**Week 3 - Appendix 1: Analyzing Algorithmic Efficiency**

### **User-Generated Prompt for AI (Example)**

"I need a Python script that:\*

* **Generates and compares the efficiency of iterative and recursive approaches to solving a problem.**
* **Implements both iterative and recursive functions for calculating the Fibonacci sequence.**
* **Measures execution time for both implementations and compares their performance using Python’s time module.**
* **Uses memoization to optimize the recursive approach and improve efficiency.** Can you generate an efficient Python script for these tasks?"\*

### **AI-Generated Python Code for Algorithmic Efficiency Analysis**

import time

from functools import lru\_cache

# Recursive Fibonacci with memoization

@lru\_cache(maxsize=None)

def fibonacci\_recursive(n):

if n == 0:

return 0

elif n == 1:

return 1

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

# Iterative Fibonacci

def fibonacci\_iterative(n):

a, b = 0, 1

for \_ in range(n):

a, b = b, a + b

return a

# Function to measure execution time

def measure\_execution\_time(func, n):

start\_time = time.time()

result = func(n)

end\_time = time.time()

return result, end\_time - start\_time

# Test and compare execution times

n = 30 # Adjust this for testing larger/smaller values

recursive\_result, recursive\_time = measure\_execution\_time(fibonacci\_recursive, n)

iterative\_result, iterative\_time = measure\_execution\_time(fibonacci\_iterative, n)

print(f"Recursive Fibonacci({n}): {recursive\_result}, Time: {recursive\_time:.5f} sec")

print(f"Iterative Fibonacci({n}): {iterative\_result}, Time: {iterative\_time:.5f} sec")

### **Discussion Questions:**

1. How does memoization improve recursive function performance?
2. What is the time complexity of the iterative vs. recursive approach?
3. At what input size does the naive recursive implementation become inefficient, and how does memoization mitigate this?

**Week 3 - Appendix 2: Advanced Problem-Solving Challenges**

### **User-Generated Prompt for AI (Example)**

"I need a Python script that:\*

* **Solves a multi-step data processing pipeline using AI-generated code.**
* **Performs data cleaning, transformation, and aggregation on a dataset.**
* **Uses modular functions to break down each step into reusable components.**
* **Optimizes performance using vectorized operations in Pandas.** Can you generate an efficient Python script for these tasks?"\*

### **AI-Generated Python Code for Advanced Problem-Solving Challenge**

import pandas as pd

# Step 1: Load and clean data

def load\_and\_clean\_data(file\_path):

df = pd.read\_csv(file\_path)

df.dropna(inplace=True) # Remove missing values

df.columns = df.columns.str.strip().str.lower() # Standardize column names

return df

# Step 2: Transform data (Example: Creating a new calculated column)

def transform\_data(df):

df['temperature\_f'] = df['temperature\_c'] \* 9/5 + 32 # Convert Celsius to Fahrenheit

return df

# Step 3: Aggregate data (Example: Average temperature by category)

def aggregate\_data(df):

return df.groupby('category')['temperature\_f'].mean().reset\_index()

# Step 4: Save processed data

def save\_data(df, output\_path):

df.to\_csv(output\_path, index=False)

print(f"Processed data saved to {output\_path}")

# Example usage

file\_path = "Environmental\_Dataset\_example.csv"

output\_path = "processed\_environmental\_data.csv"

df = load\_and\_clean\_data(file\_path)

df = transform\_data(df)

df = aggregate\_data(df)

save\_data(df, output\_path)

### **Discussion Questions:**

1. How does breaking the problem into functions improve maintainability?
2. What are the performance benefits of vectorized Pandas operations over loops?
3. How could this pipeline be extended to handle additional transformations?