{cases} n & \text{if } n \leq 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

10. Define a recursive function `gcd(m: int, n: int) -> int` that returns the greatest common divisor of `m` and `n` using Euclides' algorithm:

$$gcd(m,n) = \begin{cases} m & \text{if } m = n \\ gcd(m, n-m) & \text{if } m < n \\ gcd(m-n, n) & \text{if } m > n \end{cases}$$

11. Define a recursive function `lcm(m: int, n: int) -> int` that returns the least common multiple of `m` and `n`.

12. Define a recursive function `sum_between(m: int, n: int) -> int)` that returns the sum of all integer numbers greater than `m` and smaller than `n`.

13. Define a recursive function `sum_odd_up_to(n: int) -> int` that returns the sum of all odd natural numbers smaller than or equal to `n`.

14. Define a recursive function `sum_even_up_to(n: int) -> int` that returns the sum of all even natural numbers smaller than or equal to `n`.

15. Define a recursive function `sum_even_between(m: int, n: int) -> int` that returns the sum of all integer even numbers greater than `m` and smaller than `n`.

16. Define a recursive function `sum_odds_between(m: int, n: int) -> int` that returns the sum of all integer odd numbers greater than `m` and smaller than `n`.

17. Define functions `f(n: int) -> int` and `f(n: int) -> int` that compute the n-th element of the Hofstadter Female-Male sequence using the algorithm below:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n - m(f(n-1)) & \text{otherwise} \end{cases} \qquad m(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - f(m(n-1)) & \text{otherwise} \end{cases}$$

    This type of algorithm is called *mutually recursive* since f and m call each other. The first 10 numbers for the f sequence are 1, 1, 2, 2, 3, 3, 4, 5, 5, 6; and for the m sequence are 0, 0, 1, 2, 2, 3, 4, 4, 5, 6.

18. Define a function `is_prime(n: int) -> bool` that given a positive integer `n` returns `True` if `n` is prime and `False` otherwise. (Hint: use an auxiliary function).