

DS 2nd exam test, February 10th, 2025

Download the data

1. Consider the file `sciprog-ds-10-02-2025-FIRSTNAME-LASTNAME-ID.zip` and extract it on your desktop.
2. Rename `sciprog-ds-10-02-2025-FIRSTNAME-LASTNAME-ID` folder:

Replace **FIRSTNAME**, **LASTNAME**, and **ID** with your first name, last name and student id number. Failure to comply with these instructions will result in the loss of 1 point on your grade.

like `sciprog-ds-10-02-2025-alessandro-romanel-432432`

From now on, you will be editing the files in that folder.

3. Edit the files following the instructions.
4. At the end of the exam, **compress** the folder in a zip file

`sciprog-ds-10-02-2025-alessandro-romanel-432432.zip`

and submit it. This is what will be evaluated. Please, include in the zip archive all the files required to execute your implementations!

NOTE: You can only use the data structures and packages provided in the exam script files. **Importing other Python packages IS NOT allowed** unless explicitly stated in the exam instructions. Using Python collections or other libraries will impact your final grade. Still, **IT IS ALLOWED** to use **built-in Python operators** as we have done during the practical classes (max, min, len, reversed, list comprehensions, etc).

Exercise 1 [FIRST MODULE]

You are given a dataset containing information about car prices from 2000 to 2023. The file is located in the **data** folder. Below is an example of the first five entries in the file:

	Brand	Model	Year	Engine_Size	Fuel_Type	Transmission	Mileage	Doors	Owner_Count	Price
0	Kia	Rio	2020	4.2	Diesel	Manual	289944	3	5	8501
1	Chevrolet	Malibu	2012	2.0	Hybrid	Automatic	5356	2	3	12092
2	Mercedes	GLA	2020	4.2	Diesel	Automatic	231440	4	2	11171
3	Audi	Q5	2023	2.0	Electric	Manual	160971	2	1	11780
4	Volkswagen	Golf	2003	2.6	Hybrid	Semi-Automatic	286618	3	3	2867

Each row in the dataset includes various details about a car, such as the brand, model, year of manufacture, engine size, fuel type, transmission type, additional specifications, and price.

Exercise 1.1

Load the dataset.

Exercise 1.2

Define a function named `get_most_expensive` that prints the brand, model, and year of the most expensive car in the dataset.

```
def get_most_expensive(...):  
    ...  
    print ...
```

```
> get_most_expensive(df)  
>> Toyota Corolla 2021
```

Exercise 1.3

Define a function named `get_most_expensive_filter` that takes a dataframe as input, along with optional parameters: `max_engine_size`, `year`, and `transmission`.

The function should:

- Filter the dataset based on the given parameters:
 - Include only cars with an engine size smaller than `max_engine_size` (if specified).
 - Include only cars from the specified `year` (if provided).
 - Include only cars with the specified `transmission` type (if given).
- Identify the most expensive car that meets the filtering criteria.
- **Return** the brand and model of that car.

If a parameter (`max_engine_size`, `year`, or `transmission`) is not specified, the function should ignore that filter.

Example usage:

```
get_most_expensive_filter(df, max_engine_size=2.3, year=2021)
```

In this case, the function filters cars with an engine size smaller than 2.3 and from the year 2021, then returns the brand and model of the most expensive car among them.

```
def get_most_expensive_filter(...):
    ...
    return ...

> get_most_expensive_filter(df)
>> ('Toyota', 'Corolla')

> get_most_expensive_filter(df, eng_size=2.3)
>> ('Honda', 'CR-V')

> get_most_expensive_filter(df, eng_size=2.3, year=2002)
>> ('Volkswagen', 'Passat')

> get_most_expensive_filter(df, eng_size=2.3, year=2002, trans="Manual")
>> ('Audi', 'A3')
```

Exercise 1.4

We are interested in analyzing the average price of **Electric** and **Hybrid** cars over the years. Define a function named `get_prices` that takes `fuel_type` as an input and returns a dictionary where each **year** is mapped to the corresponding **average car price** for that year.

The function should calculate the average price of cars for each year based on the specified fuel type. If no fuel type is provided, the default should be **Electric**.

```
def get_prices(...):
    ...
    return dictionary

> get_prices(df, fuel="Electric")
>> {2000: 6410.222222222223,
    2001: 6838.45871559633,
    2002: 6985.532110091744,
    2003: 7529.824175824176,
    2004: 7602.838383838384,
    2005: 8096.38524590164,
    2006: 8720.137614678899,
    ....}
```

Exercise 1.5

As shown in the results, car prices have increased significantly, rising from approximately **6,000 in 2020** to **13,000 in 2023**. However, making a direct comparison without accounting for the general increase in car prices over time would be misleading.

To address this, define a new function named `get_price_correct`. Instead of returning the average price of electric/hybrid cars in each year, this function should compute the **price difference** between the average price of electric/hybrid cars and the overall average price of all cars in that year.

Example Calculation:

Consider the following dataset for the year **2020**:

Brand	Model	Year	Fuel Type	...	Price
B1	m1	2020	Electric	...	20
B2	m2	2020	Electric	...	30
B3	m3	2020	Diesel	...	25
B4	m4	2020	Diesel	...	15

1. **Compute the overall average price for all cars in 2020:**
 $(20+30+25+15)/4=22.5$
2. **Compute the average price of electric cars in 2020:**
 $(20+30)/2=25$
3. **Compute the difference:**
 $25-22.5=2.5$

The function should return a dictionary where the **keys** are the years and the **values** are the computed differences.

For this example, the dictionary would contain: {2020:2.5}

```
def get_prices_correct(...):  
    ...  
    return dictionary
```

```
> get_prices_correct(df, fuel="Electric")  
>> {2000: 1016.48685326548,  
      2001: 934.3946761874631,  
      2002: 1028.7810278406614,  
      2003: 1303.989530154884,  
      2004: 1272.112495513511,  
      2005: 1153.3481229318022,  
      ...}
```

Exercise 1.6

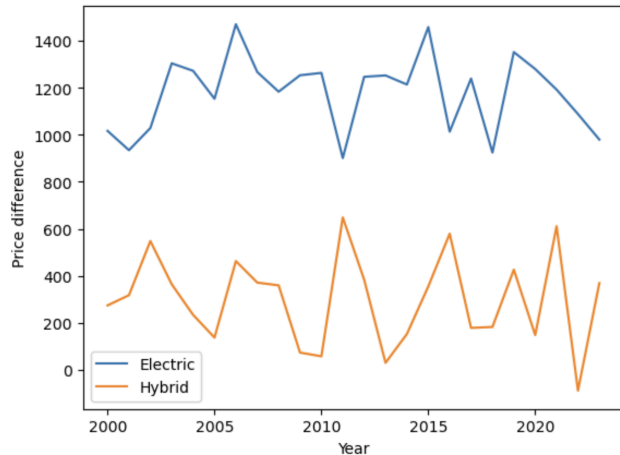
Now that you have two dictionaries—one for **Electric cars** and one for **Hybrid cars**—you need to visualize the results using **Matplotlib**.

Plot Requirements:

- The plot should contain **two lines**:
 - One for **Electric cars**.
 - One for **Hybrid cars**.
- The **x-axis** represents the **years** (labeled "**Years**").
- The **y-axis** represents the **difference** between the average price of all cars and the average price of Electric/Hybrid cars (labeled "**Price difference**").
- A **legend** should be included to distinguish between the two lines.
- The figure should resemble the provided example.
- Save the plot as "**fig.png**".

Alternative Approach:

If you were unable to complete the previous exercise (Exercise 1.5), define two lists of **23 values** each and assume they represent the computed differences for **Electric** and **Hybrid** cars, then proceed with the plot using these lists.



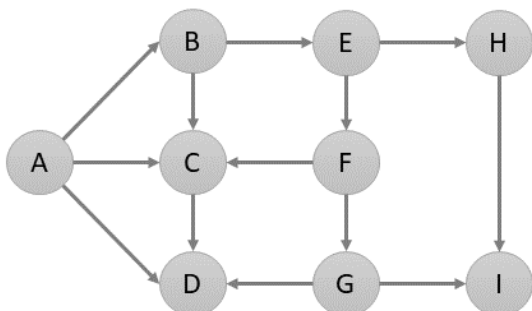
Exercise 2 [SECOND MODULE, theory]

Given a list L of n elements, please compute the asymptotic computational complexity of the following *func* function, explaining your reasoning.

```
def func(L):  
    n = len(L);  
    k = 0;  
    m = 2;  
    for i in range(n//2, n):  
        j = 2;  
        while j <= n:  
            k = k + n / 2;  
            j = j * m;  
    return k;
```

Exercise 3 [SECOND MODULE, theory]

Perform a Breadth-First Search (BFS) traversal on the graph below and list the nodes in the order they are visited.



Exercise 4.1 [SECOND MODULE, practical]

Slow Sort is a deliberately inefficient recursive sorting algorithm that follows a divide-and-conquer strategy. Unlike efficient sorting algorithms like Merge Sort or Quick Sort, Slow Sort makes unnecessary recursive calls, making it much slower. The algorithm repeatedly sorts halves of the list and swaps elements in a wasteful manner.

In this exercise you are required to implement the Recursive Sorting Function `_slow_sort()`. That will be used inside the already coded `sort` method. The template is provided in the python script [es4_1.py](#).

The `_slow_sort()` method is the core of the algorithm, which operates as follows:

1. Base Case:
 - * If the sublist has one or zero elements ($\text{left} \geq \text{right}$), it is already sorted, so return.
2. Recursive Case:
 - * Find the middle index: $\text{mid} = (\text{left} + \text{right}) // 2$
 - * Recursively call `_slow_sort()` on the first half (left to mid).
 - * Recursively call `_slow_sort()` on the second half (mid + 1 to right).
 - * Compare the last element of the first half (`arr[mid]`) with the last element of the second half (`arr[right]`).
 - * If they are out of order, swap them.
 - * Make a final recursive call on the entire range (left to right - 1) to propagate swaps.

Exercise 4.2 [SECOND MODULE, practical]

Given the provided Binary Tree Class in the [es4_2.py](#) script implement a method `to_sorted_list()` that returns the values of the nodes of the binary tree as a sorted list.

The script comes with a `unittest` class to test the code that you can use, do not modify the test class.

Built-in `sort` functions are not required and are not supposed to be used.

Hint: Remember the properties of a binary tree!