PRACTICE SHEET

What you will learn:

Variable-Length Arguments (*args), Functions, Custom Recursion Logic, Scope(Local vs global variable behavior), Combinatorics & Recursive Path Counting, Tower of Hanoi

[EASY] Q1) What does the *args parameter allow in a function?
a) Passing unlimited **named** arguments
b) Passing unlimited **positional** arguments
c) Passing only **two** arguments
d) Returning multiple values

[EASY] Q2) def twisted_fact(n):

```
    if n == 0:
        return 1
    if n % 2 == 0:
        return n * twisted_fact(n-1)
    else:
        return twisted_fact(n-1)
print(twisted_fact(5))
```

Question: What will the output be and why?

(Think: it skips odd numbers in multiplication, but does recursion continue?)

[EASY] Q3) x = 5

```
def outer():
    x = 10
    def inner():
        global x
```

```
        x += 1
        print("Inner:", x)
    inner()
    print("Outer:", x)

outer()
print("Global:", x)
```

Question: Predict the output. Explain how global affects the variables here.

[MEDIUM] Q4)

```
def count_paths(x, y):
    if x == 0 or y == 0:
        return 1
    return count_paths(x-1, y) + count_paths(x, y-1)
print(count_paths(2, 2))
```

Question: How many paths are there from (0,0) to (2,2) only by moving right or down?

[EASY] Q5)

```
def nested(n):
    if n > 100:
        return n - 10
    return nested(nested(n+11))
print(nested(95))
```

Question: What is the output of this classic McCarthy 91 function? Explain how recursion works twice in each call.

[EASY] Q6) Write a function named getCostOfCoffee() that has a parameters named numberOfCoffees and pricePerCoffee. Given this information, the function returns the total cost of the coffee order. This is not a simple multiplication of cost and quantity, however, because the coffee shop has an offer where you get one free coffee for every eight coffees you buy. For example, buying eight coffees for $2.50 each costs $20 (or 8 × 2.5). But buying nine coffees also costs $20, since the first eight makes the ninth coffee free. Buying ten coffees calculates as follows: $20 for the first eight coffees, a free ninth coffee, and $2.50 for the tenth coffee for a total of $22.50.

[HARD] Q7) Imagine this:
You have **3 golden disks** stacked on rod **A**, each smaller than the one below it. Your goal? Move the entire tower to rod **C**—but there's a catch:

1. **Only one disk** can be moved at a time.
2. **No disk** may be placed on top of a smaller one.

The Question: If you follow the recursive steps below, how many moves does it take to solve the puzzle for 3 disks(n = 3)?

```
def hanoi(n, source, helper, target):
    if n == 0:
        return
    hanoi(n-1, source, target, helper)
    print(f"Move disk {n} from {source} to {target}")
    hanoi(n-1, helper, source, target)
hanoi(3, 'A', 'B', 'C')
```

Question: How many total moves are made?

[EASY] Q8) You are given this code:

```
x = 5
def outer():
    x = 10
```

```
    def inner():
        print("Inner x:", x)
    inner()
outer()
```

What will be printed?
Modify the code so that inner() prints the **global x** instead of the local one.

[MEDIUM] Q9) Write a recursive function sort_digits(n) that returns a number whose digits are **sorted in ascending order**.

Example :

sort_digits(4312) → 1234

sort_digits(989) → 899

Hint: Convert to string at each level and reassemble recursively.

[HARD] Q10) Write an isValidDate() function with parameters year, month, and day. The function should return True if the integers provided for these parameters represent a valid date. Otherwise, the function returns False. Months are represented by the integers 1 (for January) to 12 (for December) and days are represented by integers 1 up to 28, 29, 30, or 31 depending on the month and year. Additionally, make isLeapYear() function, as February 29th is a valid date on leap years. September, April, June, and November have 30 days. The rest have 31, except February which has 28 days. On leap years, February has 29 days.

Example :

isValidDate(1999, 12, 31) == True

isValidDate(2000, 2, 29) == True

isValidDate(2001, 2, 29) == False

isValidDate(2029, 13, 1) == False

isValidDate(1000000, 1, 1) == True

isValidDate(2015, 4, 31) == False

isValidDate(1970, 5, 99) == False

isValidDate(1981, 0, 3) == False

isValidDate(1666, 4, 0) == False

[HARD] Q11) In a Smart City, there is a central **water tank** on a hill that supplies water to N buildings via a **cascade pipe system**.

Each building receives water **only if the previous one gets enough**.
Water flow decreases by **10%** with every building due to leakage and height difference.

There are 4 constraints:

1. **Initial water** given to the first building is W liters.
2. Each building needs **at least 20 liters** to function.
3. If any building gets less than 20 liters, **the remaining buildings receive nothing**.
4. Water flow is calculated using:
   flow_to_next = 90% of previous building's water

Your Tasks:

1. Write a recursive function supply_water(water, buildings=0) that returns:

- The number of buildings that **successfully receive** water.

2. Also create a function city_report(city_name, *args, required=20):

- Where args represent initial water levels in **different sectors** of the city.
- It must print a summary:
  "Sector {i} of {city_name}: X buildings received water" for each sector.

Example:

supply_water(100) → 7

city_report("Raipur", 100, 80, 200)

Output:

Sector 1 of Raipur: 7 buildings received water

Sector 2 of Raipur: 6 buildings received water

Sector 3 of Raipur: 9 buildings received water

[HARD] Q12) Write a Python function that prints a *fake CAPTCHA* using random letters and numbers. Do not use any images or complex libraries. Just use strings and randomness.

[MEDIUM] Q13) Create a custom function that behaves like the bool() function without using it.

[HARD] Q14) You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Input: n = 3

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

Follow Up: Is your recursion code optimized? Will it run for large inputs?

Think: Dynamic Programming (DP)

Imagine solving the same puzzle repeatedly - it's exhausting! DP avoids this by:

1. **Breaking problems** into smaller subproblems.
2. **Storing solutions** to avoid recomputing them.

DP Trick - Reuse solutions : ways(n) = ways(n-1) + ways(n-2) (Fibonacci!).

[EASY] Q15) You are playing the following Nim Game with your friend:

- Initially, there is a heap of stones on the table.
- You and your friend will alternate taking turns, and **you go first**.
- On each turn, the person whose turn it is will remove 1 to 3 stones from the heap.

- The one who removes the last stone is the winner.

Given n, the number of stones in the heap, return true if you can win the game assuming both you and your friend play optimally, otherwise return false.

Input: n = 4

Output: false

Explanation: These are the possible outcomes:

1. You remove 1 stone. Your friend removes 3 stones, including the last stone. Your friend wins.

2. You remove 2 stones. Your friend removes 2 stones, including the last stone. Your friend wins.

3. You remove 3 stones. Your friend removes the last stone. Your friend wins.

In all outcomes, your friend wins.