---

1. Log Analysis System

Scenario: A system that processes log files and finds the most frequent IP addresses.

```
from collections import Counter


def most_frequent_ips(logs):
    ip_counts = Counter(logs)  # Count occurrences of each IP
    return ip_counts.most_common(10)  # Get top 10 most common IPs
```

Time Complexity Analysis

- Counter(logs) iterates through the logs list (O(n)) and constructs a hash map.
- most_common(10) sorts the frequencies using heapq.nlargest(), which is O(k log n), where k=10.
- Total complexity: **O(n + k log n) ≈ O(n)** (since k is a small constant).

Commercial Use Case: This approach is widely used in log aggregation services like Splunk, ELK Stack, and Datadog.

---

2. E-Commerce Price Matching System

Scenario: A price comparison tool that matches product prices from different vendors.

```
def find_matching_prices(store_prices, competitor_prices):
    store_prices.sort()  # O(n log n)
    competitor_prices.sort()  # O(m log m)


    i, j = 0, 0
    matches = []


    while i < len(store_prices) and j < len(competitor_prices):
        if store_prices[i] == competitor_prices[j]:
            matches.append(store_prices[i])
            i += 1
            j += 1
```

```
        elif store_prices[i] < competitor_prices[j]:
            i += 1
        else:
            j += 1

    return matches
```

Time Complexity Analysis

- Sorting both lists: O(n log n + m log m)
- Two-pointer search through sorted lists: O(n + m)
- Total complexity: **O(n log n + m log m)**

Commercial Use Case: Used in price comparison tools like Google Shopping, Amazon, and Walmart's competitive pricing engines.

---

3. Fraud Detection in Financial Transactions

Scenario: Detect if a transaction pattern is suspicious (e.g., repeated payments to different accounts in a short time).

```
def detect_fraud(transactions, time_window):
    suspicious = set()
    transactions.sort(key=lambda x: x[1])  # Sort by timestamp (O(n log n))

    for i in range(len(transactions)):
        user, timestamp = transactions[i]
        j = i + 1
        while j < len(transactions) and transactions[j][1] - timestamp <= time_window:
            if transactions[j][0] == user:
                suspicious.add(user)
            j += 1

    return list(suspicious)
```

Time Complexity Analysis

- Sorting transactions: O(n log n)
- Nested while loop: O(n log n) (on average, assuming a balanced time distribution)
- Total complexity: **O(n log n)**

Commercial Use Case: This technique is used in fraud detection algorithms for payment services like PayPal, Stripe, and banks.

---

4. Social Media Feed Ranking

Scenario: A recommendation system that merges user-post interactions based on relevance.

```
import heapq


def merge_feeds(*feeds):
    merged_feed = []
    min_heap = []

    for feed in feeds:
        if feed:
            heapq.heappush(min_heap, (feed[0], 0, feed))  # Store (post, index, feed)

    while min_heap:
        post, idx, feed = heapq.heappop(min_heap)
        merged_feed.append(post)
        if idx + 1 < len(feed):
            heapq.heappush(min_heap, (feed[idx + 1], idx + 1, feed))

    return merged_feed
```

Time Complexity Analysis

- Using a min-heap to merge k sorted lists of n elements:
    - **Heap insertion/removal**: O(log k)

- **Total merges**: O(nk log k), where n is the average feed length and k is the number of feeds.

- Total complexity: **O(nk log k)**

Commercial Use Case: Used in personalized social media feeds (e.g., Facebook, Instagram, TikTok).

---

5. Dynamic Pricing in Online Retail

Scenario: A system that dynamically updates product prices based on demand.

```
import bisect


class DynamicPricing:
    def __init__(self):
        self.prices = []


    def add_price(self, price):
        bisect.insort(self.prices, price)  # O(log n) insertion


    def get_median_price(self):
        n = len(self.prices)
        if n % 2 == 0:
            return (self.prices[n // 2 - 1] + self.prices[n // 2]) / 2
        else:
            return self.prices[n // 2]


# Usage example
pricing = DynamicPricing()
pricing.add_price(100)
pricing.add_price(200)
pricing.add_price(150)
print(pricing.get_median_price())  # 150
```

Time Complexity Analysis

- bisect.insort(): O(log n) per insertion.
- Median retrieval: O(1).
- For n price updates, total complexity: **O(n log n)**.

Commercial Use Case: Used in airline ticket pricing, Uber's surge pricing, and e-commerce flash sales.

Summary of Complexity Results

Use Case	**Complexity**
Log analysis (top IPs)	O(n)
Price matching	O(n log n + m log m)
Fraud detection	O(n log n)
Social media feed merge	O(nk log k)
Dynamic pricing	O(n log n)