02 Log File Analysis & Reporting System

Project 2: Log File Analysis & Reporting System 📊🔍

Overview

Develop a command-line application that processes web server log files (e.g., Apache Common Log Format), parses relevant information, stores it in a MySQL database, and generates various analytical reports to understand web traffic patterns. This project simulates a core data engineering task: ingesting and analyzing semi-structured operational data.

Difficulty Level

Advanced | Estimated Time: 18-22 hours

Problem Statement

Create a CLI tool to ingest, process, and analyze web server log files. The application should be able to read large log files, extract structured data from each log entry, persist this data into a well-designed MySQL database, and then offer commands to query and generate reports on web traffic, user behavior, and server performance.

Learning Objectives

- Master parsing semi-structured text data using regular expressions.
- Design and implement an optimized relational database schema for log data.
- Efficiently load and manage large volumes of data in MySQL.
- Develop advanced SQL queries for data aggregation, filtering, and trend analysis.
- Build a robust and user-friendly command-line interface.
- Implement error handling for file I/O and database operations.
- Understand performance considerations for processing large files and database queries.

Core Requirements

1. Log File Ingestion & Parsing

- Read large web server log files line by line (e.g., Apache Common Log Format).
- Use regular expressions to extract key fields from each log entry:
  - IP Address
  - Timestamp
  - HTTP Method (GET, POST, etc.)
  - Requested URL Path
  - HTTP Status Code (200, 404, 500, etc.)
  - Bytes Sent
  - User Agent string
  - Referrer (optional but recommended)
- Handle malformed or incomplete log entries gracefully.

2. Data Transformation & Normalization

- Convert extracted string fields to appropriate Python data types (e.g., timestamp string to `datetime` object, bytes sent to `int`).
- Normalize data for efficient storage and querying in MySQL:

    - Separate user agent strings into a `user_agents` table to store distinct user agents and their parsed components (OS, browser, device).
    - Potentially normalize IP addresses or requested URLs if a large number of distinct values warrant it.
- Design a robust MySQL schema for log entries, considering indexing for common query patterns.

3. Data Loading (ELT)

- Connect to a MySQL database.
- Create database tables programmatically (DDL).
- Efficiently insert parsed and transformed log entries into the MySQL database.
- Implement strategies for handling large log files (e.g., batch inserts, handling millions of rows).
- Ensure idempotency for re-processing log files (avoiding duplicate inserts if a file is processed again).

4. CLI Interface & Reporting

- Develop a comprehensive CLI using `argparse` or `Click`.
- Commands for:
    - `process_logs <file_path>`: Parses and loads logs from a specified file.
    - `tail_logs <file_path>`: (Optional, Bonus) Continuously monitors a log file for new entries.
    - `generate_report <report_type> [options]`: Generates various reports.
        - `report_type examples`:
            - `top_n_ips <n>`: Top N requesting IP addresses.
            - `status_code_distribution`: Breakdown of HTTP status codes (2xx, 3xx, 4xx, 5xx).
            - `hourly_traffic`: Requests distribution by hour of day.
            - `top_n_pages <n>`: Top N most requested URLs.
            - `traffic_by_os`: Breakdown of traffic by operating system (using User Agent data).
            - `error_logs_by_date <date>`: List of error logs (4xx/5xx) for a specific date.
- Display reports in a clear, formatted, tabular output in the terminal.

Technical Specifications

Required Features

Core Log Processing & DB (60 points)

- [ ] **Log File Reading**: Efficiently reads large log files line by line.
- [ ] **Regex Parsing**: Accurately extracts IP, timestamp, method, URL, status, bytes, user agent.
- [ ] **Data Type Conversion**: Correctly converts extracted strings to Python types.
- [ ] **MySQL Schema Design**: Creates `log_entries` table and `user_agents` lookup table with appropriate keys and indexes.
- [ ] **Efficient Loading**: Uses `executemany` for batch insertion into MySQL.
- [ ] **Data Normalization**: Populates `user_agents` table and links to `log_entries` via foreign key.
- [ ] **Idempotency**: Logic to prevent duplicate log entries when reprocessing.
- [ ] **Error Handling**: Graceful handling of malformed log lines and DB errors.
- [ ] **Logging**: Informative logging of parsing progress and errors.

CLI & Reporting (25 points)

- [ ] **CLI Framework**: Uses `argparse` or `Click` for structured commands.
- [ ] **`process_logs` Command**: Successfully ingests and loads log files.

- ☐ **Report Commands**: Implements at least 5 different report types as specified above.
- ☐ **Flexible Reporting**: Reports allow parameters (e.g., `n` for top N, `date` for date filtering).
- ☐ **Clear Output**: Reports are presented in a well-formatted, readable table in the terminal.

Bonus Features (Extra 25 points)

- ☐ `tail_logs` **Command**: Continuously monitors and processes new log entries (simulated real-time).
- ☐ **Geolocalization**: Integrate with a free IP-to-Geolocation API (e.g., `ip-api.com` with rate limits) to add city/country data.
- ☐ **Advanced User Agent Parsing**: Use a dedicated Python library like `user-agents` for more detailed parsing.
- ☐ **Configuration File**: Manage database credentials and log file patterns in a config file (e.g., `config.ini`).
- ☐ **Performance Benchmarking**: Include basic metrics on processing speed and database insertion rate.
- ☐ **Data Retention Policy**: Implement a command to clean up old log data from the database.

Implementation Guidelines

Suggested Program Structure

```python
import re
import mysql.connector # or sqlalchemy
from datetime import datetime
import argparse # or click
import logging
from collections import Counter
from tabulate import tabulate # For nice table output

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

class LogParser:
    """Parses individual log lines using regex."""
    # Apache Common Log Format regex pattern
    LOG_PATTERN = re.compile(
        r'(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}) - - \[(.*?)\] "(.*?)" (\d{3}) (\d+) "(.*?)"
"(.*?)"'
    )
    # Group indices: 1=IP, 2=timestamp, 3=request, 4=status_code, 5=bytes, 6=referrer,
7=user_agent

    def parse_line(self, log_line):
        """Extracts structured data from a single log line."""
        match = self.LOG_PATTERN.match(log_line)
        if match:
            # Extract and convert types
            ip_address = match.group(1)
            timestamp_str = match.group(2)
            request = match.group(3)
            status_code = int(match.group(4))
            bytes_sent = int(match.group(5)) if match.group(5) != '-' else 0
            referrer = match.group(6) if match.group(6) != '-' else None
            user_agent = match.group(7) if match.group(7) != '-' else None

            # Further parse request (method, path)
            request_parts = request.split(' ')
            method = request_parts[0] if len(request_parts) > 0 else None
            path = request_parts[1] if len(request_parts) > 1 else None

            # Convert timestamp
            timestamp = datetime.strptime(timestamp_str, '%d/%b/%Y:%H:%M:%S %z')

            return {
                'ip_address': ip_address,
                'timestamp': timestamp,
                'method': method,
                'path': path,
                'status_code': status_code,
                'bytes_sent': bytes_sent,
                'referrer': referrer,
                'user_agent': user_agent
            }
        logging.warning(f"Malformed log line: {log_line.strip()}")
        return None

class MySQLHandler:
    """Manages MySQL database connections and operations."""
    def __init__(self, host, user, password, database):
        pass # Establish connection

    def create_tables(self):
        """Creates log_entries and user_agents tables."""
        pass # DDL statements

    def insert_log_entry(self, log_data):
        """Inserts a single parsed log entry, handling user agent normalization."""
        # Check if user_agent exists in user_agents table, if not, insert it
        # Then insert into log_entries using the user_agent_id
        pass

    def insert_batch_log_entries(self, log_data_list):
        """Efficiently inserts a batch of parsed log entries."""
        # For each log_data in log_data_list:
        #   Check/insert user_agent, get user_agent_id
        #   Prepare log_entry with user_agent_id
        # Use executemany for main log_entries table
        pass

    def get_top_n_ips(self, n):
        """Runs SQL query for top N IP addresses."""
        pass

    def get_status_code_distribution(self):
        """Runs SQL query for status code distribution."""
        pass

    # ... other reporting methods

    def close(self):
        """Closes the database connection."""
        pass

class CLIManager:
    """Manages the command-line interface."""
    def __init__(self, db_handler):
        self.db_handler = db_handler
        self.parser = argparse.ArgumentParser(description="Web Server Log Analyzer &
Reporting CLI")
        self._setup_parser()

    def _setup_parser(self):
        subparsers = self.parser.add_subparsers(dest='command', help='Available commands')

        process_parser = subparsers.add_parser('process_logs', help='Parse and load logs
from a file.')
        process_parser.add_argument('file_path', type=str, help='Path to the log file.')
        process_parser.add_argument('--batch_size', type=int, default=1000, help='Batch
size for DB inserts.')

        report_parser = subparsers.add_parser('generate_report', help='Generate various
analytical reports.')
        report_subparsers = report_parser.add_subparsers(dest='report_type', help='Types of
reports')

        top_ips_parser = report_subparsers.add_parser('top_n_ips', help='Top N requesting
IP addresses.')
        top_ips_parser.add_argument('n', type=int, default=10, help='Number of top IPs.')

        status_code_parser = report_subparsers.add_parser('status_code_distribution',
help='HTTP status code breakdown.')

        hourly_traffic_parser = report_subparsers.add_parser('hourly_traffic',
help='Traffic distribution by hour.')

        # ... add other report types

    def run(self):
        args = self.parser.parse_args()
        if args.command == 'process_logs':
            log_parser = LogParser()
            processed_count = 0
            batch = []
            with open(args.file_path, 'r', encoding='utf-8', errors='ignore') as f:
                for line in f:
                    parsed_data = log_parser.parse_line(line)
                    if parsed_data:
                        batch.append(parsed_data)
                        if len(batch) >= args.batch_size:
                            self.db_handler.insert_batch_log_entries(batch)
                            processed_count += len(batch)
                            batch = []
                            logging.info(f"Processed {processed_count} lines...")
                if batch: # Insert remaining
                    self.db_handler.insert_batch_log_entries(batch)
                    processed_count += len(batch)
                logging.info(f"Finished processing log file. Total lines loaded:
{processed_count}")

        elif args.command == 'generate_report':
            if args.report_type == 'top_n_ips':
                results = self.db_handler.get_top_n_ips(args.n)
                print(tabulate(results, headers=["IP Address", "Request Count"],
tablefmt="grid"))
            elif args.report_type == 'status_code_distribution':
                results = self.db_handler.get_status_code_distribution()
                print(tabulate(results, headers=["Status Code", "Count", "Percentage"],
tablefmt="grid"))
            # ... handle other report types
        else:
            self.parser.print_help()

def main():
    # Database configuration (should be in a config file)
    db_config = {
        'host': 'localhost',
        'user': 'root',
        'password': 'your_password', # Use environment variable or config file!
        'database': 'weblogs_db'
    }

    db_handler = MySQLHandler(**db_config)
    db_handler.create_tables() # Ensure tables exist

    cli_manager = CLIManager(db_handler)
    cli_manager.run()

    db_handler.close()

if __name__ == "__main__":
    main()
```

Required Files Structure

```
log_analyzer_cli/
├── main.py                     # Main application entry point and CLI setup
├── log_parser.py               # Module for log line parsing logic
├── mysql_handler.py            # Module for all MySQL operations and reporting queries
├── config.ini                  # Configuration file for DB credentials, log patterns
(optional but recommended)
├── requirements.txt            # Required Python packages
├── README.md                   # Project documentation
├── sample_logs/                # Directory for sample log files
│   └── access.log
├── sql/                        # Directory for SQL schema scripts
│   └── create_tables.sql
└── docs/                       # Optional: project design documents, regex details, etc.
```

Sample Application Flow

```
# First time setup (create tables)
python main.py process_logs --help # See available options
# Output: (help message)

# Process a sample log file
python main.py process_logs sample_logs/access.log --batch_size 500
```

```
# Output: Processing log file: sample_logs/access.log
# INFO - Processed 500 lines...
# INFO - Processed 1000 lines...
# INFO - Finished processing log file. Total lines loaded: 1250.

# Generate report: Top 5 requesting IP addresses
python main.py generate_report top_n_ips 5
# Output:
# +---------------+-----------------+
# | IP Address    | Request Count   |
# +---------------+-----------------+
# | 192.168.1.100 | 250             |
# | 203.0.113.45  | 180             |
# | 172.16.0.2    | 120             |
# | 10.0.0.5      | 90              |
# | 198.51.100.1  | 75              |
# +---------------+-----------------+

# Generate report: HTTP Status Code Distribution
python main.py generate_report status_code_distribution
# Output:
# +-------------+-------+------------+
# | Status Code | Count | Percentage |
# +-------------+-------+------------+
# | 200         | 900   | 72.0%      |
# | 404         | 250   | 20.0%      |
# | 500         | 70    | 5.6%       |
# | 302         | 30    | 2.4%       |
# +-------------+-------+------------+

# Generate report: Hourly Traffic
python main.py generate_report hourly_traffic
# Output:
# +-------+-----------------+
# | Hour  | Request Count   |
# +-------+-----------------+
# | 08:00 | 150             |
# | 09:00 | 220             |
# | 10:00 | 300             |
# | 11:00 | 280             |
# | 12:00 | 100             |
# +-------+-----------------+
```

Data Models

MySQL Table Design

`log_entries` table

Column Name	Data Type	Constraints	Description
`id`	`INT`	`PRIMARY KEY`, `AUTO_INCREMENT`	Unique log entry ID
`ip_address`	`VARCHAR(45)`	`NOT NULL`	Client IP address (IPv4/IPv6)
`timestamp`	`DATETIME`	`NOT NULL`	Exact time of the request
`method`	`VARCHAR(10)`	`NOT NULL`	HTTP method (GET, POST, etc.)
`path`	`VARCHAR(2048)`	`NOT NULL`	Requested URL path

Column Name	Data Type	Constraints	Description
`status_code`	`SMALLINT`	`NOT NULL`	HTTP response status code
`bytes_sent`	`INT`	`NOT NULL`	Bytes sent to client (0 for missing)
`referrer`	`VARCHAR(2048)`	`NULLABLE`	Referrer URL
`user_agent_id`	`INT`	`FOREIGN KEY` to `user_agents.id`	Link to user agent details
`created_at`	`TIMESTAMP`	`DEFAULT CURRENT_TIMESTAMP`	When record was loaded

`user_agents` table

Column Name	Data Type	Constraints	Description
`id`	`INT`	`PRIMARY KEY`, `AUTO_INCREMENT`	Unique User Agent ID
`user_agent_string`	`VARCHAR(512)`	`UNIQUE`, `NOT NULL`	Full User Agent string
`os`	`VARCHAR(100)`	`NULLABLE`	Operating System (e.g., 'Windows', 'Linux', 'iOS')
`browser`	`VARCHAR(100)`	`NULLABLE`	Browser (e.g., 'Chrome', 'Firefox', 'Safari')
`device_type`	`VARCHAR(50)`	`NULLABLE`	Device type (e.g., 'Desktop', 'Mobile', 'Tablet')
`created_at`	`TIMESTAMP`	`DEFAULT CURRENT_TIMESTAMP`	When record was added

Indexing Strategy:

- `log_entries.timestamp`: For time-based queries.
- `log_entries.ip_address`: For IP-based queries.
- `log_entries.status_code`: For status code analysis.
- `log_entries.path`: For top pages/URL analysis.
- `user_agents.user_agent_string`: For quick lookups during normalization.

Required Features Detail

1. Log Parsing with Regular Expressions

- Explain the chosen log format (e.g., Apache Common Log Format).
- Detail the regex pattern and how each group corresponds to a log field.
- Show how to handle `-` (dash) values for missing fields.

2. User Agent Normalization

```
# Conceptual logic within MySQLHandler for inserting log entries
def insert_log_entry(self, log_data):
    user_agent_string = log_data.get('user_agent')
    user_agent_id = None

    if user_agent_string:
        # Check if user agent already exists in user_agents table
        self.cursor.execute("SELECT id FROM user_agents WHERE user_agent_string = %s",
(user_agent_string,))
        result = self.cursor.fetchone()
        if result:
            user_agent_id = result[0]
        else:
            # Parse user agent string (simplified for example, can use 'user_agents'
library)
            os = "Unknown OS"
            browser = "Unknown Browser"
            device_type = "Unknown Device"

            # Simple parsing examples:
            if "Windows" in user_agent_string: os = "Windows"
            elif "Macintosh" in user_agent_string: os = "macOS"
            elif "Linux" in user_agent_string: os = "Linux"

            if "Chrome" in user_agent_string: browser = "Chrome"
            elif "Firefox" in user_agent_string: browser = "Firefox"
            elif "Safari" in user_agent_string: browser = "Safari"

            if "Mobile" in user_agent_string: device_type = "Mobile"
            elif "Tablet" in user_agent_string: device_type = "Tablet"
            else: device_type = "Desktop"


            # Insert new user agent
            self.cursor.execute(
                "INSERT INTO user_agents (user_agent_string, os, browser, device_type)
VALUES (%s, %s, %s, %s)",
                (user_agent_string, os, browser, device_type)
            )
            self.conn.commit()
            user_agent_id = self.cursor.lastrowid # Get the ID of the newly inserted row
            logging.info(f"New user agent '{user_agent_string}' added with ID:
{user_agent_id}")

    # Prepare data for log_entries table
    log_entry_values = (
        log_data['ip_address'], log_data['timestamp'], log_data['method'],
        log_data['path'], log_data['status_code'], log_data['bytes_sent'],
        log_data['referrer'], user_agent_id
    )
    # Insert into log_entries table (simplified, typically done in batch)
    self.cursor.execute(
        "INSERT INTO log_entries (ip_address, timestamp, method, path, status_code,
bytes_sent, referrer, user_agent_id) "
        "VALUES (%s, %s, %s, %s, %s, %s, %s, %s)",
        log_entry_values
    )
    self.conn.commit()
```

3. Reporting SQL Queries

```sql
-- SQL for top_n_ips
SELECT ip_address, COUNT(*) AS request_count
FROM log_entries
GROUP BY ip_address
ORDER BY request_count DESC
LIMIT %s;
```

```sql
-- SQL for status_code_distribution
SELECT status_code, COUNT(*) AS count,
       (COUNT(*) * 100.0 / (SELECT COUNT(*) FROM log_entries)) AS percentage
FROM log_entries
GROUP BY status_code
ORDER BY count DESC;

-- SQL for hourly_traffic
SELECT DATE_FORMAT(timestamp, '%H:00') AS hour_of_day, COUNT(*) AS request_count
FROM log_entries
GROUP BY hour_of_day
ORDER BY hour_of_day ASC;

-- SQL for traffic_by_os (requires join with user_agents)
SELECT ua.os, COUNT(le.id) AS request_count
FROM log_entries le
JOIN user_agents ua ON le.user_agent_id = ua.id
GROUP BY ua.os
ORDER BY request_count DESC;

-- SQL for error_logs_by_date
SELECT ip_address, timestamp, path, status_code, user_agent_string
FROM log_entries le
JOIN user_agents ua ON le.user_agent_id = ua.id
WHERE DATE(timestamp) = %s AND status_code >= 400
ORDER BY timestamp ASC;
```

Testing Checklist

Log Processing

- [ ] Correctly parses various valid log lines.
- [ ] Handles malformed log lines without crashing (skips or logs error).
- [ ] Efficiently processes large log files (e.g., 100MB+).
- [ ] Batch insertion works correctly.
- [ ] Duplicate log entries are avoided when processing the same file multiple times.

Database Operations

- [ ] MySQL tables are created successfully.
- [ ] Data types match the parsed data.
- [ ] `user_agents` table is correctly populated with unique user agents.
- [ ] Foreign key relationships are maintained.
- [ ] Database connection and closing are handled properly.

CLI & Reporting

- [ ] `process_logs` command works with `file_path` and `batch_size`.
- [ ] All `generate_report` types produce correct results.
- [ ] Report filtering (e.g., `top_n_ips <n>`, `error_logs_by_date <date>`) works.
- [ ] Output formatting is clean and easy to read.
- [ ] Error messages for invalid commands/parameters are clear.

Required Python Packages

Create `requirements.txt`:

```
mysql-connector-python>=8.0.30 # Or pymysql/SQLAlchemy
tabulate>=0.8.9 # For pretty table output
# click>=8.0.0 # If using Click instead of argparse
# user_agents>=2.2.0 # If using the advanced user agent parsing bonus
```

Install with: `pip install -r requirements.txt`

Evaluation Criteria

Criteria	Excellent (A)	Good (B)	Satisfactory (C)	Needs Work (D/F)
Log Parsing	Accurate regex, robust malformed line handling, efficient	Good parsing, minor issues with edge cases	Basic parsing works	Fails on common log formats
Data Transformation	Comprehensive normalization, efficient lookup/insert for user agents	Good normalization, minor inefficiencies	Basic data type conversion	Data not normalized or inconsistent
Data Loading	Highly efficient batch inserts, complete idempotency	Good loading performance, minor duplicate issues	Basic inserts work, slow for large files	Data integrity issues, very slow
Database Design	Optimized schema, well-indexed, thoughtful relationships	Good schema, some indexing missing	Basic table structure	Poorly designed, inefficient queries
CLI & Reporting	Intuitive CLI, diverse, accurate, and parameterized reports	Functional CLI, good reports	Basic CLI, few reports, some inaccuracies	Unusable CLI, incorrect/missing reports
Error Handling & Logging	Comprehensive error handling, detailed and useful logs	Good error messages, helpful logs	Basic try-except blocks	Errors crash application, no logs
Code Quality	Excellent modularity, clean code, well-documented	Good structure, readable, documented	Adequate organization, some docs	Poor structure, hard to maintain

Submission Requirements

What to Submit

1. **Source Code**: All Python files, organized into modules.
2. **SQL Scripts**: `create_tables.sql` file containing the DDL for the database schema (both `log_entries` and `user_agents`).
3. **Configuration File**: `config.ini` (or equivalent) for database credentials and maybe log format details.

4. **Requirements File**: `requirements.txt` with all dependencies.
5. **Documentation**: `README.md` with:
    - Installation and setup instructions (including MySQL setup).
    - Description of the log format assumed.
    - CLI command usage guide with examples.
    - Database schema diagram/description.
    - Regex pattern explanation.
    - Known limitations and future improvements.
6. **Sample Data**: A `sample_logs/access.log` file with at least 50-100 diverse log entries (including some malformed ones).
7. **Demo Screenshots**: Screenshots of CLI usage showing log processing and generated reports.

Testing Data

- Provide a sample `access.log` file that includes:
    - Various IP addresses.
    - Different HTTP methods (GET, POST).
    - A mix of status codes (200, 302, 404, 500).
    - Different user agent strings (browsers, OS).
    - Some malformed lines to test error handling.

Code Quality Requirements

- **Modularity**: Separate concerns into different Python modules (e.g., `log_parser.py`, `mysql_handler.py`).
- **Error Handling**: Implement `try-except` blocks for all file I/O, regex, and database operations.
- **Input Validation**: Validate CLI arguments and user inputs.
- **Documentation**: Use docstrings for all classes, methods, and complex functions.
- **Readability**: Follow PEP 8 guidelines for code style.

Advanced Challenges (Optional)

For students who finish early or want extra challenges:

1. **Dashboard Integration**: Create a simple HTML report or integrate with a tool like Metabase (connects to MySQL) to visualize the data.
2. **Alerting for Errors**: Set up a basic alerting mechanism (e.g., print a warning, or send a mock email) if a high volume of 4xx or 5xx errors are detected within a time window.
3. **Log File Rotation**: Simulate processing of multiple log files, possibly with a naming convention (e.g., `access_log.1`, `access_log.2`).
4. **Security Analysis**: Implement reports to detect suspicious activity (e.g., brute-force attempts from a single IP, scanning for common vulnerabilities).
5. **Multi-threaded/Async Processing**: For extremely large log files, explore `threading` or `asyncio` for parallel processing.

Common Pitfalls to Avoid

1. **Complex Regex**: Overly complex regex patterns can be hard to debug and maintain. Break them down if possible.
2. **Line-by-Line DB Inserts**: Inserting each log entry individually will be extremely slow for large files. Use `executemany`.

3. **Ignoring Database Indexes**: Without proper indexing, queries on large log datasets will be very slow.
4. **Memory Overload**: Reading entire large log files into memory at once can cause memory issues. Process line by line or in chunks.
5. **Timezone Issues**: Be mindful of timezones when parsing timestamps and performing time-based aggregations. Store timestamps in UTC.

Resources

- Python `re` (Regular Expression) Documentation
- MySQL Connector/Python Documentation
- Apache HTTP Server Log Files
- Python `datetime` Module
- The `tabulate` Library
- Database Indexing Best Practices

Academic Integrity

This project requires significant original implementation. While discussion of general approaches with classmates is encouraged, direct sharing of code is strictly prohibited. Proper attribution for any external code snippets or ideas is mandatory.

Happy parsing, and may your logs reveal insightful patterns! 📊 🔍