Task 1 - Optimise The Sorting Code

Objective

- You’ve got a messy list of products from an online store, and your job is to sort them like a pro. Make it fast, clean, and super efficient using Python and libraries - no excuses for messy code here!

Problem Statement

- You are given a list of products with the following attributes
- name: Name of the product.
- price: Price of the product (in USD).
- rating: Customer rating (out of 5).
- availability: Either "in_stock" or "out_of_stock."

- Sort the products efficiently based on these rules:
- Products that are in stock appear before those that are out of stock.
- Among products with the same availability, sort by descending rating.
- If ratings are tied, sort by ascending price.

Python Code

Code A - Unoptimised (Provided)

```
def custom_sort(products):
   for i in range(len(products)):
       for j in range(i + 1, len(products)):
           if products[i]["availability"] == "out_of_stock" and
products[j]["availability"] == "in_stock":
               products[i], products[j] = products[j], products[i]
           elif (
               products[i]["availability"] == products[j]["availability"]
               and products[i]["rating"] < products[j]["rating"]
           ):
               products[i], products[j] = products[j], products[i]
           elif (
               products[i]["availability"] == products[j]["availability"]
               and products[i]["rating"] == products[j]["rating"]
               and products[i]["price"] > products[j]["price"]
           ):
               products[i], products[j] = products[j], products[i]
   return products
```

Code B - Optimised

```python
def efficient_custom_sort(products):
   def sort_key(product):
       # Priority order:
       # 1. In-stock items first (availability: "in_stock" < "out_of_stock")
       # 2. Higher ratings come first (negative rating to sort in descending order)
       # 3. Lower prices come first
       return (
           product["availability"] == "out_of_stock",
           -product["rating"],
           product["price"]
       )

   return sorted(products, key=sort_key)
```

Key Differences between Unoptimised vs Optimised code

Aspect	Unoptimised	Optimised
Sorting Method	Nested loops and manual swaps	Built-in `sorted()` with a key function
Time Complexity	$O(n^2)$	O(n log n)
Practical Time (31 product rows)	99 µs	18 µs
Code Readability	Hard to follow, complex conditions	Clear, centralized sorting logic
Optimization Level	Inefficient for large datasets	Efficient and optimized for large datasets
Stability	Not guaranteed	Stable sort (preserves order of equal items)

Explaining Code Optimising

1. **Using `sorted()` Instead of Nested Loops**:
    - Code A manually swaps products in nested loops, which is inefficient. Code B uses the built-in `sorted()`function, which is optimized for sorting lists and avoids unnecessary comparisons.
    - `sorted()` ensures that the list is sorted in **O(n log n)** time, whereas the nested loops result in a time complexity of **$O(n^2)$**.
2. **Key Function for Sorting**:
    - In Code B, we define a **sort key** inside the function `sort_key()`, which is a tuple:
        - **First element**: A boolean indicating whether the product is out of stock (inverted so that "in_stock" comes first).
        - **Second element**: The negative rating, so products with higher ratings come first (since `sorted()` sorts in ascending order by default).
        - **Third element**: The price, which sorts in ascending order (lower prices first).
    - This approach simplifies the sorting conditions and makes the logic cleaner and easier to understand.