PRACTICE SHEET

What you will learn:

Lists, Default Mutable Arguments, Tuples & Mutability Inside Containers, Generator Expressions & Tuple Comprehension, Sliding Window, Statistics, Dynamic Programming.

[EASY] Q1) What is the output of this code:

```
def strange(lst=[]):
    lst.append(len(lst))
    print(lst)
strange()
strange()
strange()
```

[EASY] Q2) What is the output of this code?

```
x = ([1, 2], [3, 4])
x[0].append(5)
print(x)
```

A. ([1, 2], [3, 4])
B. ([1, 2, 5], [3, 4])
C. Error due to tuple
D. ([5], [3, 4])

[EASY] Q3) Python supports list comprehension: [x**2 for x in range(5)].
Can you do the same with tuples? Try:

```
t = (x**2 for x in range(5))
print(type(t))
```

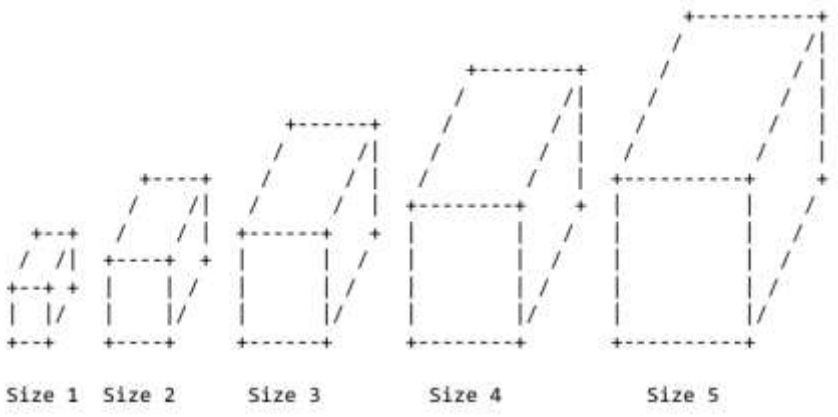
[MEDIUM] Q4) Signal Degradation in Communication Towers (Recursive)
Domain: Telecom / Electronics
A signal starts at Tower 1 with a strength of 100 units.
It passes through a total of n towers, including the starting one.
At each tower, the signal degrades by a factor r (i.e., it's multiplied by r, where r < 1).
You are to write a recursive function that:

- Calculates the final signal strength after passing through all n towers.
- Counts how many towers received a signal **less than 40%** of the original (i.e., less than 40 units).

Return a **tuple** of two values:

1. Final signal strength (rounded to 2 decimal places).
2. The number of towers that received less than 40 units.

[MEDIUM] Q5) In this question, draw a 3D ASCII art by programmatically generating boxes at any given size. Write a drawBox() function with a size parameter. The size parameter contains an integer for the width, length, and height of the box. The horizontal lines are drawn with - dash characters, the vertical lines with | pipe characters, and the diagonal lines with / forward slash characters. The corners of the box are drawn with + plus signs.



[MEDIUM] Q6) Find the largest subarray (contiguous) where all elements are unique.

Input: [5, 1, 3, 5, 2, 3, 4, 1]

Output: 5 ([5, 2, 3, 4, 1])

[MEDIUM] Q7) Create a function smart_sum(*args) that takes any number of inputs and returns their sum. Twist: If an argument is a list or tuple, sum its elements too.

Example:

smart_sum(1, 2, [3, 4], (5, 6)) → 21

[EASY] Q8) Write a recursive function max_depth(lst) that returns the maximum depth of nested lists.

Example:

max_depth([1, [2, [3, [4]]]]) → 4

max_depth([1, 2, 3]) → 1

Constraints: You can't use isinstance() inside loops. Must solve recursively.

[EASY] Q9) Write a median() function that has a numbers parameter. This function returns the statistical median of the numbers list. The median of an odd-length list is the number in the middlemost number when the list is in sorted order. If the list has an even length, the median is the average of the two middlemost numbers when the list is in sorted order. Feel free to use Python's built-in sort() method to sort the numbers list. Passing an empty list to median() should cause it to return None.

[MEDIUM] Q10) Take two 10-digit hexadecimal numbers in two lists. Each digit (0 or 1) of first number will be stored as one element of first list and similarly for second one. Multiply these two numbers to get the answer in a 20 element list.

Example input arrays (10-digit hexadecimal numbers)

lst1 = ['1', 'A', '3', 'F', '0', '4', '2', 'C', '9', 'B']

lst2 = ['0', 'F', '2', '3', 'D', '1', 'A', 'E', 'B', '7']

Output = ['0', '1', '8', 'D', '5', 'D', '5', 'B', '8', 'B', '2', 'E', '7', 'D', 'D', '8', '3', 'C', 'C', 'D']

[MEDIUM] Q11) Rotate a list to the right by k steps without using extra list. Use no in-built functions.

Input: [1, 2, 3, 4, 5, 6], k = 2

Output: [5, 6, 1, 2, 3, 4]

Hint: Use reverse function.

[EASY] Q12) Given a square matrix of odd size n x n, find the sum of both diagonals **without repeating the center**.

Function:

def diagonal_sum(matrix: List[List[int]]) -> int:

Input:

[[1,2,3],

[4,5,6],

[7,8,9]]

Output: 25 (1+5+9+3+7 = 25, center 5 only once)

[EASY] Q13) Clocks have an unusual counting system compared to the normal decimal number system we're familiar with. Instead of beginning at 0 and going to 1, 2, and so on forever, clocks start at 12 and go on to 1, 2, and so on up to 11. Then it loops back to 12 again. (Clocks are quite odd if you think about it: 12 am comes before 11 am and 12 pm comes before 11 pm.) This is a bit more complicated than simply writing a program that counts upward.

Write a program that displays the time for every 15 minute interval from 12:00 am to 11:45 pm. Your solution should produce the following output:

12:00 am

12:15 am

12:30 am

12:45 am

1:00 am

1:15 am

--cut—

11:30 pm

11:45 pm

There are 96 lines in the full output.

Prerequisite concepts: for loops, lists, nested loops, string concatenation

[HARD] Q14) Write a **recursive function** product_except_self(lst) that returns a new list where each element at index i is the product of **all elements in the input list except the one at index i**, **without using division**.

Your solution must use recursion.

Examples:

Input	**Output**	**Explanation**
`[1, 2, 3, 4]`	`[24, 12, 8, 6]`	24 = 2×3×4, 12 = 1×3×4, etc.
`[0, 1, 2, 3]`	`[6, 0, 0, 0]`	Only one zero → all others become zero
`[1, 0, 3, 0]`	`[0, 0, 0, 0]`	More than one zero → all become zero
`[5]`	`[1]`	No other elements → default is 1

[HARD] Q15) You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer list nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Input: nums = [2,7,9,3,1]

Output: 12

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

Input: nums = [2,1,1,9]

Output: 11

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9).

Total amount you can rob = 2 + 9 = 11.

Follow up:

When we solve a problem using recursion, the function calls itself for smaller subproblems.
Each call is placed on a call stack, which remembers where to return once the inner call finishes.

Recursive solutions often **recompute the same values** again and again.
This leads to:

- High time complexity
- Stack overflow in large inputs

Dynamic Programming (DP) is a technique that:

- **Stores** results of subproblems (also called **memoization**)
- **Avoids redundant calculations** by checking if we've already solved it

So instead of re-solving the same input, we **look it up**!

Now each subproblem is solved **only once**, saving time and avoiding deep recursion.

So try the same question using Dynamic Programming.