Weekly Lab

Recursion

In this lab session, we will explore recursion technique.

Instructions

Recursion is a common technique where a difficult problem is broken down into smaller instances of the same problem. By applying the same procedure repeatedly, the problem becomes simpler until reaching a base case that can be solved directly.

1. Basic Concepts

Definition: A function that is defined in terms of itself is called a **recursive function**.
A recursive function always has:

- **Base Case**: Defines the termination condition.
- **Recursive Case**: The function is redefined in a smaller scope.

For example, the Fibonacci sequence is defined as:

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ f(n-1) + f(n-2), & \text{if } n \geq 2 \end{cases} \tag{1}$$

Therefore, the Fibonacci sequence is as follows: 0, 1, 1, 2, 3, 5, 8, etc.

The following Python implementation demonstrates how recursion can be used to compute the Fibonacci sequence following the mathematical definition above:

```python
def fibo(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibo(n-1) + fibo(n-2)  # Recursive call
```

2. Types of Recursion

A recursive function can be classified in many different ways based on the criteria of classification. Below are common types of recursion:

Recursion Call Type

- Direct Recursion: A function calls itself within its body.

```
def func():
    func()  # Calls itself
```

- Indirect Recursion: A function calls another function, and that one calls back the original one.

```
def A():
    B()

def B():
    A()
```

Tail and Non-Tail Recursion

- Tail Recursion: The recursive call is the last operation in the function.

```
def tail_recursion(n):
    if n == 0:
        return
    print(n, " ")
    tail_recursion(n - 1)
```

- Non-Tail Recursion: There are computations or calls after the recursive call.

```
def non_tail_recursion(n):
    if n == 0:
        return
    non_tail_recursion(n - 1)
    print(n, " ")
```

Additionally, there are other classification methods, such as classification based on the way the problem size is reduced (linear recursion, binary recursion, multiple recursion), or based on the data structures used (recursion on linked lists, recursion on trees, recursion on graphs), etc.

3. Removing Recursion (Iteration)

A naive recursive implementation is inefficient due to redundant calculations. The Python code below implements a recursive Fibonacci function with a call counter.

```
call_count = 0  # Counter for recursive calls

def fibo(n):
    global call_count
    call_count += 1
    ...

# Example usage
call_count = 0
result = fibo(10)
print("Total recursive calls:", call_count)
```

For $n = 10$, the total recursive calls is 177 with many redundant calculations. Indeed, the number of recursive calls grows exponentially in an exponential time complexity $O(2^n)$.
The following code computes Fibonacci numbers using an iterative approach (removed recursion).

```
def fibo_iterative(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    calc_count = 0
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
        calc_count += 1
    print("Total Fibonacci calculations:", calc_count)
    return b
```

The iterative approach only requires a loop of $n-1$ iterations, making it significantly more efficient than recursion. The time complexity is $O(n)$ and does not involve redundant calculations.

n	Recursive Calls $O(2^n)$	Iterative Loops $O(n)$
10	177	9
20	21,891	19
30	2,692,537	29

Table 1: Comparison of recursive and iterative Fibonacci computations.

4. Memoization

A naive recursive implementation is inefficient due to redundant calculations, each call recomputes the same values multiple times. Memoization is a technique that stores previously computed values to avoid redundant computations.

The following Python code demonstrates the Fibonacci function using memoization techinique with a dictionary to store intermediate results.

```
memo = {}

def fibo(n):
    if n in memo:
        return memo[n]
    if n == 0:
        return 0
    if n == 1:
        return 1
    memo[n] = fibo(n-1) + fibo(n-2)
    return memo[n]

print("Fibonacci(5):", fibo(5))
```

With memoization technique, each Fibonacci number is computed only once and stored for reuse. The time complexity is reduced to $O(n)$ compared to the exponential complexity of naive recursion.

n	Recursive Calls $O(2^n)$	Memoized Calls $O(n)$
5	15	5
10	177	10
20	21,891	20
30	2,692,537	30

Table 2: Comparison of naive recursion and memoization in Fibonacci computation.

Memoization significantly improves performance while retaining the simplicity of recursion.

Additionally, there is an existing **decorator**, which is `lru_cache`, solves all those problems easily. It is in the `functools` module, and it only takes one line of code to set it up:

```
from functools import lru_cache

@lru_cache()
def fibonacci(n):
    ...
```

Exercises

Exercise 1. Factorial

Write a program to compute the factorial of a given non-negative integer n. The factorial of n is defined as:

$$n! = n \times (n-1) \times \cdots \times 1,$$

where $0! = 1$.

Input:

- A non-negative integer n ($0 \leq n \leq 20$).

Output:

- The factorial of n.

Example:

Input	Output
5	120
0	1

Exercise 2. Sum of Digits

Write a program to compute the sum of the digits of a given positive integer.

Input:

- A positive integer n ($1 \leq n \leq 10^9$).

Output:

- The sum of the digits of n.

Example:

Input	Output
1234	10
9876	30

Exercise 3. Greatest Common Divisor

Write a program to compute the greatest common divisor (GCD) of two integers a and b.

Input:

- Two positive integers a and b ($1 \leq a, b \leq 10^6$).

Output:

- The greatest common divisor of a and b.

Example:

Input	Output
`24 18`	`6`
`100 75`	`25`

Exercise 4. Check Strictly Increasing List

Write a program to determine whether a given list of integers is strictly increasing.

Input:

- An integer n ($1 \leq n \leq 100$), the number of elements in the list.
- A sequence of n integers ($-10^6 \leq a_i \leq 10^6$).

Output:

- `Yes` if the list is strictly increasing.
- `No` otherwise.

Example:

Input	Output
`5` `1 3 5 7 9`	`Yes`
`4` `2 2 3 4`	`No`

Exercise 5. Reverse a String

Write a program to reverse a given string.

Input:

- A string consisting of lowercase or uppercase characters.

Output:

- The reversed string.

Example:

Input	Output
`hello`	`olleh`
`world`	`dlrow`

Exercise 6. Check Palindrome String

Write a program to check whether a given string is a palindrome. A palindrome is a string that reads the same backward as forward.

Input:

- A string consisting of lowercase characters only.

Output:

- `Yes` if the string is a palindrome.
- `No` otherwise.

Example:

Input	Output
`madam`	`Yes`
`hello`	`No`

Exercise 7. Generate Permutations

Write a program to generate all permutations of a given set of distinct integers.

Input:

- An integer n ($1 \leq n \leq 10$) representing the number of elements in the set.
- A sequence of n distinct integers.

Output:

- A list of all possible permutations of the given set, each on a new line.

Example:

Input	Output
3 1 2 3	1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

Exercise 8. Generate Subsets

Write a program to generate all subsets of a given set of distinct integers.

Input:

- An integer n ($1 \leq n \leq 10$) representing the number of elements in the set.
- A sequence of n distinct integers.

Output:

- A list of all possible subsets, each on a new line.
- The subsets should be sorted in lexicographical order.

Example:

Input	Output
3 1 2 3	{} {1} {2} {3} {1,2} {1,3} {2,3} {1,2,3}

Exercise 9. Generate Binary Strings

Write a program to generate all binary strings of length n.

Input:

- An integer n ($1 \leq n \leq 10$) representing the length of the binary string.

Output:

- A list of all possible binary strings of length n, each on a new line.

Example:

Input	Output
3	000 001 010 011 100 101 110 111

Exercise 10. N-Queens Problem

Write a program to solve the N-Queens problem, which places n queens on an $n \times n$ chessboard such that no two queens attack each other.

Input:

- An integer n ($1 \leq n \leq 10$) representing the size of the chessboard.

Output:

- A list of all possible valid solutions.
- Each solution should be represented as an n-length list, where the i-th number represents the column position of the queen in row i.

Example:

Input	Output
`4`	`[2, 4, 1, 3]` `[3, 1, 4, 2]`

Regulations

Please follow these guidelines:

- You may use any Python IDE.
- After completing assignment, check your submission before and after uploading to Moodle.
- Do not use the following modules: `numpy`, `pandas`, `collections`, `heapq`, and `deque`.
- You may use `list`, `tuple`, and `set` but no external libraries.

Your submission must be contributed in a compressed file, named in the format `StudentID.zip`, with the following structure:

```
StudentID
├── Exercise_1.py
└── ...
```

The end.